Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften

im Fachgebiet Informatik
am Fachbereich Mathematik/Informatik
der Universität Osnabrück

von

Diplom-Informatiker
Gregor Engels

Osnabrück
1986

Referent: Prof. Dr. M. Nagl (Universität Osnabrück)

Korreferent: Prof. Dr. H.J. Schneider (Universität Erlangen)

Tag der mündlichen Prüfung: 14. März 1986

Kurzfassung

Bei der Durchführung großer Softwareprojekte bieten **Software-Entwicklungsumgebungen** durch Werkzeuge geeignete Unterstützung für die einzelnen Aufgabenbereiche an. Im Projekt IPSEN (Incremental **P**rogramming **S**upport **En**vironment), in dessen Rahmen diese Arbeit entstanden ist, werden insbesondere die Aufgabenbereiche Programmieren-im-Kleinen, Programmieren-im-Großen, Dokumentationserstellung sowie Projektorganisation und -management unterstützt. Bei der Realisierung einer derartigen Software-Entwicklungsumgebung stellt sich für den Entwickler die Frage, wie die in den einzelnen Aufgabenbereichen zu bearbeitenden Software-Dokumente und zugehörigen werkzeugspezifischen Informationen intern repräsentiert werden. Wir zeigen in dieser Arbeit, daß zur Vermeidung von Redundanzen und dadurch entstehenden Konsistenzproblemen eine **einzige zentrale Datenstruktur** in jedem Aufgabenbereich gewählt werden kann. Weiterhin zeigen wir, daß **attributierte Graphen** sowohl auf konzeptioneller Ebene als auch bei der Realisierung ein adäquates und universelles Datenmodell darstellen.

Exemplarisch für den Aufgabenbereich des Programmierens-im-Kleinen erläutern wir, wie eine Klasse von **Modulgraphen** systematisch entwickelt werden kann. Diese Modulgraphen bilden die zentrale Datenstruktur zur Darstellung eines Moduls der in IPSEN unterstützten Programmiersprache Modula-2. Das heißt, daß neben kontextfreien und kontextsensitiven Beziehungen in einem Modula-2-Modul im zugehörigen Modulgraphen alle Informationen abgelegt werden, die die einzelnen, zum Bereich des Programmierens-im-Kleinen gehörenden Werkzeuge benötigen. Hierbei stellen wir insbesondere das Werkzeug **statische Analyse** zur Durchführung von Kontroll- und Datenflußanalysen sowie die Werkzeuge zur **Ausführung** und zum **Test** eines Moduls vor.

Die Benutzung einer einzigen Datenstruktur hat zur Folge, daß diese Modulgraphen eine sehr komplexe Struktur haben. Um auf der konzeptionellen Ebene diese Komplexität in den Griff zu bekommen, wurde eine angemessene Methode zur **Spezifikation** entsprechender Zugriffsoperationen entwickelt. Diese Methode beruht auf der Verwendung von **Graph-Ersetzungssystemen** zur Spezifikation des funktionalen Verhaltens von Zugriffsoperationen auf einen Graphen. Wir erläutern, wie auf systematische Art und Weise eine Spezifikation mit Hilfe von Graph-Ersetzungssystemen entwickelt werden kann, durch die eine Klasse von Graphen, hier die Modulgraphen, formal beschrieben wird.

Eine derartige Spezifikation mit Graph-Ersetzungssytemen dient weiterhin als Grundlage für die Entwicklung einer entsprechenden **Software-Architektur.** Wir erläutern die Komponenten dieser Architektur für die vorgestellten Werkzeuge und berichten über den derzeitigen Stand der **Implementierung** der Software-Entwicklungsumgebung IPSEN.

Gliederung

1 Einleitung

Um den ständig steigenden Aufwand und die damit verbundenen Kosten für die Entwicklung und Wartung von Softwaresystemen in den Griff zu bekommen, wird in zunehmendem Maße der Rechner selbst zur Unterstützung dieser Tätigkeiten herangezogen. Dies geschieht in erster Linie durch spezielle Programme, auch (Software-)**Werkzeuge** genannt, die dem Software-Entwickler zur Unterstützung seiner Tätigkeiten zur Verfügung stehen.
Herkömmliche Programmiersysteme, bestehend aus einer Ansammlung von Werkzeugen wie Editor, Compiler, Binder, Lader etc., haben sich hierbei als nicht ausreichend erwiesen. Sie haben häufig den Nachteil, daß alle Werkzeuge eine unterschiedliche Benutzerschnittstelle aufweisen. Weiterhin wird in der Regel nicht berücksichtigt, daß größere Softwaresysteme von einem Entwicklerteam und nicht von einem einzelnen Entwickler erstellt werden.

In den letzten zehn Jahren haben **Software-Entwicklungsumgebungen** in der Forschung immer stärkere Bedeutung erlangt. Diese haben u.a. das Ziel, derartige Nachteile zu vermeiden. Sie bestehen aus einer Menge integrierter Werkzeuge, die je nach Umgebung mehr oder weniger umfangreiche Unterstützung für die verschiedenen Phasen der Software-Entwicklung anbieten. Insbesondere präsentieren sich alle Werkzeuge mit einer einheitlichen Benutzerschnittstelle dem Software-Entwickler. Mittlerweile existieren eine ganze Reihe derartiger Software-Entwicklungsumgebungen, wie man den einschlägigen Tagungsbänden entnehmen kann (vgl. /Hu 81/, /He 84/, /Pr 85a/, /Pr 85b/). Beispiele für derartige Umgebungen sind Gandalf (/Ha 82/), Mentor (/DH 84/), Cornell Program Synthesizer (/Tr 81a/), PECAN (/Re 85/) und PSG (/Sn 85/). Allen diesen und ähnlichen Entwicklungsumgebungen ist gemeinsam, daß sie die Entwicklung von Programmen einer **anweisungsorientierten** Programmiersprache (z.B. Pascal, Modula-2) unterstützen. Viele der Charakteristika solcher Umgebungen findet man auch in Umgebungen, die die Entwicklung von Programmen funktionaler oder objektorientierter Programmiersprachen unterstützen (z.B. Interlisp /TM 81/, Smalltalk-80 /GR 83/). Auf derartige Umgebungen wollen wir aber im Rahmen dieser Arbeit nicht eingehen.

Der **Leistungsumfang** der einzelnen Software-Entwicklungsumgebungen ist sehr unterschiedlich. Einige von ihnen bestehen im wesentlichen nur aus einem, in der Regel syntaxgesteuerten Editor, andere bieten Werkzeuge an, die alle Tätigkeiten bei der Eingabe und beim Testen eines Programms unterstützen. Allerdings haben erst wenige dieser Umgebungen das Ziel, auch die Entwicklung sehr großer Softwaresysteme und die damit verbundenen Aufgaben zu unterstützen. Ansätze in dieser Richtung findet man z.B. in den Projekten Gandalf (/Ha 82/), SAGA (/KT 85/) bzw. MUPE-2 (/ML 85/).

Zu der Klasse von Software-Entwicklungsumgebungen, durch die die Entwicklung

großer Softwaresysteme unterstützt werden soll, gehört insbesondere das Projekt **IPSEN** (Incremental Programming Support **EN**vironment), in dessen Rahmen die hier vorliegende Arbeit entstanden ist. Aufbauend auf Ideen, die in /Sc 75/ und /Na 80/ vorgestellt wurden, wurde es 1981 an der Universität Osnbrück ins Leben gerufen. Die Zielvorstellungen für dieses Projekt wurden ausführlich in verschiedenen Veröffentlichungen diskutiert (z.B. /Na 85/, /EL 86/), so daß wir sie an dieser Stelle nur kurz skizzieren wollen.

Das Projekt IPSEN hat zum Ziel, eine Software-Entwicklungsumgebung zu entwerfen, durch die alle Tätigkeiten des sogenannten Programmierens-im-Großen und Programmierens-im-Kleinen sowie Organisations- und Management-Tätigkeiten durch entsprechende Werkzeuge unterstützt werden. Unter dem Begriff **Programmieren-im-Kleinen** werden dabei alle Tätigkeiten verstanden, die dazu dienen, einen einzelnen Modul zu erstellen und zu testen. Alle Tätigkeiten oberhalb der Ebene einzelner Moduln gehören zum **Programmieren-im-Großen.** Hierzu gehört z.B. die Erstellung eines Entwurfs, auch Software-Architektur genannt, in der festgelegt ist, welche Moduln existieren und wie sie zueinander in Beziehung stehen. Weiterhin gehört hierzu die Integration der einzelnen Moduln zu einem ablauffähigen Softwaresystem.

Neben der Entwicklung neuer konzeptioneller Überlegungen beim Entwurf derartiger Werkzeuge ist ein weiteres wichtiges Ziel des IPSEN-Projekts die Machbarkeit der Überlegungen an Hand einer **Prototypimplementation** auf einem Arbeitsplatzrechner der Größenordnung IBM XT/AT nachzuweisen.

Um eine einheitliche Umgebung zur Unterstützung der oben angesprochenen Aufgabenbereiche zu erhalten, werden eine Reihe von externen und internen Charakteristika für die zu realisierenden Werkzeuge gefordert. Das heißt im einzelnen, daß alle Werkzeuge der zu entwickelnden Umgebung, im folgenden auch mit IPSEN bezeichnet, die folgenden **externen Charakteristika** besitzen sollen:

- Alle Tätigkeiten eines IPSEN-Benutzers sind **inkrementorientiert** bzw. **syntaxgestützt.** Das heißt, daß im Gegensatz zu einer repräsentationsorientierten Arbeitsweise alle Tätigkeiten an syntaktischen Einheiten, sogenannten **Inkrementen,** des zu bearbeitenden Softwaredokuments orientiert sind. Das derzeit bearbeitete Inkrement wird **aktuelles Inkrement** genannt. Diese Vorgehensweise hat den Vorteil, daß nach einer Modifikation eines Inkrements unmittelbar die syntaktische Korrektheit des gesamten Softwaredokuments gewährleistet werden kann. Hierbei wird im IPSEN-Projekt unter **syntaktischer Korrektheit** sowohl die kontextfreie als auch kontextsensitive syntaktische Korrektheit verstanden.
- In der Regel werden die Tätigkeiten durch Eingabe eines Kommandos vom IPSEN-Benutzer initiiert. Diese **kommandogesteuerte** Vorgehensweise hat den Vorteil, daß stets sichergestellt werden kann, daß die gewünschte Tätigkeit des Benutzers in der aktuellen Situation erlaubt ist. In den Fällen, wo eine derartige Vorgehensweise zu unbequem für den IPSEN-Benutzer wäre, wird auch die sogenannte **freie Eingabe** ermöglicht.
- Der IPSEN-Benutzer hat jederzeit die Möglichkeit, ein in der aktuellen Situation gültiges Kommando eines beliebigen Werkzeugs zu aktivieren. Diese **integrierte** und **modifreie** Arbeitsweise ermöglicht den problemlosen Wechsel zwischen den einzelnen Werkzeugen.
- Alle Werkzeuge präsentieren sich in einer einheitlichen Benutzerschnittstelle dem IPSEN-Benutzer. Hierzu gehört insbesondere ein **Fenstersystem** mit der Möglichkeit sich überlappender Fenster. Es existieren vier Fenstertypen: Text-, Nachrichten-, Eingabe- und Menüfenster. In einem **Textfenster** wird ein Ausschnitt des aktuell bearbeiteten Softwaredokuments dargestellt. Einen derartigen Ausschnitt bezeichnen wir im folgenden auch als "**View**". Der Ausschnitt, auf den sich in der aktuellen Situation die Tätigkeiten des IPSEN-Benutzers beziehen, wird **aktueller View** genannt. Das im aktuellen View dargestellte aktuelle Inkrement wird durch eine besondere Darstellung (z.B. Fettschrift) gekennzeichnet. Im **Nachrichtenfenster** werden dem IPSEN-Benutzer Meldungen angezeigt, die er u.U. bestätigen muß. Ein **Eingabefenster** dient dazu, textuelle Eingaben des Benutzers einzulesen. Im **Menüfenster** wird dem Benutzer eine Auswahl der derzeit gültigen Kommandos angezeigt, von denen er eines selektieren kann. Für derartige und ähnliche Selektionsvorgänge steht ihm eine **Maus** als zusätzliches Eingabemedium zur Verfügung. Mit Hilfe der Maus hat der IPSEN-Benutzer z.B. auch die Möglichkeit, das Blättern in einem Textfenster anzustoßen oder ein neues aktuelles Inkrement zu selektieren.

Ein wichtiger Punkt bei der Realisierung einer Software-Entwicklungsumgebung ist eine geeignete Wahl einer Datenstruktur zur Abspeicherung der zu bearbeitenden Softwaredokumente. Während nahezu alle vergleichbaren Forschungsprojekte baumartige Datenstrukturen verwenden (z.B. /Ha 82/, /TR 81a/, /DH 84/, /Sn 85/), ist es ein wesentliches **internes Charakteristikum** von IPSEN, **graphartige Datenstrukturen** für alle zu bearbeitenden Softwaredokumente zu benutzen. Derartige Graphen bestehen aus einem abstrakten Syntaxbaum, der zusätzliche Kanten zur Darstellung kontextsensitiver und werkzeugspezifischer Beziehungen enthält. Ziel dieser Vorgehensweise ist, sämtliche **stukturellen** Beziehungen in einem Softwaredokument durch Knoten und Kanten darzustellen. **Nichtstrukturelle** Informationen werden in zusätzlichen Knotenattributen abgelegt. Je nach Aufgabenbereich haben diese Graphen eine unterschiedliche Gestalt: Einzelne Moduln werden durch **Modulgraphen** dargestellt, Software-Architekturen werden durch **Systemgraphen** dargestellt und z.B. technische Dokumentationen durch **Dokumentationsgraphen.**

Unter Berücksichtigung dieser Charakteristika wurde im bisherigen Verlauf des IPSEN-Projekts in erster Linie der Bereich des Programmierens-im-Kleinen

untersucht. Die dabei erzielten Ergebnisse werden in dieser Arbeit und in einer zweiten, mit dieser eng zusammenhängenden Dissertation (/Sc 86/) vorgestellt. Wir wollen im folgenden kurz die durch diese beiden Arbeiten abgedeckten Fragestellungen vorstellen und erläutern, welche Teilprobleme von welchem Autor untersucht wurden.

Ein Hauptziel der gesamten Untersuchungen war stets, einerseits neue Konzepte für die Entwicklung einer Software-Entwicklungsumgebung zu finden, die **unabhängig** sind von einer konkreten, zu unterstützenden **Programmiersprache** oder einer speziellen **Hardware**. Andererseits wurden alle konzeptionellen Überlegungen auch direkt angewendet, um zu einer lauffähigen **Prototypimplementation** zu gelangen.
Als die momentan zu unterstützende Programmiersprache im Bereich des Programmiens-im-Kleinen wurde **Modula-2** (/Wi 82/) ausgewählt. Aktuell zu bearbeitende Modula-2-Moduln werden intern durch Modulgraphen dargestellt. Hierzu wurde eine systematische Vorgehensweise entwickelt, wie ausgehend von der formalen Beschreibung der kontextfreien Syntax einer Programmiersprache durch eine EBNF ein **Erzeugendensystem** für die Klasse der Modulgraphen abgeleitet werden kann. Da die zu erzeugenden Datenstrukturen Graphen sind, boten sich **Graph-Grammatiken** als adäquates Beschreibungsmittel an (/Sc 77/, /Na 79/). Alle Werkzeuge des Programmierens-im-Kleinen benutzen einen derartigen Modulgraph als **gemeinsame** Datenstruktur. Die dadurch entstehende Komplexität der Zugriffsoperationen auf einen derartigen Modulgraphen muß vom Entwerfer einer Entwicklungsumgebung wie IPSEN bewältigt werden. Aus diesem Grunde wurde aufbauend auf dem Kalkül der Graph-Grammatiken eine Methode entwickelt, die es erlaubt, auf konzeptioneller Ebene derartige Zugriffsoperationen **formal zu spezifizieren.** Da hierbei im wesentlichen aus dem Bereich des Software-Engineering bekannte, ingenieurmäßige Methoden auf die Benutzung von Graph-Grammatiken übertragen wurden, nennen wir diese Methode **"Graph Grammar Engineering"** (/ES 85a/). Mit Hilfe dieser Methode sind prinzipiell alle Werkzeuge eines Aufgabenbereichs formal spezifizierbar. Bisher haben wir uns jedoch darauf beschränkt, das Verhalten des syntaxgestützten Editors mit Hilfe dieser Methode zu spezifizieren, da durch Aktivitäten dieses Werkzeugs die meisten der Modulgraphveränderungen durchgeführt werden. Die Spezifikation des Verhaltens der anderen Werkzeuge aus dem Bereich des Programmierens-im-Kleinen wurde zurückgestellt, da hierzu zunächst die Spezifikationsmethode weiter entwickelt werden sollte, um auch Graphalgorithmen einfach spezifizieren zu können. Weiterhin ist es erstrebenswert, den Spezifikationsvorgang durch Einsatz eines Graph-Grammatik-Editors durch den Rechner unterstützen zu lassen. Beide Aspekte werden momentan untersucht (/Qu 86/). Ein ausführliche Erläuterung dieser Spezifikationsmethode befindet sich in dieser Arbeit. In /Sc 86/ wird diese Methode angewandt, um den syntaxgestützten Editor für Modula-2 zu spezifizieren. Ein weiteres Anwendungsbeispiel befindet sich

in /Do 84/, in der diese Methode bei der Spezifikation eines Editors für die Programmiersprache Ada angewandt wurde.

Neben dem syntaxgestützten Editor werden weitere Werkzeuge im Bereich des Programmierens-im-Kleinen untersucht. Hierzu gehört ein Werkzeug zur **statischen Analyse**, mit dem es möglich ist, den im aktuell bearbeiteten Modul enthaltenen Kontroll- und Datenfluß zu untersuchen. Weitere Werkzeuge existieren zur Unterstüzung der **Ausführung** und des **Tests** des aktuell bearbeiteten Moduls. Da während der Ausführung umfangreiche Testunterstützungsmöglichkeiten angeboten werden sollen, die jederzeit an-/ausschaltbar bzw. veränderbar sein sollen, wurde eine **interpretative** Vorgehensweise für die Ausführung gewählt. Alle diese Werkzeuge benutzen ebenfalls die **gemeinsame** Datenstruktur Modulgraph. Aus diesem Grunde wurde zunächst auf konzeptioneller Ebene untersucht, welche weiteren Informationen zur Realisierung dieser Werkzeuge im Modulgraphen abzulegen sind. Wir beschreiben in dieser Arbeit ausführlich, wie alle diese Werkzeuge unter Benutzung der gemeinsamen, zentralen Datenstruktur Modulgraph realisiert werden können.

Alle diese Überlegungen liefen auf **konzeptioneller** Ebene, also der Spezifikationsebene ab. Um zu einer konkreten Implementierung zu kommen, mußte nun zunächst eine Software-Architektur entwickelt werden. Grundsätzliches Ziel hierbei war, eine Architektur zu entwickeln, bei der die Punkte **Adaptabilität an geänderte Anforderungen** und **Portabilität bei veränderter Hardware** berücksichtigt worden sind. In diesem Sinne sollte eine **Standardarchitektur** entstehen, die ohne Aufwand übertragbar ist bei einer Realisierung der Werkzeuge der anderen Aufgabenbereiche bzw. bei der Entwicklung ähnlicher Softwaresysteme. Auf diesen Aspekt wird ausführlich in /Sc 86/ eingegangen. Die so entstandene Architektur wollen wir schematisch an Hand der Hauptbestandteile skizzieren (vgl. Abb 1.1):

Graphen werden in IPSEN nicht nur auf konzeptioneller Ebene, sondern auch auf der Implementierungsebene als zentrale Datenstrukturen benutzt. Aus diesem Grunde wurde ein relationales Datenbanksystem (GraphDataBase) entwickelt, in dem beliebige Graphen abgespeichert werden können (/BL 85/). Dieses Datenbanksystem, auch **Graphenspeicher** genannt, bildet die Projektdatenbasis im IPSEN-System. Aufbauend auf diesem Graphenspeicher kann ein Teilsystem realisiert werden (Module-, System-, Documentation-Graph), durch das spezielle Graphklassen verkapselt werden. In /Sc 86/ wird gezeigt, daß die Feinstruktur dieses Teilsystems systematisch aus der vorher erstellten Graph-Grammatik-Spezifikation abgeleitet werden kann. Die für die Ausführung eines Modulgraphen benötigten **Laufzeitdaten** (RuntimeData) liegen wie üblich im Hauptspeicher der Ausführungsmaschine. Das gesamte **IPSEN-spezifische Ein-/Ausgabesystem** (I/O-System) wird in einem Teilsystem verkapselt. Auch dieser Anteil von IPSEN, also die gesamte Realisierung der

Benutzerschnittstelle wird in /Sc 86/ beschrieben. Oberhalb der IPSEN-spezifischen Datenstrukturen liegen eine Reihe von **Transformationsbausteinen** (Transformers). Hierzu gehören zum Beispiel der Unparser, der aus einer Modulgraphdarstellung eine textuelle Darstellung erzeugt, und der Parser für die umgekehrte Abbildung. Während der Parser in /Sc 86/ beschrieben wird, ist die Darstellung des Unparsers Bestandteil dieser Arbeit. In den oberen Schichten liegen die Bausteine für die **Steuerung** der verschiedenen auf dem Bildschirm liegenden Fenster (ViewManager) und die Ablaufsteuerung der Werkzeuge für die verschiedenen Aufgabenbereiche in IPSEN (Abk.: PiS - Programming-in-the-Small/ PiL - Programming-in-the-Large). Während die Beschreibung der Ablaufsteuerung der einzelnen Werkzeuge in dieser Arbeit zu finden ist, wird in /Sc 86/ die Gesamtsteuerung dargestellt.

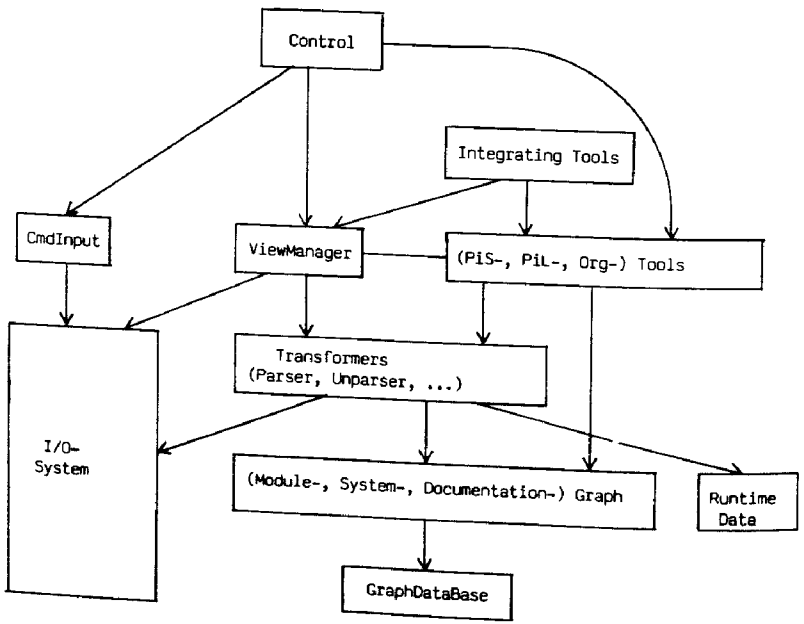


Abb. 1.1: Überblick über die IPSEN-Architektur

Die etwas unsystematische Aufteilung der in den beiden Dissertationen behandelten Themenbereiche ist in erster Linie dadurch begründet, daß erst im Laufe des Projektfortschritts erkannt wurde, welche konkreten Themengebiete zu bearbeiten sind. Die daraufhin ausgeschriebenen Diplomarbeitsthemen und die damit verbundene Zuteilung an einen Betreuer geschah dann mehr nach Arbeitsbelastungsgesichtspunkten als nach inhaltlichen Gesichtspunkten. Andererseits hat dies den Vorteil, daß beide Arbeiten sowohl konzeptionelle, stärker **theoretische Anteile**

enthalten als auch Implementierungs-, also **praktische Anteile.** Außerdem stellt sich die Frage, inwieweit eine disjunkte Zerlegung der Themenbereiche möglich gewesen wäre, da doch ein wesentliches Charakteristikum von IPSEN die Integration, d.h. Verzahnung aller Tätigkeiten ist.

Kapitelübersicht

Die vorliegende Arbeit ist in 10 Kapitel unterteilt:

Im **zweiten** Kapitel wird im Stil einer **Anforderungsdefinition** aus der Sicht eines IPSEN-Benutzers erläutert, welche Kommandos von den Werkzeugen "statische Analyse", "Testvorbereitung" bzw. "Ausführung und Test" im Bereich des Programmierens-im-Kleinen zur Verfügung gestellt werden. Zu jedem Kommando wird seine Funktionalität erläutert. Das Kapitel schließt mit einem umfangreicheren Beispiel ab, in dem das Zusammenspiel der verschiedenen Werkzeuge in der Testphase erläutert wird.

Das **dritte** Kapitel erläutert die bereits erwähnte Spezifikationsmethode des **Graph Grammar Engineering.** Es wird ein Erzeugendensystem für die Klasse der Modulgraphen systematisch hergeleitet. An Beispielen wird erläutert, wie Graph-Grammatiken durch den IPSEN-Entwerfer zur Spezifikation von Werkzeugaktivitäten eingesetzt werden können. Da Graph-Grammatiken bzw. Graph-Ersetzungssysteme eine operationale, d.h. ausführbare Spezifikation darstellen, wird kurz auf die Möglichkeit eingegangen, Graph-Grammatik-Spezifikationen im Sinne eines 'rapid prototyping' auszuführen.

Im **vierten** Kapitel werden verschiedene Möglichkeiten diskutiert, **wie** die einzelnen Werkzeuge ihre spezifischen Informationen im Modulgraphen ablegen können.

Im **fünften** Kapitel wird erläutert, welche zusätzlichen Informationen im Modulgraphen abzulegen sind, damit die Kommandos des Werkzeugs der **statischen Analyse** effizient realisiert werden können.

Im **sechsten** Kapitel werden zwei verschiedene Strategien für die Realisierung eines **Unparsers** vorgestellt, eine tabellengesteuerte Version und eine Version, die nach dem Prinzip des rekursiven Abstiegs arbeitet.

Im **siebten** und **achten** Kapitel wird erläutert, wie der **Interpreter** realisiert ist, der alle Testunterstützungsmöglichkeiten des IPSEN-Systems berücksichtigt. Es wird gezeigt, wie auch hier der Modulgraph als zentrale Datenstruktur für die Realisierung aller **Testunterstützungskommandos** eingesetzt werden kann.

Während alle bisherigen Erläuterungen auf konzeptioneller Ebene durchgeführt wurden, wird im **neunten** Kapitel die konkrete **Software-Architektur** des IPSEN-Systems erläutert. Für die in dieser Arbeit auf konzeptioneller Ebene vorgestellten Realisierungsmöglichkeiten wird gezeigt, wo sich die zugehörigen Moduln zur tatsächlichen Realisierung in der Architektur befinden. Am Ende dieses Kapitels wird

ein kurzer Überblick über den jetzigen Stand der Implementierung gegeben.
Im **zehnten** Kapitel fassen wir noch einmal die wesentlichen **Ergebnisse dieser Arbeit** zusammen und geben Hinweise, welche **offenen Probleme** unserer Meinung nach Gegenstand zukünftiger Forschung im IPSEN-Projekt sein sollten.
Am **Ende der Arbeit** befindet sich ein Literaturverzeichnis, ein Stichwortverzeichnis sowie ein Verzeichnis der verwendeten Abkürzungen für Knoten- bzw. Kantenmarkierungen.
Im **Anhang A** wird eine **EBNF** für die Programmiersprache Modula-2 angegeben, die in dieser Arbeit angegebene Eigenschaften erfüllt und als Ausgangsbasis für eine systematische Herleitung eines Erzeugendensystems für die Klasse der Modulgraphen dient.

Danksagungen

Diese Arbeit wurde betreut von Prof. Dr. M. Nagl, dem Initiator des Projekts IPSEN. Für die intensive Betreuung, seine ständige Diskussionsbereitschaft und zahlreiche, stundenlange Gespräche möchte ich ihm an dieser Stelle danken.

Herrn Prof. Dr. H. J. Schneider (Erlangen) danke ich für die Übernahme des Zweitgutachtens.

Mein besonderer Dank gilt all denen, die mit dazu beigetragen haben, daß in unserer Arbeitsgruppe eine Atmosphäre herrscht, in der das Arbeiten fast immer Spaß gemacht hat. Das gilt vor allem für meine beiden Kollegen Claus Lewerentz und Wilhelm Schäfer, mit denen ich viele interessante und kreative Gespräche geführt habe, aber auch für alle anderen Mitglieder des IPSEN-Projekts, nämlich Chr. Beer, Th. Brandes, F. Erdtmann, H. Haverkamp, Th. Janning, A. Sandbrink, U. Schleef, P. Tillmann, B. Westfechtel und D. Zerulla.
C. Lewerentz bzw. D. Zerulla danke ich außerdem für das "semantische" bzw. "syntaktische" Korrekturlesen dieser Arbeit und für Ihre wertvollen Hinweise.

Meiner Frau Johanna und meinen Kindern Sarah und Alexander möchte ich für die große Geduld und das Verständnis danken, das sie mir vor allem während des gesamten letzten Jahres entgegengebracht haben. Die emotionale Unterstützung, die sie mir in den wenigen gemeinsamen Stunden gegeben haben, hat mir bei der Fertigstellung dieser Arbeit sehr geholfen.

2 Anforderungen an die Werkzeuge des Programmierens-im-Kleinen

Zum Aufgabenbereich des Programmierens-im-Kleinen zählen wir in IPSEN alle Aktivitäten zur Bearbeitung eines einzigen Moduls. Hierbei verstehen wir im Bereich des Programmierens-im-Kleinen unter einem Modul einen Programm-Modul oder einen "implementation module" im Modula-2-Sinn. Zur Bearbeitung im Bereich des Programmierens-im-Kleinen gehören nun einerseits Aktivitäten zur Erstellung, Modifikation und Analyse eines Modula-2-Moduls und andererseits Aktivitäten zur Ausführung und zum Test eines Modula-2-Moduls. Alle diese Aktivitäten können vom IPSEN-Benutzer durch entsprechende Kommandos aktiviert werden. Um die Vielzahl der möglichen Aktivitäten sowohl für den IPSEN-Benutzer als auch für den IPSEN-Entwerfer übersichtlicher zu machen, werden logisch zusammengehörende Aktivitäten in einzelnen **Werkzeugen** zusammengefaßt (vgl. Abb. 2.1):
Alle Aktivitäten zur Erstellung und Modifikation eines Modula-2-Moduls gehören zum **syntaxgestützten Editor.** Umfangreiche Analysen des aktuell bearbeiteten Modula-2-Moduls ermöglichen die Kommandos des Werkzeugs der **statischen Analyse.** Für die Ausführung eines Modula-2-Moduls ermöglichen die Kommandos des **Ausführungs- und Testwerkzeugs** die Festlegung unterschiedlicher Ausführungsarten. Insbesondere wird die Möglichkeit angeboten, die Ausführung eines Moduls an bestimmten Stellen zu unterbrechen. An diesen Unterbrechungspunkten besteht dann durch weitere Kommandos die Möglichkeit, sich über den Stand der Ausführung zu informieren bzw. den weiteren Verlauf der Ausführung zu steuern. Zur Festlegung spezifischer Testumgebungen hat der IPSEN-Benutzer außerdem die Möglichkeit, das aktuell bearbeitete Modula-2-Programm vor der Ausführung mit Hilfe der Kommandos des Werkzeugs **Testvorbereitung** zu instrumentieren. Die dadurch eingefügten, zusätzlichen Informationen in einem Modula-2-Modul bewirken dann unterschiedliche Ausführungsarten oder die implizite Aktivierung weiterer Aktivitäten während der Ausführung.

Werkzeuge des Programmierens-im-Kleinen

- syntaxgestützter Editor
- statische Analyse
- Testvorbereitung
- Ausführung und Test

Abb. 2.1: Werkzeugübersicht

Es ist Ziel dieses Kapitels, bis auf die Erläuterung des syntaxgestützten Editors, zu allen oben genannten Werkzeugen exemplarisch den Leistungsumfang einiger Kommandos vorzustellen. Hierbei werden wir weniger Gewicht auf die Darstellung

der zugehörigen Benutzerschnittstelle als auf die Funktionalität der einzelnen Kommandos legen. Sowohl die Darstellung der in IPSEN realisierten Benutzerschnittstelle als auch eine umfassende Darstellung des zu IPSEN gehörenden syntaxgestützten Editors befindet sich in /Sc 86/.

Die Existenz verschiedener Werkzeuge bedeutet für den IPSEN-Benutzer nicht, daß der Wechsel von einem Werkzeug zu einem anderen nur durch aufwendige Modiwechsel möglich ist. Prinzipiell ist jederzeit jedes derzeit gültige Kommando vom IPSEN-Benutzer aktivierbar. Eine ausführliche Beschreibung der verschiedenen Kommandoeingabemöglichkeiten befindet sich in /Sc 86/. Wir beschränken uns im folgenden darauf, für jedes vorgestellte Kommando die zugehörige **Kurzkommandobezeichnung** anzugeben. Diese besteht aus maximal drei Buchstaben, wobei der erste Buchstabe stets das entsprechende Werkzeug kennzeichnet und der zweite bzw. dritte Buchstabe die eigentliche Aktivität. Da alle Kommandos und Meldungen in IPSEN derzeit in englischer Sprache sind, kann einer der folgenden vier Buchstaben an erster Stelle in einer Kurzkommandobezeichnung stehen:

E - syntax-aided Editor
A - static Analysis
T - Testing preparation
X - eXecution

Wir werden im nächsten Paragraphen eine kurze Übersicht über den in IPSEN eingesetzten syntaxgestützten Editor geben, so weit dies für das weitere Verständnis der Arbeit nötig ist. Im Paragraphen 2.2 werden dann einige Kommandos des Werkzeugs der statischen Analyse erläutert. In den Paragraphen 2.3 und 2.4 erläutern wir die Kommandos der Werkzeuge zum Ausführen und Testen eines Modula-2-Moduls. Jeder Paragraph wird mit einer zusammenfassenden Auflistung der neu eingeführten Kommandos abgeschlossen. Im Paragraphen 2.5 zeigen wir dann an einem ausführlichen Beispiel, in welcher Form die vorgestellten Kommandos vom IPSEN-Benutzer benutzt werden können, um einen fehlerfreien Modula-2-Modul zu erstellen.

2.1 Syntaxgestützter Editor

Der Einsatz eines syntaxgestützten, kommandogesteuerten Editors bedeutet für den IPSEN-Benutzer, daß sich alle seine Aktivitäten nicht an der textuellen, zeilenorientierten Repräsentation eines Moduls orientieren, sondern bezogen sind auf die dem aktuell bearbeiteten Modul zugrundeliegende syntaktische Struktur. Da jedoch eine vollständige, bis auf Zeichenebene gehende Einteilung eines Moduls in Inkremente eine sehr aufwendige Benutzerschnittstelle eines Editors bedeuten würde, ist die Einteilung eines Moduls in sogenannte **Edierinkremente** gröber als die durch

die zugrundeliegende Syntax gegebene Inkrementstruktur (vgl. insbesondere hierzu /Sc 86/). Dies gilt insbesondere für die Eingabe und Modifikation von arithmetischen Ausdrücken, Variablen und Bezeichnern, die der Benutzer wie bei einem bildschirmorientierten Editor in der sogenannten **freien Eingabe** edieren kann.

Bei einem Vergleich der in IPSEN verfolgten inkrementorientierten, kommandogesteuerten Arbeitsweise des Editors mit einem herkömmlichen zeilen- oder bildschirmorientierten Editor zeigen sich die folgenden Vorteile:

- Da dem System stets die syntaktische Struktur des aktuellen Inkrements bekannt ist, kann sichergestellt werden, daß durch das vom Benutzer eingegebene Kommando der aktuell bearbeitete Modul kontextfrei korrekt bleibt.
- Außerdem können auch sofort die zugehörigen kontextsensitiven Regeln überprüft werden, um ebenso stets die kontextsensitive Korrektheit des bis dahin erstellten Modula-2-Moduls zu garantieren.
- In der textuellen Repräsentation kann schließlich automatisch sämtliche konkrete Syntax erzeugt werden. Die noch existierenden Lücken im aktuellen Modul werden durch entsprechend bezeichnete Platzhaltersymbole gekennzeichnet. Abb. 2.1.1 gibt ein Beispiel eines Ausschnitts aus der textuellen Darstellung eines teilweise erstellten Modula-2-Moduls:

```
...
WITH ActStack DO
  IF (< Expression >) THEN
     Last := Last + 1
  (< ElsifPart >)
  (< ElsePart >)
  END;
END;
...
```

Abb. 2.1.1: Auszug aus einem Modula-2-Modul

Da bei der Änderung größerer Programmteile eine kommandogesteuerte Vorgehensweise sehr umständlich sein kann, wird dem IPSEN-Benutzer z.B. für derartige Tätigkeiten zusätzlich die **freie Eingabe** angeboten. Hierbei hat der Benutzer dann die Möglichkeit, die textuelle Repräsentation des aktuellen Inkrements wie bei einem herkömmlichen bildschirmorientierten Editor zu modifizieren. Nach Abschluß der freien Eingabe wird dieses Textstück vom Parser analysiert und dem Benutzer etwaige Verletzungen der kontextfreien oder kontextsensitiven Syntax mitgeteilt. Weitergehende Erläuterungen der freien Eingabe und des Parsers befinden sich in der Diplomarbeit /Sl 86/ bzw. in /Sc 86/.

2.2 Statische Analyse

Nach jeder Veränderung eines Inkrements durch ein Kommando des oben beschriebenen syntaxgestützten Editors ist sichergestellt, daß der vorliegende Programmtext einen Ausschnitt eines kontextfrei und kontextsensitiv korrekten Modula-2-Moduls darstellt. Neben dieser rein programmiersprachlichen Korrektheit sollte ein Modula-2-Modul weitere Charakteristika besitzen, um ein qualitativ hochwertiges Softwareprodukt darzustellen. Hierzu zählen:

- **Konsistenz zwischen Deklarations- und Anweisungsteil:** Das heißt z.B., daß jede Deklaration auch angewandt wird.
- **Minimalität:** Das heißt z.B., daß im Programmtext keine Anweisungen existieren, die nie ausgeführt werden.
- **Laufzeitsicherheit:** Damit ist z.B. gemeint, daß zu jeder deklarierten Variablen mindestens eine Initialisierungsanweisung existiert.

Alle diese Charakteristika sind **weitergehende kontextsensitive Regeln,** die in einem (Modula-2-) Modul erfüllt sein sollten. Im Gegensatz zu den zur Programmiersprache Modula-2 gehörenden kontextsensitiven Regeln werden diese jedoch nicht sofort bei jeder Veränderung eines Inkrements durch den syntaxgestützten Editor überprüft. Eine derartige inkrementelle Überprüfung dieser zusätzlichen kontextsensitiven Regeln wäre auch nicht angebracht, da viele dieser Regeln sinnvollerweise erst dann überprüft werden sollten, wenn das zu analysierende Inkrement (z.B. Modul, Prozedur) ganz oder zumindest größtenteils erstellt worden ist. Aus diesem Grunde werden derartige Analysen nur auf expliziten Wunsch des IPSEN-Benutzers durchgeführt. Hierzu stehen ihm die Kommandos eines Werkzeugs zur Verfügung, das wir **statische Analyse** genannt haben. Dieser Name wurde gewählt, weil viele der hier vorgestellten Analysen auch in der Analysephase von optimierenden Compilern, also zur Compilezeit, durchgeführt werden (vgl. /AU 79/). Während bei einer compilativen Vorgehensweise diese Informationen dem Programmierer nicht zugänglich sind, kann der IPSEN-Benutzer sich derartige Analyseergebnisse anzeigen lassen.

Neben dem Zeitpunkt der Analyse hat der IPSEN-Benutzer weiterhin die Möglichkeit, auch den Umfang und den Verlauf der Analyse im Dialog zu beeinflussen. Das bedeutet z.B., daß bei einer bestimmten Analyse eines gesamten Moduls die Analyse auf einzelne modullokale Prozeduren nur auf expliziten Wunsch des IPSEN-Benutzers ausgedehnt wird. Wir werden auf diese **dialoggesteuerte** Vorgehensweise bei der Analyse bei der Vorstellung der einzelnen Kommandos in diesem Paragraphen zurückkommen.

Bei der Ausführung eines Analysekommandos bekommt der IPSEN-Benutzer mitgeteilt, ob und an welchen Programmstellen eine bestimmte Regel verletzt wird. Dieses Ergebnis hat für den Benutzer stets nur **empfehlenden** Charakter, d.h. er muß selbst entscheiden, inwieweit er anschließend den Programmtext mit Hilfe des

syntaxgestützten Editors verändert. Wir halten es nicht für sinnvoll, solche Modultextänderungen automatisch durchzuführen, da z.B. bei noch nicht vollständig erstellten Moduln vom Benutzer bewußt gegen einige dieser zusätzlichen kontextsensitiven Regeln verstoßen wird. So ist es sicherlich nicht sinnvoll, bei noch unvollständigen Moduln automatisch alle Deklarationen von noch nicht angewandten Datenobjekten zu löschen.

Neben der Unterstützung des Benutzers zur Erstellung qualitativ hochwertiger Software können die Kommandos der statischen Analyse auch dazu benutzt werden, den IPSEN-Benutzer bei der Suche nach der Ursache von bei der Ausführung eines Programms festgestellten Fehlern zu unterstützen. Denn da bei vielen der zusätzlichen kontextsensitiven Regeln der im gegebenen Modul verkapselte Kontroll- und Datenfluß untersucht werden muß, sind die dabei ermittelten Informationen auch für das Auffinden der **Ursache von Laufzeitfehlern** hilfreich. Wir kommen auf diesen Punkt bei der folgenden Vorstellung der einzelnen Analysekommandos zurück.

Bei den zu einer Programmiersprache gehörenden kontextsensitiven Beziehungen innerhalb eines Programms kann zwischen den **drei** folgenden **Arten** unterschieden werden:

a) kontextsensitive Beziehungen zwischen Deklarations- und Anweisungsteilen,
b) kontextsensitive Beziehungen innerhalb von Deklarationsteilen,
c) kontextsensitive Beziehungen innerhalb von Anweisungsteilen.

Beispiele für a) sind, daß zu allen im Anweisungsteil auftretenden Bezeichnern in einem entsprechenden Deklarationsteil eine zugehörige Deklaration existiert, für b), daß in Typdeklarationen auftretende Typbezeichner für Komponententypen an anderer Stelle im Deklarationsteil deklariert sind, und für c), daß zu einem benutzenden Auftreten einer Variablen in einem arithmetischen Ausdruck an einer anderen Stelle ein setzendes Auftreten, z.B. auf der linken Seite einer Wertzuweisung existiert.
Im nächsten Abschnitt werden Kommandos vorgestellt, die zusätzliche Regeln zu den unter a) und b) angesprochenen kontextsensitiven Beziehungen überprüfen. Die unter Punkt c) angesprochenen kontextsensitiven Beziehungen betreffen in erster Linie den dort verkapselten Datenfluß. Dieser wiederum hängt eng zusammen mit dem im Anweisungsteil verkapselten Kontrollfluß, der durch die vorliegende Reihung und Ineinanderschachtelung verschiedener Kontrollstrukturen gegeben ist. Kommandos zur Untersuchung des Kontroll- bzw. Datenflusses in einem Modul werden im Abschnitt 2.2.2 vorgestellt.

Da der Schwerpunkt dieser Arbeit nicht auf der Darstellung der Benutzerschnittstelle von IPSEN liegt, wird bei der Erläuterung der einzelnen Kommandos die zugehörige Benutzerschnittstelle nur angedeutet.

2.2.1 Analyse von Deklarationen

Die meisten der üblicherweise zu einer Programmiersprache mit Typkonzept (wie z.B. Modula-2) gehörenden kontextsensitiven Regeln betreffen die kontextsensitiven Beziehungen zwischen Deklarations- und Anweisungsteilen bzw. innerhalb von Deklarationsteilen. Diese Regeln fordern in erster Linie, daß alle angewandten Bezeichner geeignet deklariert sind. Um die oben angesprochene Minimalität eines Programmtextes und auch Transparenz zu erzielen, ist es jedoch andererseits auch nicht sinnvoll, daß im Programmtext Deklarationen existieren, die an keiner anderen Stelle angewandt werden. Derartige Deklarationen könnten aus dem Programmtext gestrichen werden, ohne daß dadurch die Semantik des Programms geändert würde. Zum Auffinden solcher nicht angewandter Deklarationen dient das Kommando **An - non-applied (declarations/imports)** des Werkzeugs statische Analyse. Dieses Kommando kann vom IPSEN-Benutzer aktiviert werden, wenn das aktuelle Inkrement eine einzelne Deklaration oder eine einzelne importierte Ressource ist. Nach Aufruf des Kommandos wird dann für das aktuelle Inkrement untersucht, ob es an einer anderen Stelle im Programmtext angewandt wird. Das bedeutet a) für Konstanten, daß sie in einem Ausdruck benutzt werden, b) für Typdeklarationen, daß der Typbezeichner in einer anderen Typdeklaration benutzt wird oder ein Datenobjekt dieses Typs deklariert wird, c) für Datenobjektdeklarationen bzw. formale Parameter, daß ein benutzendes oder setzendes Auftreten dieses Datenobjekts stattfindet, oder d) eine deklarierte Prozedur aufgerufen wird.

Dieses Kommando kann insbesondere dazu benutzt werden, die **Schnittstelle zwischen dem Programmieren-im-Großen und Programmieren-im-Kleinen** zu untersuchen: Die während des Programmierens-im-Großen vom IPSEN-Benutzer für eine Realisierung eingetragenen importierten Ressourcen werden zu Beginn des Programmierens-im-Kleinen vom IPSEN-System automatisch in die Import-Liste eines Moduls aufgenommen (vgl. /Ja 86/). Mit Hilfe des obigen Kommandos kann überprüft werden, ob alle vom Programmieren-im-Großen vorgegebenen Ressourcen beim Programmieren-im-Kleinen auch wirklich angewandt werden. Daneben können auf diese Weise durch mehrfache Änderungen des Quelltextes überflüssig gewordene Deklarationen herausgefiltert werden.
Existieren z.B. vom Programmieren-im-Großen vorgegebene, noch nicht angewandte Ressourcen, deutet dies darauf hin, daß das Programm noch nicht vollständig ist und ergänzt werden muß. Existieren andererseits vom Benutzer beim Programmieren-im-Kleinen eingefügte Deklarationen, die nicht angewandt werden, könnten diese auch wieder gelöscht werden, ohne die Semantik des Programms zu ändern. Derartige Veränderungen des Quelltextes können vom Benutzer im Anschluß an ein derartiges Analysekommando mit Hilfe des Editors durchgeführt werden.

Es wäre sicherlich sehr aufwendig, wenn bei der Analyse größerer Programm-

einheiten der IPSEN-Benutzer gezwungen wäre, ein solches Analysekommando für jede einzelne Deklaration bzw. importierte Ressource zu aktivieren. Aus diesem Grunde sind alle Analysekommandos auch auf allen, ein analysierbares Inkrement umfassenden Inkrementen aktivierbar. Durch die Auswahl eines geeigneten Inkrements, z.B. eine Liste von Variablendeklarationen oder der gesamte Deklarationsteil, hat der Benutzer somit die Möglichkeit, den **Umfang der Analyse** festzulegen. Darüberhinaus bieten wir in IPSEN dem Benutzer eine weitere Möglichkeit, den **Verlauf der Analyse** im Dialog zu steuern. Hierzu wird der IPSEN-Benutzer bei der Analyse gesamter Deklarationsteile vor jedem Einstieg in die Analyse einer Prozedurdeklaration gefragt, ob die derzeitige Analyse auf die Prozedurdeklaration ausgedehnt werden soll oder diese Prozedurdeklaration übergangen werden soll. Eine derartige, dialoggesteuerte Analyse ist insbesondere deshalb sinnvoll, da, wie alle anderen IPSEN-Kommandos, auch die Analysekommandos bei noch unvollständigen Moduln aktiviert werden dürfen. In diesem Fall kann der IPSEN-Benutzer am besten entscheiden, welche Prozedurdeklarationen bereits in einem sinnvollerweise analysierbaren Zustand sind. Neben dieser Steuerung des Analyseverlaufs hat der IPSEN-Benutzer auch stets die Möglichkeit, den Analysevorgang abzubrechen. Hierzu läßt er die Frage des Systems, ob die Analyse auf eine weitere Prozedurdeklaration ausgedehnt werden soll, unbeantwortet und aktiviert ein beliebiges anderes auf dem aktuellen Inkrement gültiges Kommando.

Bei der Ausführung des Kommandos An - non-applied wird das aktuelle Inkrement in der **textuellen Reihenfolge** untersucht. Jede entdeckte nicht angewandte Deklaration wird im aktuellen View besonders markiert und eine entsprechende Meldung in einem Nachrichtenfenster ausgegeben. Hierzu wird u.U. der Quelltext im aktuellen View automatisch vorwärtsgeblättert.

Bei der bisher beschriebenen Vorgehensweise ist der IPSEN-Benutzer gezwungen, sich die gefundenen, nicht angewandten Deklarationen zu merken. Eine sinnvolle Erweiterung der Funktionalität dieses und auch aller folgenden Analysekommandos ist deshalb die Möglichkeit, während der Analyse ein **Druckerprotokoll** anschalten zu können. Dieses Druckerprotokoll erleichtert dem IPSEN-Benutzer dann anschließend eine geeignete Auswertung der ermittelten Analyseergebnisse. Eine andere denkbare Erweiterung ist, derartige Analysekommandos vollständig ohne Benutzerdialog ablaufen zu lassen und anschließend ein Druckerprotokoll mit allen relevanten Analyseergebnissen ausgeben zu lassen.

Das oben vorgestellte Analysekommando könnte im Falle von Datenobjektdeklarationen dahingehend verfeinert werden, daß bei angewandten Auftreten dieses Datenobjekts im Anweisungsteil noch einmal unterschieden wird zwischen einem setzenden und einem benutzenden Auftreten. Hierbei sprechen wir von einem **setzenden Auftreten** eines Datenobjekts, wenn es an dieser Stelle (möglicherweise)

einen neuen Wert erhält, und von einem **benutzenden Auftreten,** wenn an dieser Stelle nur sein Wert gelesen wird.
Bei Datenobjektdeklarationen sind dementsprechend zwei weitere, zum obigen Kommando analoge Kommandos denkbar, bei denen nur überprüft wird, ob ein deklariertes Datenobjekt an keiner Stelle gesetzt oder benutzt wird. Bei den durch die Ausführung dieses Kommandos entdeckten Datenobjekten, die zwar gesetzt, aber nie benutzt werden, könnten dann sowohl die Deklaration als auch alle setzenden Auftreten vom Benutzer gelöscht werden, ohne die Semantik des Programms zu ändern. Bei Datenobjekten, die nur benutzt aber nie gesetzt werden, sollte der IPSEN-Benutzer auf jeden Fall den Programmtext modifizieren. Denn sobald eine Anweisung während der Ausführung erreicht wird, in der dieses Datenobjekt benutzt wird, tritt u.U. ein Laufzeitfehler auf, da der augenblickliche, zufällige Speicherinhalt des Datenobjekts nicht im erwarteten Wertebereich liegt. Zum Erkennen solcher möglicher Laufzeitfehler bereits zur Programmerstellungszeit dienen diese Kommandos.

2.2.2 Analyse des Kontroll- und Datenflusses

Bei der Entwicklung eines Programms liegt es in der Verantwortung des Programmierers, also des IPSEN-Benutzers, dafür zu sorgen, daß der Kontroll- und Datenfluß im Anweisungsteil so beschaffen ist, daß bei einer Ausführung des Programms das erwartete Verhalten eintritt. In diesem Zusammenhang sind die folgenden Kommandos sinnvoll, die es dem IPSEN-Benutzer ermöglichen, bereits zur Programmerstellungszeit weitergehende Informationen bez. Kontroll- und Datenfluß zu erhalten.

Das erste Kommando in diesem Zusammenhang dient dazu, den im aktuellen Programmtext verkapselten Kontrollfluß näher zu untersuchen. Da in Modula-2 keine beliebigen Sprunganweisungen erlaubt sind, sind alle Modula-2-Programme wohlstrukturiert. Das heißt insbesondere, daß neben einfachen und bedingten Anweisungen nur sogenannte "single-entry single-exit loops" existieren (vgl. /Di 72/, /Wi 74/). Im Gegensatz zu vielen anderen Programmiersprachen ist daher das Entdecken pathologischen Kontrollflusses relativ problemlos. Hierbei verstehen wir im folgenden unter **nicht erreichbaren** Anweisungen solche Anweisungen, die aufgrund der rein syntaktischen Struktur des Programmtextes während der Ausführung nicht erreichbar sind. Dies sind Anweisungen, die in einem Prozedurrumpf unmittelbar hinter einer return-Anweisung oder im Rumpf einer loop-Schleife unmittelbar hinter einer exit-Anweisung stehen. Weiterhin gehören hierzu alle Anweisungen, die unmittelbar hinter einer loop-Schleife ohne exit-Anweisung stehen. Damit sind dementsprechend keine Anweisungen gemeint, die statisch (zur Compilezeit) bzw.

dynamisch (zur Ausführungszeit) als nicht erreichbare Anweisungen erkannt werden können.

Das Kommando zur Entdeckung solcher nicht erreichbarer Anweisungen heißt **Au - unreachable (statements).** Es kann vom IPSEN-Benutzer aktiviert werden, wenn das aktuelle Inkrement der Rumpf eines Moduls oder einer Prozedurdeklaration oder eine beliebige Anweisung ist. Bei Aufruf dieses Kommandos werden die im aktuellen Inkrement enthaltenen Anweisungen in textueller Reihenfolge überprüft, ob sie nicht erreichbar sind. Etwaige nicht erreichbare Anweisungen werden im aktuellen View entsprechend markiert und dem Benutzer eine entsprechende Meldung ausgegeben.
Auch dieses Kommando ist auf größeren Inkrementen wie Prozedurdeklaration oder dem gesamten Modul aktivierbar. In diesem Fall kann der IPSEN-Benutzer wiederum im Dialog entscheiden, ob die Analyse auf prozedur- bzw. modullokale Prozedurdeklarationen auszudehnen ist.

Die nächsten Kommandos dienen dazu, den im aktuellen Programmtext verkapselten Datenfluß näher zu untersuchen.

Durch eine geeignete Plazierung von Datenobjektdeklarationen hat der Programmierer die Möglichkeit, den Gültigkeits- und Sichtbarkeitsbereich von Datenobjekten festzulegen. Um dem IPSEN-Benutzer in diesem Zusammenhang bei der Programmerstellung Unterstützung anzubieten und eventuelle Laufzeitfehler, wie z.B. unerwünschte Seiteneffekte bei Ausführung von Prozeduren, zu vermeiden, wird das Kommando **Ag - global (data objects)** zur Verfügung gestellt. Wird dieses Kommando aktiviert, wenn das aktuelle Inkrement eine Prozedurdeklaration ist, werden in einem weiteren Fenster alle bei Aufruf dieser Prozedur u.U. angewandten globalen Datenobjekte angezeigt. (Datenobjekte sind **global** zu einer Prozedurdeklaration, wenn ihre Deklaration in einer statisch umgebenden Programmeinheit liegt). Hierbei werden auch solche Datenobjekte ermittelt, die in einer im Rumpf der untersuchten Prozedurdeklaration aufgerufenen Prozedur angewandt werden und global zur untersuchten Prozedurdeklaration sind.
Werden durch diese Analyse bei der Ausführung mögliche, u.U. unbeabsichtigte Seiteneffekte ermittelt, hat der Benutzer die Möglichkeit, mit Hilfe des syntaxgestützten Editors z.B. zusätzliche Datenobjektdeklarationen einzutragen oder die Parameterlisten von Prozeduren zu erweitern, um solch einen globalen Zugriff explizit zu machen.

Die letzten beiden Kommandos der in diesem Abschnitt exemplarisch vorgestellten Kommandos des Werkzeugs der statischen Analyse ermöglichen es, über die Verwendung eines bestimmten Datenobjekts weitere Informationen zu erhalten. Während mit dem Kommando An - non-applied (declarations/imports) nur ermittelt werden konnte, welche Datenobjekte überhaupt nicht angewandt werden, dient das Kommando **Ao - (setting/using) occurrences** dazu, zu einem Datenobjekt alle

benutzenden und setzenden Auftreten dieses Datenobjekts zu ermitteln und anzeigen zu lassen. Es kann vom IPSEN-Benutzer aktiviert werden, wenn das aktuelle Inkrement eine Datenobjektdeklaration oder die Deklaration eines formalen Parameters ist. Nach Aktivierung des Kommandos werden in einem zweiten Textfenster nacheinander alle setzenden und benutzenden Auftreten dieses Datenobjekts in textueller Reihenfolge angezeigt. Hierzu wird zunächst der Anweisungsteil der Programmeinheit durchsucht, in der das aktuelle Inkrement enthalten ist. Neben derartigen lokalen angewandten Auftreten des aktuell untersuchten Datenobjekts können auch globale Auftreten dieses Datenobjekts in aufgerufenen Prozeduren existieren. Analog zu der oben erläuterten Vorgehensweise eines durch den IPSEN-Benutzer im Dialog gesteuerten Analyseverlaufs wird auch bei der Ausführung dieses Kommandos der IPSEN-Benutzer gefragt, ob die Analyse auf lokal deklarierte Prozeduren ausgedehnt werden soll, in dessen Rumpf das untersuchte Datenobjekt global angewandt wird.

Damit haben wir bei der Aktivierung dieses Kommandos auf einer größeren Programmeinheit, z.B. einem ganzen Modul, eine **zweistufige Dialogsteuerung** durch den IPSEN-Benutzer:

- Erstens kann er im Dialog beeinflussen, in welchen Prozedurdeklarationen die dort enthaltenen Datenobjekt- bzw. Parameterdeklarationen analysiert werden sollen.
- Zweitens kann er bei der Durchführung der Analyse für eine einzelne Datenobjekt- bzw. Parameterdeklaration steuern, welche Anweisungsteile nach setzenden bzw. benutzenden Auftreten dieses Objekts untersucht werden sollen.

Außerdem hat der Benutzer an jeder Stelle die Möglichkeit, die Analyse vollständig abzubrechen. Wie bereits oben erwähnt, bietet diese dialoggesteuerte Vorgehensweise dem IPSEN-Benutzer die Möglichkeit, gezielt die Teile des Programmtextes analysieren zu lassen, die sich bereits in einem analysierbaren Zustand befinden. Da es außerdem aufgrund der begrenzten Bildschirmgröße nicht möglich ist, stets den gesamten Programmtext am Bildschirm darzustellen, behält der Benutzer durch den expliziten Übergang zu anderen Prozeduren leichter den Überblick, welche Teile des Programmtextes derzeit analysiert werden.

Das Kommando **As - set/use chains** dient der Ermittlung der in der Datenflußanalyse bekannten "set/use-Ketten" (vgl. /Au 79/). Hiermit sind ausgehend von der Position, an der einem Datenobjekt ein Wert zugewiesen wird, alle die Stellen im Programmtext gemeint, an denen auf diesen Wert der Variablen lesend zugegriffen wird. Das Kommando kann vom IPSEN-Benutzer aktiviert werden, wenn das aktuelle Inkrement ein Datenobjekt auf der linken Seite einer Zuweisungsanweisung oder ein aktueller call-by-reference-Parameter in einem Prozeduraufruf ist. In einem zweiten Textfenster werden nacheinander alle die Stellen angezeigt, an denen auf diesen Wert des Datenobjekts lesend zugegriffen wird. Wie beim obigen Kommando Ao wird auch hier zunächst nur prozedur- bzw. modullokal gesucht. Nur

auf expliziten Wunsch des Benutzers wird bei globaler Benutzung die Analyse auf weitere Prozedurrümpfe ausgedehnt.

Bei der Ausführung dieses Kommandos wird der Programmtext in Richtung des Kontrollflusses untersucht. Im Gegensatz zu einer solchen "Vorwärtsanalyse" ist auch eine "Rückwärtsanalyse" (vgl. /Za 83/) denkbar, bei der zu einem bestimmten benutzenden Auftreten eines Datenobjekts untersucht wird, an welchen Stellen dieses Datenobjekt seinen Wert bekommen haben kann. Diese Analyse ist in der Datenflußanalyse auch unter dem Namen "use/set-Analyse" bekannt (/AU 79/).

Derartige Kommandos zur Untersuchung des Datenflusses sind vor allem bei der Suche nach der Ursache von Laufzeitfehlern hilfreich. Ganz analog verlaufen auch Analysen, ob kontextsensitive Regeln der folgenden Art erfüllt sind:

- existiert bez. des Kontrollflusses zwischen je zwei setzenden Auftreten mindestens ein benutzendes Auftreten;
- wird jedes Datenobjekt initialisiert, d.h. existiert bez. des Kontrollflusses zwischen dem Anfang der Prozedur, in der dieses Datenobjekt deklariert wurde, und jedem benutzenden Auftreten mindestens ein setzendes Auftreten, oder eingeschränkter, ist kein benutzendes Auftreten vom Anfang der Prozedur aus direkt erreichbar.

2.2.3 Kommandoübersicht

Die vorgestellten Kommandos des Werkzeugs zur statischen Analyse werden in der folgenden **Übersicht** zusammengefaßt.

A - static Analysis

An - non-applied
Au - unreachable
Ag - global
Ao - occurrences
As - set/use

Abb. 2.2.3.1: Kommandos des Werkzeugs der statischen Analyse

Dieser Kommandosatz ist nur als erster Ansatz für ein derartiges Analysewerkzeug in IPSEN für den Bereich des Programmierens-im-Kleinen zu verstehen. Alle diese Kommandos werden in der Prototypimplementierung von IPSEN realisiert. Weitere Kommandos sind in späteren Versionen von IPSEN denkbar. Einige von ihnen wurden in den obigen Erläuterungen bereits erwähnt.

2.3 Ausführung

In den letzten beiden Paragraphen haben wir einige Kommandos vorgestellt, die vom IPSEN-System dem Benutzer zur Verfügung gestellt werden, um ein Modula-2-Programm zu erstellen und zu analysieren. Daneben werden dem IPSEN-Benutzer auch umfangreiche Hilfen für **Ausführungs- und Testaktivitäten** zur Verfügung gestellt. Es ist Ziel dieses und des nächsten Paragraphen, diese Unterstützungsmöglichkeiten im einzelnen vorzustellen. Im Paragraphen 2.5 werden wir dann an einem Beispiel verdeutlichen, wie in der integrierten Entwicklungsumgebung IPSEN diese Erstellungs-, Analyse-, Ausführungs- und Testkommandos vom Benutzer eingesetzt werden können, um einen fehlerfreien Modula-2-Modul zu erstellen.

2.3.1 Ausführbare Inkremente

Ein wesentliches Charakteristikum des IPSEN-Systems besteht darin, daß der Benutzer grundsätzlich ohne großen Aufwand nach Beendigung einer Werkzeugaktivität eine Aktivität eines beliebigen anderen Werkzeugs initiieren kann. Voraussetzung hierfür ist in der Regel nur, daß das gewünschte Kommando auf dem aktuellen Inkrement auch erlaubt ist. Für die Ausführung bedeutet dies, daß zu **jedem beliebigen Zeitpunkt**, u.a. auch bei größtenteils noch unvollständig erstellten Moduln, die Ausführung gestartet werden kann, um die bereits erstellten Modulteile zu testen. Wir werden im nächsten Abschnitt erläutern, wie das IPSEN-System sich verhält, wenn während der Ausführung noch fehlende Quelltextstücke entdeckt werden. Die Ausführung auch erst teilweise erstellter Programmeinheiten kann durchaus sinnvoll sein, um bereits in einem sehr frühen Zustand einzelne Ressourcen eines Moduls auszutesten. Hierdurch sollte der IPSEN-Benutzer aber nicht dazu verleitet werden, durch "Ausprobieren" zu korrekten Programmen zu kommen. Der IPSEN-Benutzer sollte sich vielmehr im voraus überlegen, für welche Testdaten er das bis dahin erstellte Programm testen möchte und stets reproduzierbare Tests durchführen. Letzteres wird vom IPSEN-System insbesondere dadurch unterstützt, daß die Ausführung nur auf zwei Inkrementebenen gestartet werden darf. Derartige **ausführbare Inkremente** sind erstens gesamte Programm-Moduln und zweitens einzelne Ressourcen (Prozeduren) eines beliebigen Modula-2-Moduls.

Während Programm-Moduln unmittelbar ausgeführt werden können, müssen vor der Ausführung von Ressourcen eines Moduls zunächst alle modullokalen Datenobjekte und formalen Parameter der Ressource **initialisiert** werden. Da es bei großen Datenstrukturen sehr aufwendig wäre, wenn der IPSEN-Benutzer alle diese Datenobjekte einzeln initialisieren müßte, werden diese vom IPSEN-System zunächst mit einem "Default"-Wert vorbesetzt. Diese Variablen werden zusammen mit diesen

"Default"-Werten in einem weiteren Fenster auf dem Bildschirm angezeigt. Der Benutzer hat dann die Möglichkeit, gezielt bestimmte Variable mit einem Wert zu belegen. Abb. 2.3.1.1 zeigt einen Bildschirmausschnitt, in dem auf dem aktuellen View ein derartiges weiteres Fenster eröffnet wurde. Der Benutzer hat die Möglichkeit, in diesem weiteren Fenster zu blättern, einzelne Variablenbezeichner zu selektieren und in einem dann vom System eröffneten Eingabefenster einen neuen Wert einzutragen.

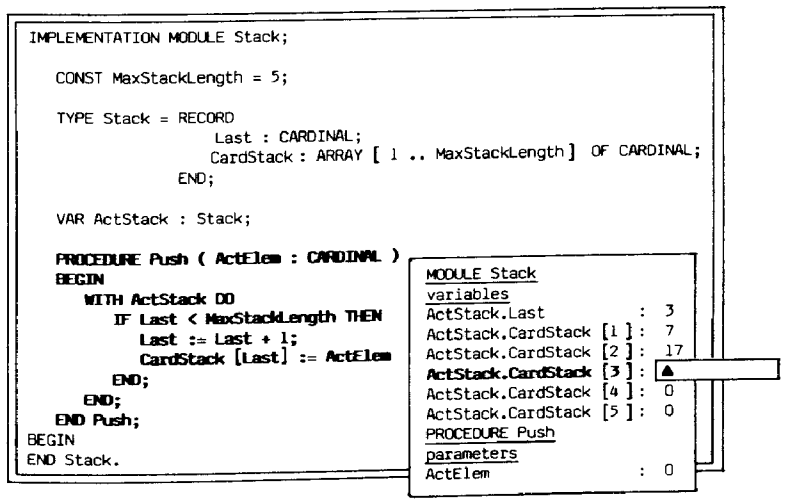


Abb. 2.3.1.1: Initialisierung von Variablen

Da in der Regel derartige Tests vom IPSEN-Benutzer wiederholt mit evtl. nur leicht veränderten Testdaten ausgeführt werden, werden diese vom Benutzer im Dialog gesetzten Initialisierungen für spätere Testläufe aufgehoben. Bei einem erneuten Aufruf der Ausführung werden die einzelnen Datenobjekte dann mit diesen Werten initialisiert.

2.3.2 Unterbrechungspunkte

In diesem Abschnitt werden verschiedene Möglichkeiten und Situationen vorgestellt, die eine Unterbrechung der aktuellen Ausführung eines Programm-Moduls oder einer einzelnen Ressource bewirken.

Bei einer Unterbrechung der Ausführung hat der Benutzer aufgrund der integrierten Arbeitsweise des Systems grundsätzlich die Möglichkeit, jedes beliebige andere auf dem aktuellen Inkrement gültige Kommando zu aktivieren. Je nach Art dieser

während einer Unterbrechung aktivierten Kommandos kann die Ausführung dann anschließend fortgesetzt werden oder muß neu gestartet werden. So kann die Ausführung z.B. nicht fortgesetzt werden, wenn während einer solchen Unterbrechung mit Hilfe des syntaxgestützten Editors der Quelltext geändert wird. Denn da bei einer solchen Änderung u.U. bereits ausgeführte Anweisungen modifiziert werden oder z.B. durch das Ändern von Deklarationen der Gültigkeitsbereich von bereits benutzten Datenobjekten modifziert wird, wären die erzielten Testergebnisse für den IPSEN-Benutzer nicht mehr nachvollziehbar. Werden also derartige Editorkommandos vom Benutzer aktiviert, wird die aktuelle Ausführung beendet. Nach anderen Aktivitäten, die vor allem im nächsten Paragraphen vorgestellt werden und dem IPSEN-Benutzer dazu dienen, sich über den Stand der Ausführung zu informieren bzw. den weiteren Verlauf zu beeinflussen, kann die Ausführung fortgesetzt werden.

Nun zu den einzelnen Möglichkeiten und Situationen, die eine Unterbrechung der Ausführung bewirken:
Wird während der Ausführung ein **Laufzeitfehler** (z.B. Überschreiten einer Feldgrenze) entdeckt, wird die Ausführung sofort unterbrochen, die entsprechende Stelle im im aktuellen View dargestellten Quelltext markiert und in einem Nachrichtenfenster eine entsprechende Meldung ausgegeben. Nach Bestätigung dieser Meldung durch den Benutzer wird die Ausführung in diesem Fall vollständig abgebrochen. Der Benutzer hat dann z.B. die Möglichkeit, den Quelltext mit Hilfe des syntaxgestützten Editors zu ändern und anschließend die Ausführung neu zu starten.
Weiterhin muß die Ausführung vom System unterbrochen werden, wenn während der Ausführung eine **Lücke im Quelltext** erreicht wird. Solche Lücken können sich insbesondere im Anweisungsteil befinden, z.B. eine noch fehlende Schleifenbedingung in einer while-Schleife oder die fehlende linke Seite in einer Zuweisungsanweisung. Bei einer derartigen Unterbrechung wird dem IPSEN-Benutzer erlaubt, mit Hilfe des syntaxgestützten Editors diese Lücke zu füllen. Falls diese Änderung des Quelltexts lokal auf das Auffüllen dieser Lücke beschränkt bleibt, ist sichergestellt, daß die dort eingefügten Inkremente keine Auswirkung auf den bereits ausgeführten Programmtext haben. In diesem Fall kann nach dem Füllen der Lücke die Ausführung mit der Ausführung der neu eingetragenen Programmstücke fortgesetzt werden. Werden jedoch z.B. beim Ausfüllen der Lücke neu zu deklarierende Datenobjekte benötigt, ist der Einfluß solcher Deklarationen auf den bereits ausgeführten Programmtext schwer zu ermitteln. In diesem Fall muß die Ausführung vom IPSEN-Benutzer neu gestartet werden.

Schließlich wird die Ausführung in der jetzigen Prototyp-Implementierung des IPSEN-Systems unterbrochen, wenn eine **importierte Prozedur** aufgerufen wird. Denn da im Moment nur der Aufgabenbereich des Programmierens-im-Kleinen abgedeckt

wird und die Integration mit dem Programmieren-im-Großen noch nicht vollzogen ist, muß der Aufruf einer solchen Prozedur vom IPSEN-Benutzer von Hand simuliert werden. Dies bedeutet im wesentlichen, daß die Ergebnisparameter eines solchen Prozeduraufrufs mit aktuellen Werten besetzt werden müssen. Wir kommen hierauf im Abschnitt 2.4.3 zurück.

Neben diesen im Programmtext "implizit" enthaltenen Unterbrechungspunkten hat der IPSEN-Benutzer die Möglichkeit, weitere Unterbrechungspunkte für die Ausführung explizit zu setzen. Derartige Unterbrechungspunkte sind insbesondere beim Testen einer Ressource oder eines Programm-Moduls hilfreich, da sich der IPSEN-Benutzer bei einer Unterbrechung z.B. über den aktuellen Stand der Ausführung informieren kann und anschließend die Ausführung fortsetzen lassen kann. Bei diesen Unterbrechungspunkten beschränken wir uns im IPSEN-System auf sogenannte **Unterbrechungsanweisungen,** die wie übliche Modula-2-Anweisungen im Anweisungsteil stehen. In diesem Sinne werden keine Unterbrechungspunkte vorgestellt, die von globalen Bedingungen wie z.B. Zeitschranken für die Ausführungsdauer abhängig sind. Ebenso werden hier keine Unterbrechungspunkte untersucht, die von dem Datenfluß bestimmter Variablen abhängig sind. So könnte z.B. immer dann die Ausführung unterbrochen werden, wenn sich der Wert einer bestimmten Variablen ändert.
Die hier vorgestellten Unterbrechungspunkte können an jedem Punkt im Quelltext stehen, an dem auch eine beliebige Modula-2-Anweisung stehen kann. Es ist Aufgabe des IPSEN-Benutzers, diese Unterbrechungsanweisungen vor dem Start der Ausführung in den Quelltext einzutragen. Hierzu stehen ihm die Kommandos des Werkzeugs **Testvorbereitung** zur Verfügung, von dem wir im folgenden einige Kommandos vorstellen werden. Damit der IPSEN-Benutzer sieht, an welchen Stellen er den Quelltext um derartige Unterbrechungsanweisungen angereichert hat, werden diese im Quelltext entsprechend dargestellt. Hierbei unterscheiden wir dann noch einmal zwischen **unbedingten** und **bedingten** Unterbrechungsanweisungen. Während beim Erreichen einer unbedingten Unterbrechungsanweisung die Ausführung auf jeden Fall unterbrochen wird, wird beim Erreichen einer bedingten Unterbrechungsanweisung die Ausführung nur dann unterbrochen, wenn eine dort stehende Zusicherung nicht erfüllt ist. Diese Zusicherung ist ein üblicher boolescher Ausdruck im Modula-2-Sinn, der u.a. aus an dieser Stelle gültigen Variablen zusammengesetzt ist. Die konkrete Darstellung einer **unbedingten Unterbrechungsanweisung** im Quelltext ist:

```
(# break #)
```

Eine **bedingte Unterbrechungsanweisung** ist abhängig von dem Nichterfülltsein eines booleschen Ausdrucks und wird deshalb wie folgt dargestellt:

```
(# assert: (< Expression >) #)
   (# break #)
(# end assert #)
```

Abb. 2.3.2.1 zeigt die entsprechend um eine bedingte Unterbrechungsanweisung ergänzte Ressource Push aus Abb. 2.3.1.1.

```
PROCEDURE Push ( ActElem : CARDINAL );
BEGIN
   WITH ActStack DO
      (# assert: Last >= 0 #)
         (# break #)
      (# end assert #)
      IF Last < MaxStackLength THEN
         Last := Last + 1;
         CardStack[ Last ]:= ActElem
      END;
   END;
END Push;
```

Abb. 2.3.2.1: Darstellung einer bedingten Unterbrechungsanweisung

Bei Aufruf der Ressource Push wird die Ausführung nun immer dann unterbrochen, wenn der Wert von ActStack.Last echt kleiner als 0 ist und somit beim Zugriff auf das Feld ActStack.CardStack ein Laufzeitfehler auftreten würde.

Eine Unterbrechungsanweisung kann an all den Stellen eingefügt werden, an denen auch eine beliebige Anweisung stehen kann. Ist somit während der Bearbeitung das aktuelle Inkrement eine beliebige Anweisung, kann der IPSEN-Benutzer davor **(insert)** oder dahinter **(extend)** eine bedingte **(assertion)** oder unbedingte **(breakpoint)** Unterbrechungsanweisung einfügen. Wie jede beliebige andere Anweisung kann der IPSEN-Benutzer auch eine Unterbrechungsanweisung als aktuelles Inkrement selektieren (z.B. durch einen entsprechenden Mausklick). Anschließend kann er dann diese Unterbrechungsanweisung löschen **(delete)** oder den Ausdruck in einer bedingten Unterbrechungsanweisung ändern **(change)**. Da bei einer umfangreichen Anreicherung des Quelltextes um derartige Unterbrechungsanweisungen der eigentliche Quelltext u.U. schwer lesbar ist, hat der IPSEN-Benutzer die Möglichkeit, sich den Quelltext mit **(on)** oder ohne **(off)** derartige Unterbrechungsanweisungen anzusehen. Alle diese Aktivitäten gehören zum Werkzeug Testvorbereitung und können durch die folgenden Kommandos aktiviert werden:

T - Testing preparation

Tib - insert breakpoint	Td - delete
Teb - extend breakpoint	Tc - change
Tia - insert assertion	To - on/off
Tea - extend assertion	

Abb. 2.3.2.2: Kommandos des Werkzeugs Testvorbereitung

Wir werden im folgenden Paragraphen 2.4 weitere Kommandos dieses Werkzeugs kennenlernen, durch die dem Benutzer zusätzliche Unterstützung während des Tests angeboten wird.

2.3.3 Ausführungsarten

Die im letzten Abschnitt vorgestellten Unterbrechungsanweisungen bewirken während der Ausführung u.U. eine Unterbrechung. Da derartige Unterbrechungsanweisungen vom IPSEN-Benutzer stets explizit eingefügt werden müssen, nennen wir im folgenden eine derartige Unterbrechung eine **kontrollierte Unterbrechung** der Ausführung. Daneben werden wir in diesem Abschnitt zwei weitere Möglichkeiten einer Unterbrechung der Ausführung kennenlernen, nämlich die **automatische** und die **manuelle** Unterbrechung. Alle diese verschiedenen Unterbrechungen der Ausführung können vom IPSEN-Benutzer vorbereitet bzw. angestoßen werden. Durch eine geeignete Kombination dieser Möglichkeiten kann er auf diese Weise den Verlauf der Ausführung und des Tests einer Ressource oder eines Programm-Moduls nach seinen Wünschen gestalten.

Bei der Vorstellung der Analysekommandos im Paragraphen 2.3 haben wir bereits diskutiert, welche Möglichkeiten der IPSEN-Benutzer hat, den Verlauf der Analyse zu steuern. So wurde z.B. die Analyse nur dann auf weitere Prozedurdeklarationen ausgedehnt, wenn der Benutzer dies explizit gewünscht hat. Dieselbe Idee findet nun auch bei der **Ausführung mit automatischer Unterbrechung** Anwendung. Dies bedeutet konkret, daß der Benutzer vor jeder Ausführung einer Anweisung entscheiden kann, wann die Ausführung das nächste Mal automatisch unterbrochen werden soll. Hierbei kann er sich in der Regel für eine der drei folgenden Möglichkeiten entscheiden:

a) Unterbrechung vor Ausführung der ersten Anweisung im Rumpf der aktuellen Anweisung
b) Unterbrechung nach vollständiger Ausführung der aktuellen Anweisung
c) Unterbrechung nach vollständiger Ausführung der Anweisung bzw. der Programmeinheit, in deren Rumpf die aktuelle Anweisung enthalten ist.

Wir verdeutlichen diese verschiedenen Möglichkeiten am folgenden Beispiel in Abb. 2.3.3.1, in dem schematisch ein Ausschnitt aus einem Modula-2-Modul angegeben ist. Außerdem sind die Anweisungen angegeben, an denen die Ausführung das nächste Mal automatisch unterbrochen wird, wenn die aktuelle Anweisung die while-Schleife ist und sich der Benutzer für eine der obigen drei Alternativen entscheidet.

```
IF X < 0 THEN
   WHILE X <= 0 DO
   (a) X := X + 1;
       ...
       ...
   END;

(b) Y := X;
    ...
 ELSE
    ...
 END;
(c) CASE ...
       ...
       ...
 END;
```

Abb. 2.3.3.1: Unterbrechungspunkte bei automatischer Unterbrechung

Man erkennt unmittelbar, in welcher Situation der Benutzer sich für welche der angegebenen Alternativen entscheiden sollte. Vermutet er z.B. in der Umgebung der aktuellen Anweisung einen Fehler, wird er sich für Atlernative a) entscheiden und auf diese Art und Weise die Ausführung **schrittweise** verfolgen. Das zugehörige Kommando lautet deshalb auch **Xs - step.**

Ist der Benutzer der Meinung, daß bei der Ausführung der aktuellen Anweisung nichts Unerwartetes passiert, kann er diese Anweisung vollständig **in einem Schritt** ausführen lassen. Das zugehörige Kommando lautet dann **Xg - go.**

Ist ihm schließlich die derzeitige Ausführung mit automatischer Unterbrechung bei jeder Anweisung zu langwierig, kann er auch die aktuelle, übergeordnete Anweisung ohne Unterbrechung ausführen lassen und so **die Ausführung beschleunigen.** Das zugehörige Kommando lautet dann **Xr - run.**

Alle diese Kommandos gehören zum Werkzeug Ausführung und Test (eXecution). Die Kommandos werden in der folgenden Übersicht zusammengefaßt.

X - eXecution

Xs	-	step
Xg	-	go
Xr	-	run

Abb. 2.3.3.2: Übersicht über Kommandos des Werkzeugs Ausführung und Test

Die Kommandos Xg - go bzw. Xr - run dienen natürlich auch dazu, eine einzelne Ressource oder einen Programm-Modul in herkömmlicher Art und Weise ohne Unterbrechung ausführen zu lassen. Denn in diesem Fall wird gemäß obiger Erläuterung die Ausführung erst unterbrochen, wenn das aktuelle Inkrement vollständig ausgeführt wurde. In diesem Fall ist dies identisch mit einer vollständigen

Ausführung des gesamten Programm-Moduls bzw. der gesamten Ressource.

Zusätzlich zu dieser Ausführung mit automatischer Unterbrechung ist es dem IPSEN-Benutzer jederzeit erlaubt, durch Drücken einer speziellen Unterbrechungstaste die **Ausführung per Hand** zu unterbrechen. Eine derartige Unterbrechung nennen wir deshalb auch **manuelle Unterbrechung.** In diesem Fall wird dann die nächste auszuführende Anweisung als aktuelles Inkrement im aktuellen View dargestellt, und der IPSEN-Benutzer kann die Bearbeitung mit einem beliebigen gültigen Kommando fortsetzen.

Häufig ist es wünschenswert, daß die im letzten Abschnitt vorgestellten Unterbrechungsanweisungen während der Ausführung nur dann beachtet werden, wenn das entsprechende Quelltextstück noch getestet werden muß. Ist der IPSEN-Benutzer dann der Meinung, daß in dem betreffenden Quelltextstück keine Fehler mehr enthalten sind, ist es hilfreich, wenn diese zusätzlichen Unterbrechungsanweisungen zeitweilig ausgeblendet werden können und während der Ausführung nicht beachtet werden. Aus diesem Grunde wird die Funktionalität des im letzten Abschnitt vorgestellten Kommandos **To - on/off** dahingehend erweitert, daß nicht nur die textuelle Darstellung von Unterbrechungsanweisungen ein- bzw. ausgeblendet werden kann, sondern daß dies gleichzeitig auch bedeutet, daß diese Unterbrechungsanweisungen während der Ausführung beachtet bzw. nicht beachtet werden. In diesem Sinne dient das Kommando To - on/off zum An- bzw. Ausschalten einer **Testumgebung.**

2.3.4 Kommandoübersicht

Die in den letzten beiden Abschnitten vorgestellten Kommandos der Werkzeuge Testvorbereitung bzw. Ausführung und Test fassen wir in der folgenden Übersicht zusammen.

T - Testing preparation	X - eXecution
Tib - insert breakpoint	Xs - step
Teb - extend breakpoint	Xg - go
Tia - insert assertion	Xr - run
Tea - extend assertion	
Td - delete	
Tc - change	
To - on/off	

Abb. 2.3.4.1: Kommandos der Werkzeuge Testvorbereitung bzw. Ausführung und Test

2.4 Testunterstützung

Wir haben im vorigen Paragraphen verschiedene Möglichkeiten vorgestellt, wie der IPSEN-Benutzer den Verlauf der Ausführung steuern und beeinflussen kann. Insbesondere wurde dabei vorgestellt, welche Möglichkeiten bestehen, die Ausführung an bestimmten Punkten zu unterbrechen bzw. unterbrechen zu lassen. Ziel dieses Paragraphen ist nun zu erläutern, welche Hilfsmittel bei einer solchen Unterbrechung dem IPSEN-Benutzer zur Verfügung stehen, um sich über den derzeitigen Stand der Ausführung zu informieren bzw. den weiteren Verlauf der Ausführung zu beeinflussen. Um nach einer derartigen Unterbrechung die Ausführung fortsetzen zu können, muß gewährleistet sein, daß durch die durchgeführten Aktivitäten Syntax und Semantik des bereits ausgeführten Programmteils unverändert bleiben. Weiterhin muß die Ausführung stets an der unterbrochenen Stelle fortgesetzt werden, um reproduzierbare Testergebnisse zu erzielen.

Zunächst einmal sind bei einer Unterbrechung alle Kommandos der statischen Analyse aktivierbar, um z.B. weitere Informationen über Kontroll- und Datenfluß des momentan getesteten Programmabschnitts zu erhalten.
Darüberhinaus sind die im folgenden vorgestellten Kommandos des Werkzeugs Ausführung und Test aktivierbar.

2.4.1 Ausgabe von Variablenwerten

Das in diesem Abschnitt beschriebene Kommando bietet dem IPSEN-Benutzer die Möglichkeit, sich gezielt die **Werte bestimmter Variablen** anzeigen zu lassen. Hierzu muß der IPSEN-Benutzer den Bezeichner einer Variablen durch Eingabe in einem Eingabefenster dem System mitteilen. Da derartige Bezeichner bei komplexeren Datenstrukturen einen komplizierteren Aufbau haben können, wird der Benutzer hierbei vom IPSEN-System geeignet unterstützt. Wir erläutern dies an dem folgenden Beispiel: Die Ausführung wurde an einem Punkt unterbrochen, an dem die folgenden Deklarationen gültig und sichtbar sind:

```
CONST  MaxStackLength = 5;

TYPE   Stack = RECORD
                  Last      : CARDINAL;
                  CardStack : ARRAY [1 .. MaxStackLength] OF CARDINAL
               END;

VAR    ActStack : Stack;
```

Abb. 2.4.1.1: Ausschnitt aus einem Deklarationsteil

Möchte sich der Benutzer nun bei einer Unterbrechung einzelne Elemente des

Datenobjekts ActStack.CardStack ansehen, aktiviert er das Kommando **Xv - variable inspection.** Auf dem Bildschirm erscheint ein Fenster mit einem Eingabefeld, in das der Benutzer nun einen Variablenbezeichner eintragen kann. Falls ihm in dieser Situation die genaue Gestalt der zugehörigen Konstanten-, Typ- bzw. Datenobjekt-Deklaration nicht bekannt ist, kann der Benutzer auch nur den Anfang eines gültigen Variablenbezeichners eintippen. In diesem Fall werden dem Benutzer in einem zusätzlichen Textfenster die relevanten Deklarationen angezeigt:

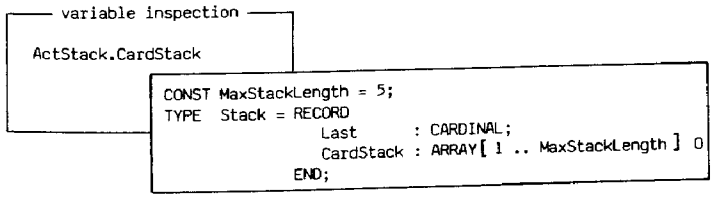


Abb. 2.4.1.2: Situation nach Eingabe eines unvollständigen Variablenbezeichners

Anschließend hat der Benutzer die Möglichkeit, im Eingabefeld den von ihm eingetippten Variablennamen zu verlängern, einen neuen Variablennamen einzutippen oder das Kommando abzubrechen. Verlängert der Benutzer den Variablennamen zu einem Bezeichner eines Objekts eines elementaren Datentyps, wird der Wert dieses Objekts ausgegeben:

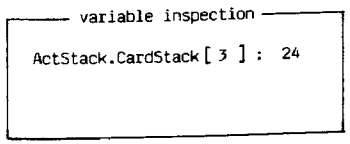


Abb. 2.4.1.3: Ausgabe eines Variablenwertes

Aktiviert der IPSEN-Benutzer dieses Kommando wiederholt, werden weitere Variablenbezeichner mit zugehörigen Werten in den folgenden Zeilen dieses Fensters angezeigt. Ist das Fenster voll, wird der Inhalt vom IPSEN-System automatisch vorwärtsgeblättert. Außerdem hat der IPSEN-Benutzer die Möglichkeit, beliebig in diesem Fenster zu blättern, um sich z.B. alte Werte einer Variablen anzusehen.

Auf die oben beschriebene Art und Weise ist es prinzipiell auch möglich, sich die Werte von Objekten dynamischer Datenstrukturen (z.B. Listen, Bäume usw.) vollständig anzusehen. Das hierzu nötige Eintippen u.U. sehr langer Variablenbezeichner ist jedoch sicherlich sehr umständlich und muß deshalb in einer späteren Version des IPSEN-Systems verbessert werden.

2.4.2 Speicherauszug

Neben dem gezielten Ansehen einzelner Variablenwerte kann sich der Benutzer auch größere Teile des **Inhalts des Laufzeitdatenbereichs** ansehen, indem er das Kommando **Xd - dump** aktiviert. Hierzu wird auf dem Bildschirm ein weiteres Fenster eröffnet, in dem Variablenbezeichner und ihre zugehörigen Werte ausgegeben werden.

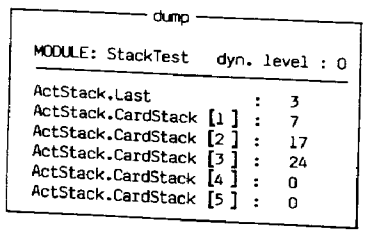


Abb. 2.4.2.1: Ausgabe eines Speicherauszugs

Zur Strukturierung dieser in der Regel sehr umfangreichen Ausgabe wird jeweils der Datenbereich zu einer Prozedurinkarnation zu einem Abschnitt zusammengefaßt. Zur Orientierung wird zu Beginn eines solchen Abschnitts der Name der aktuell dargestellten Prozedur bzw. des Moduls sowie die dynamische Verschachtelungstiefe angezeigt. Da in der Regel in einem Fenster nicht der gesamte Laufzeitdatenbereich darstellbar ist, hat der IPSEN-Benutzer die Möglichkeit, in diesem Fenster zu blättern. In der Abbildung 2.4.2.1 wird ein derartiger Ausschnitt aus dem Laufzeitdatenbereich angezeigt (vgl. hierzu die Deklarationen aus Abb. 2.4.1.1).

Auch bei diesem Kommando wird bisher nur der Inhalt des Laufzeitkellers angezeigt. Für eine spätere Version des IPSEN-Systems muß überlegt werden, wie auch der Inhalt des Laufzeitheaps geeignet dargestellt werden kann.

2.4.3 Simulation von Aufrufen von (importierten) Prozeduren

Da es im IPSEN-System erlaubt ist, noch unvollständige Programm-Moduln bzw. Ressourcen ausführen zu lassen, kann es vorkommen, daß während der Ausführung Prozeduren aktiviert werden sollen, deren Implementierung (noch) nicht vorliegt. Diese Situation tritt ein, wenn die aufgerufene Prozedur von außerhalb importiert wird oder der Benutzer eine von ihm deklarierte Prozedur noch nicht (vollständig) implementiert hat. In beiden Fällen bietet das Kommando **Xp - proc. call simulation** die Möglichkeit, den **Aufruf einer solchen Prozedur** durch geeignetes Setzen von Variablen zu **simulieren.** Um die Ausführung vor dem Aufruf einer noch

unvollständigen Prozedur zu unterbrechen, kann der IPSEN-Benutzer z.B. vor der Aufrufstelle eine unbedingte Unterbrechungsanweisung einfügen. Nach der Unterbrechung kann er dann das hier beschriebene Kommando aktivieren. Beim Aufruf importierter Ressourcen wird derzeit, da die Integration mit dem Programmieren-im-Großen noch nicht erfolgt ist, die Ausführung vom IPSEN-System automatisch unterbrochen und implizit das hier beschriebene Kommando aktiviert. Nach Aufruf dieses Kommandos wird auf dem Bildschirm analog zu dem im Abschnitt 2.3.1 erläuterten Vorgehen ein weiteres Fenster eröffnet. In diesem Fenster werden die Namen der aktuellen call-by-reference-Parameter zusammen mit ihren Werten angezeigt. Bei der Simulation noch nicht vollständig implementierter Prozeduren werden zusätzlich die Namen und Werte aller zu dieser Prozedur globalen Variablen angezeigt. Der Benutzer hat daraufhin die Möglichkeit, beliebige dieser Variablen zu selektieren und mit einem neuen Wert zu versehen (vgl. Abschnitt 2.3.1). Bei der Angabe des neuen Werts darf der Benutzer hierbei die Werte anderer, dort sichtbarer Variablen benutzen, also beliebige Ausdrücke eingeben.
(Bem.: Mit Hilfe dieses Kommandos hat der Benutzer die Möglichkeit, beliebigen (globalen) Variablen neue Werte zu geben und somit die weitere Ausführung zu beeinflussen. Dies macht eine Ausführung für den Benutzer u.U. nur schwer nachvollziehbar, so daß der IPSEN-Benutzer mit diesem Kommando verantwortungsbewußt umgehen sollte!)

2.4.4 Laufzeit-/Speicherplatzstatistik

Die bisher vorgestellten Kommandos zur Testunterstützung bieten dem IPSEN-Benutzer verschiedene Möglichkeiten, die während der Ausführung berechneten Variablenwerte anzusehen und somit die Ursache für u.U. aufgetretene Berechnungsfehler zu entdecken. Die in diesem Abschnitt vorgestellten Kommandos erlauben dem IPSEN-Benutzer darüberhinaus, mehr Information über Zeit- und Speicherplatzausnutzung während der Ausführung zu erhalten. Analog zu den im Abschnitt 2.2 beschriebenen Kommandos zur statischen Analyse haben auch die Ergebnisse dieser Kommandos für den IPSEN-Benutzer nur informativen Charakter, indem sie ihm aufzeigen, welche Teile des Programms sehr laufzeit- bzw. speicherplatzaufwendig sind.

Die in diesem Abschnitt vorgestellten Kommandos zur Laufzeitstatistik bewirken in der Regel reine Zählvorgänge während der Ausführung. Einige Beispiele für derartige Laufzeitmessungen sind etwa:

- Anzahl insgesamt ausgeführter Anweisungen
- Anzahl der Ausführung von Anweisungen einer bestimmten Art (z.B. Prozeduraufrufe)
- Anzahl ausgeführter Ein-/Ausgabeanweisungen

- Anzahl von Aufrufen einer bestimmten Prozedur
- Anzahl von Durchläufen einer bestimmten Schleife (Schleifenzähler)
- Anzahl von Zugriffen auf eine bestimmte Variable usw.

Für alle diese verschiedenen Möglichkeiten müssen eigene Zähler während der Ausführung verwaltet werden. Da der Aufwand für eine automatische Verwaltung dieser Zähler bei jeder Ausführung zu hoch wäre, muß der IPSEN-Benutzer vor Beginn der Ausführung festlegen, welche Zähler während der Ausführung verwaltet werden sollen. Wir wollen dies hier exemplarisch nur für eine Möglichkeit, einen Schleifenzähler, diskutieren.

Analog zum Setzen von Unterbrechungsanweisungen gehört auch das Setzen von Laufzeitzählern zum Werkzeug Testvorbereitung. Das heißt, daß dieses Kommando vor der Ausführung vom IPSEN-Benutzer aufzurufen ist. Der hier erläuterte **Schleifenzähler** dient dazu, bei einer bestimmten Schleife die Anzahl der Schleifendurchläufe zu zählen. Während einer Unterbrechung der Ausführung kann der Benutzer dann analog zum Ansehen von Variablenwerten sich den Wert eines solchen Schleifenzählers ansehen. Da Schleifen auch geschachtelt sein können, ist es erforderlich, daß der IPSEN-Benutzer jedem Zähler einen Namen gibt, damit er eindeutig spezifizieren kann, den Wert welchen Schleifenzählers er bei einer Unterbrechung sehen möchte. Zum Eintragen eines solchen Schleifenzählers dient das Kommando **Til - insert loop counter**, das der Benutzer stets aktivieren kann, wenn das aktuelle Inkrement eine Schleifenanweisung ist (und noch nicht mit einem Schleifenzähler instrumentiert ist). Damit der IPSEN-Benutzer sieht, daß und welchen Schleifenzähler er an einer bestimmten Schleifenanweisung gesetzt hat, wird dieser entsprechend im Quelltext dargestellt.

```
(# loop counter: Schleifel #)
WHILE NOT IsEmtpy () DO

  ...
END
```

Abb. 2.4.4.1: Beispiel für einen Schleifenzähler

Ein eingetragener Schleifenzähler kann mit dem Kommando **Td - delete** gelöscht oder mit dem Kommando **Tc - change** geändert werden. Wie alle vom Benutzer zu Testzwecken zusätzlich eingetragenen Informationen werden auch Schleifenzähler nur dann im Quelltext dargestellt bzw. bei der Ausführung beachtet, wenn der Benutzer durch das Kommando **To - on/off** die Testumgebung derzeit angeschaltet hat.

Neben Laufzeitstatistikinformationen hat der IPSEN-Benutzer die Möglichkeit, Auskunft über die Größe des während der Ausführung benutzten Speicherbereichs für Datenobjekte zu bekommen. Hierbei wird noch einmal zwischen vier unterschiedlichen Größen unterschieden. Aktiviert der IPSEN-Benutzer das Kommando **Xk - stack size** bzw. **Xh - heap size**, wird in einem Nachrichtenfenster die aktuelle Größe des Laufzeitkellers bzw. -heaps ausgegeben. Bei Aufruf des Kommandos **Xc - proc. call size** wird die für die aktuelle Prozedurinkarnation auf dem Laufzeitkeller reservierte Anzahl von Speicherplätzen ausgegeben. Schließlich kann sich der Benutzer auch für ein einzelnes Datenobjekt die auf dem Laufzeitkeller reservierte Anzahl von Speicherplätzen ausgeben lassen. Dazu wird nach Aufruf des Kommandos **Xo - object size** zunächst ein weiteres Eingabefenster eröffnet, in dem der Benutzer in freier Eingabe den Bezeichner eines Datenobjekts eintippen kann. Dies ist z.B. bei offenen Feldern als formaler Prozedurparameter interessant, da hier die tatsächliche Größe erst zur Laufzeit bekannt ist.
Abb. 2.4.4.2 zeigt beispielhaft das Aussehen dieses Fensters, nachdem die obigen vier Kommandos aktiviert wurden.


```
┌──── size ────┐
│              │
│ stack size :  9
│ heap size  :  0
│ proc. Push :  9
│ ActStack   :  6
└──────────────┘
```

Abb. 2.4.4.2: Ausgabe von Speicherplatzstatistik

2.4.5 Aufbau einer Testumgebung

Wir haben im Paragraphen 2.3 erläutert, welche Möglichkeiten der IPSEN-Benutzer hat, die Ausführung eines Programm-Moduls oder einer einzelnen Ressource zu steuern. Hierzu zählt u.a. die Möglichkeit, vor dem Start der Ausführung Unterbrechungsanweisungen in den Quelltext einzufügen und dadurch kontrollierte Unterbrechungen während der Ausführung vorzusehen. In diesem Paragraphen haben wir dann beschrieben, welche Unterstützungsmöglichkeiten das IPSEN-System während der Testphase dem Benutzer bietet, um sich bei einer Unterbrechung der Ausführung über den derzeitigen Stand der Ausführung zu informieren. Hierzu zählt insbesondere die Ausgabe des Wertes einzelner Variablen und Laufzeitstatistikzähler. Bei der bisher beschriebenen Vorgehensweise ist der IPSEN-Benutzer gezwungen, sämtliche Aktivitäten zur Testunterstützung jeweils explizit durch Aufruf eines entsprechenden Kommandos zu aktivieren. Häufig sieht die Testphase jedoch so aus, daß bei jedem erneuten Testlauf vom IPSEN-Benutzer größtenteils dieselben Aktivitäten bei einer Unterbrechung der Ausführung durchgeführt werden. Diese Beobachtung legt es nahe, die Funktionalität der vorgestellten Kommandos derart zu erweitern, daß sie auch **implizit** vom IPSEN-System aktiviert werden können. Hierzu wird der IPSEN-Benutzer nach jeder Aktivierung eines Testunterstützungskommandos gefragt, ob das aktivierte Kommando aufgehoben werden soll und beim nächsten

Erreichen dieser Stelle implizit aktiviert werden soll. Auf diese Weise hat der IPSEN-Benutzer die Möglichkeit, sich schrittweise während der Ausführung eine geeignete **Testumgebung** aufzubauen.

Im Gegensatz zu diesem mehr oder weniger spontanen Aufbau einer Testumgebung bieten wir dem IPSEN-Benutzer darüberhinaus die Möglichkeit an, analog zum Setzen von Unterbrechungsanweisungen bereits vor dem Start der Ausführung eine derartige Testumgebung vorzubereiten. Die hierzu angebotenen Kommandos gehören deshalb auch zum Werkzeug **Testvorbereitung.**
Damit der IPSEN-Benutzer erkennt, an welchen Stellen er eine während der Ausführung implizit zu aktivierende Aktivität im Quelltext eingetragen hat, wird jede dieser Stellen im Quelltext entsprechend kenntlich gemacht. Analog zur Darstellung einer Unterbrechungsanweisung ist auch hier der syntaktische Aufbau wie folgt:

```
(# ... #)
```

In der folgenden Übersicht ist das konkrete Aussehen aller während der Ausführung implizit aktivierbaren Aktivitäten beispielhaft dargestellt:

```
(# inspec:  Last #)
(# inspec:  Schleifel #)
(# dump #)
(# proc. call simulation #)
(# stack size #)
(# heap size #)
(# proc. call size #)
(# data object size: ActStack #)
```

Abb. 2.4.5.1: Konkrete Darstellung implizit aktivierbarer Aktivitäten

Entsprechend existiert für jede dieser Möglichkeiten ein Kommando im Werkzeug Testvorbereitung, um vor (insert) bzw. hinter (extend) dem aktuellen Inkrement einen derartigen Aufruf einzufügen.

T - Testing preparation

Tiv - insert variable inspection	Tev - extend variable inspection
Tid - insert dump	Ted - extend dump
Tip - insert proc. call simulation	
Tik - insert stack size	Tek - extend stack size
Tih - insert heap size	Teh - extend heap size
Tic - insert proc. call size	Tec - extend proc. call size
Tio - insert object size	Teo - extend object size

Abb. 2.4.5.2: Übersicht über Kommandos des Werkzeugs Testvorbereitung

Die meisten dieser Kommandos dürfen aktiviert werden, wenn das aktuelle Inkrement eine beliebige Anweisung ist. Nur das Kommando Tip - insert proc. call simulation macht nur dann Sinn, wenn das aktuelle Inkrement ein Prozeduraufruf ist.

Analog zu bedingten Unterbrechungsanweisungen ist es schließlich auch möglich, die implizite Aktivierung dieser Kommandos von dem Nichterfülltsein einer bestimmten Bedingung abhängig zu machen. Das bedeutet, daß es auch erlaubt ist, ein implizit zu aktivierendes Kommando durch (# assert: (<Expression>) #)/ (# end assert #) zu klammern. Gemäß der im Abschnitt 2.3.2 erläuterten Semantik wird es nur dann aktiviert, wenn die Bedingung nicht erfüllt ist.
Hierzu wird die Funktionalität der im Abschnitt 2.3.2 erläuterten Kommandos Tia - insert assertion bzw. Tea - extend assertion dahingehend verändert, daß bei Aufruf dieser Kommandos das folgende Inkrement im Quelltext eingetragen wird und der Benutzer aufgefordert wird, eine entsprechende Bedingung einzutragen.

```
(# assert: (<Expression>) #)
   (<Implicit Activities>)
(# end assert #)
```

Anschließend hat der IPSEN-Benutzer die Möglichkeit, eine beliebige implizit zu aktivierende Tätigkeit einzutragen, u.a. auch eine Unterbrechungsanweisung.

Wir werden im Paragraphen 2.5 in einem ausführlichen Beispiel erläutern, wie alle diese Hilfsmittel in der Testphase geeignet eingesetzt werden können. Alle diese zusätzlichen Inkremente zur Testunterstützung können mit dem Kommando **Td - delete** wieder gelöscht werden. Soll zum derzeitigen Zeitpunkt die eingetragene Testumgebung nicht angezeigt und bei der Ausführung auch nicht beachtet werden, kann sie mit Hilfe des Kommandos **To - on/off** ausgeschaltet werden.

Analog zur statischen Analyse ist auch für die Ausführung mit umfangreicher Testunterstützung eine sinnvolle Erweiterung des Leistungsumfangs des IPSEN-Systems, die ermittelten Ergebnisse und Informationen nicht nur auf dem Bildschirm anzuzeigen, sondern in einem zusätzlichen **Druckerprotokoll** auszugeben. Diese Möglichkeit sollte von einer späteren Version des IPSEN-Systems unterstützt werden.

2.4.6 Taschenrechner

Einen weiteren Komfort bietet das IPSEN-System durch die Möglichkeit, zu jedem beliebigen Zeitpunkt einen **Taschenrechner** zu aktivieren. Geschieht dies während einer Unterbrechung der Ausführung, besteht zusätzlich die Möglichkeit, die im Programm benutzten Variablen vom Typ CARDINAL, INTEGER oder REAL bei solch einer Nebenrechnung mitzubenutzen. Bei Aktivierung des Taschenrechners (Xt - pocket calculator) wird am Bildschirm ein zusätzliches Eingabefenster eröffnet, in

dem der Benutzer einen beliebigen arithmetischen Ausdruck vom Typ CARDINAL, INTEGER oder REAL eintragen kann. Der berechnete Wert wird dann zusammen mit dem eingegebenen Ausdruck dem Benutzer angezeigt.

2.4.7 Kommandoübersicht

Die in diesem Paragraphen vorgestellten Kommandos der Werkzeuge Testvorbereitung bzw. Ausführung und Test fassen wir in der folgenden Übersicht zusammen:

T - Testing preparation

Tiv - insert variable inspection	Tev - extend variable inspection
Tid - insert dump	Ted - extend dump
Tip - insert proc. call simulation	
Tik - insert stack size	Tek - extend stack size
Tih - insert heap size	Teh - extend heap size
Tic - insert proc. call size	Tec - extend proc. call size
Tio - insert object size	Teo - extend object size
Til - insert loop counter	

X - eXecution

Xv - variable inspection	Xk - stack size
Xd - dump	Xh - heap size
Xp - proc. call simulation	Xc - proc. call size
Xt - pocket calculator	Xo - object size

Abb. 2.4.7.1: Kommandos der Werkzeuge Testvorbereitung bzw. Ausführung und Test

2.5 Integration der Werkzeuge

Wir haben in diesem Kapitel einige Kommandos verschiedener Werkzeuge des IPSEN-Systems vorgestellt. Alle diese Werkzeuge dienen dazu, den IPSEN-Benutzer im Bereich des Programmierens-im-Kleinen zu unterstützen. Die Einteilung der hierbei durchzuführenden Aktivitäten in logisch zusammengehörende Gruppen, d.h. Werkzeuge, dient nur der Übersichtlichkeit an der Benutzerschnittstelle und der internen Strukturierung des IPSEN-Systems. Prinzipiell kann der IPSEN-Benutzer jedes auf dem aktuellen Inkrement gültige Kommando eines beliebigen Werkzeugs

aktivieren. Wir werden im folgenden Abschnitt an einem umfangreichen Beispiel erläutern, wie der IPSEN-Benutzer durch eine geeignete Folge von Kommandos einen Modula-2-Modul erstellen und testen kann. Aufgrund des an dieser Stelle nur begrenzt zur Verfügung stehenden Platzes kann dieses Beispiel nur dazu dienen, **prinzipiell** zu erläutern, wie die bisher vorgestellten Kommandos vor allem in der Testphase geeignet kombiniert werden können.

2.5.1 Beispielsitzung

In den in diesem Kapitel angegebenen Beispielen haben wir häufiger Auszüge aus einem Modula-2-Modul zum Testen eines abstrakten Datenobjekts "Stack" angegeben. Das zugehörige vollständige Programm wollen wir uns in diesem Abschnitt genauer ansehen. Hierbei gehen wir davon aus, daß der bisherige Programmtext durch eine geeignete Folge von Kommandos des syntaxgestützten Editors erstellt worden ist (vgl. /Sc 86/). Die Realisierung dieses abstrakten Datenobjekts enthält die üblichen Zugriffsoperationen eines Kellers, nämlich "Init", "Push", "Pop" und "IsEmpty". Im Hauptprogramm steht eine kurze Folge von Aufrufen dieser Zugriffsoperationen:

```
MODULE StackTest;

  CONST MaxStackLength = 5;

  TYPE Stack = RECORD
                 Last : CARDINAL;
                 CardStack : ARRAY [ 1 .. MaxStackLength ] OF CARDINAL
               END;

  VAR ActStack : Stack;


  PROCEDURE Push ( ActElem : CARDINAL );
  BEGIN
     WITH ActStack DO
        IF Last < MaxStackLength THEN
           Last := Last + 1;
           CardStack [ Last] := ActElem
        END;
     END;
  END Push;

  PROCEDURE Pop ( );
  BEGIN
     ActStack.Last := ActStack.Last - 2;
  END Pop;

  PROCEDURE IsEmpty ( ) : BOOLEAN;
  BEGIN
     RETURN (< Expression >)
  END IsEmpty;

  PROCEDURE Init ( );
  BEGIN
     (< Statement >)
  END Init;
```

```
BEGIN  (* StackTest *)
   Init();
   Push( 1 );
   WHILE NOT IsEmpty() DO
      Pop()
   END;
   Push( 2 );
   Push( 3 );
END StackTest.
```

Abb. 2.5.1.1: Modula-2-Modul zum Testen eines abstrakten Datenobjekts

Der Modul ist noch nicht vollständig erstellt worden. So fehlt z.B. noch vollständig der Rumpf der Prozedur Init und ein Ausdruck in der Return-Anweisung im Rumpf der Funktion IsEmpty. Dennoch möchte der IPSEN-Benutzer bereits zu diesem Zeitpunkt das bis dahin erstellte Programm testen, was im IPSEN-System auch ohne weiteres möglich ist.

Vor Beginn der Ausführung hat der IPSEN-Benutzer die Möglichkeit, mit Hilfe der Kommandos des Werkzeugs Testvorbereitung den Quelltext um den Aufruf von Kommandos zu erweitern, die während der Ausführung bei Erreichen der entsprechenden Stelle implizit zu aktivieren sind. Da der Rumpf der Prozedur Init noch nicht ausgefüllt ist, macht es keinen Sinn, diese Prozedur während der Ausführung zu aktivieren. Aus diesem Grunde wird zunächst der Aufruf der Prozedur Init im Hauptprogramm als aktuelles Inkrement selektiert. Durch Aufruf des Kommandos "Tip - insert proc. call simulation" des Werkzeugs Testvorbereitung wird der Quelltext dann wie folgt verändert. Im folgenden geben wir stets nur den im (hier verkleinerten) aktuellen View dargestellten Ausschnitt des Quelltextes an.

```
BEGIN  (* StackTest *)
   Init();
   Push( 1 );
   WHILE NOT Isempty() DO
      Pop()
   END;
```

Tip →

```
BEGIN  (* StackTest *)
   (# proc. call simulation #);
      Init();
      Push( 1 );
      WHILE NOT IsEmpty() DO
         Pop()
```

Abb. 2.5.1.2: Aufruf des Kommandos Tip - insert proc. call simulation

Durch Aufruf dieses Kommandos wird implizit die Testumgebung angeschaltet und das während der Ausführung implizit zu aktivierende Kommando im Quelltext entsprechend dargestellt.

Danach wird der gesamte Modul als aktuelles Inkrement selektiert und die Ausführung mit dem Kommando "Xr - run" gestartet. Bei Erreichen des Aufrufs der Prozedur Init wird die Ausführung unterbrochen. In einem zweiten Fenster werden die aktuellen Werte der zur Prozedur Init globalen Datenobjekte angezeigt. Alle Cardinal-Werte werden hier automatisch mit 0 vorbesetzt.


```
BEGIN  (* StackTest *)
   (# proc. call simulation #);
   Init();
   Push( 1 );
   WHILE NOT IsEmpty() DO
      Pop()
```

```
MODULE StackTest
variables
ActStack.Last                 :  0
ActStack.CardStack [ 1 ] :  0
ActStack.CardStack [ 2 ] :  0
ActStack.CardStack [ 3 ] :  0
ActStack.CardStack [ 4 ] :  0
ActStack.CardStack [ 5 ] :  0
```

Abb. 2.5.1.3: Impliziter Aufruf von proc. call simulation

Der Benutzer ist mit dieser Vorbesetzung einverstanden und setzt die Ausführung mit dem Kommando "Xr - run" fort. Bei Ausführung der Prozedur IsEmpty wird dann ein zweites Mal mit einer entsprechenden Meldung unterbrochen, weil der Ausdruck in der Return-Anweisung fehlt. Durch Aktivierung eines Editorkommandos trägt der Benutzer in freier Eingabe einen Ausdruck ein, so daß dieser Ausschnitt des Quelltexts wie folgt aussieht:

```
PROCEDURE IsEmpty ( ) : BOOLEAN;
BEGIN
   RETURN (ActStack.Last < 0);
END IsEmpty;
```

Abb. 2.5.1.4: Quelltextausschnitt mit Ausdruck in der Return-Anweisung

Auch hier kann die Ausführung anschließend mit dem Kommando "Xr - run" fortgesetzt werden, bis sie bei Ausführung der Prozedur Push abgebrochen werden muß, weil ein Laufzeitfehler (Bereichsüberschreitung bei Zugriff auf das Feld ActStack.CardStack) aufgetreten ist. Dem Benutzer wird die entsprechende Quelltextposition zusammen mit einer Fehlermeldung angezeigt. Der IPSEN-Benutzer hat nun verschiedene Möglichkeiten, die Ursache des Fehlers zu finden.

Zunächst einmal kann er verhindern, daß bei nochmaliger Ausführung wiederum die Ausführung wegen eines Laufzeitfehlers unterbrochen wird, indem er am Anfang der Prozedur Push eine bedingte Unterbrechungsanweisung einfügt (vgl. auch Beispiel Abb. 2.3.2.1):

```
PROCEDURE Push ( ActElem : CARDINAL );
BEGIN
   WITH ActStack DO
      (# assert: Last >= 0 #)
         (# break #);
      (# end assert #);
```

Abb. 2.5.1.5: Quelltext mit bedingter Unterbrechungsanweisung

Falls beim nächsten Durchlauf ein Laufzeitfehler auftreten würde, weil der Wert von

ActStack.Last kleiner als 0 ist, würde die Ausführung an dieser Stelle zunächst unterbrochen und nicht sofort abgebrochen. Der Benutzer hätte dann z.B. die Möglichkeit, sich die Werte bestimmter Variablen anzusehen.
Weiterhin hat der Benutzer in diesem Fall den Verdacht, daß der Laufzeitfehler aufgetreten ist, weil die im Hauptprogramm stehende while-Schleife zu häufig ausgeführt worden ist. Aus diesem Grunde selektiert er die while-Schleife als aktuelles Inkrement und aktiviert das Kommandos "Til - insert loop counter". Dies bewirkt, daß vor der while-Schleife eine weitere Zeile eingefügt wird und ein Eingabefenster eröffnet wird, in dem der Benutzer einen von ihm gewünschten Bezeichner als Schleifenzähler eintragen kann:

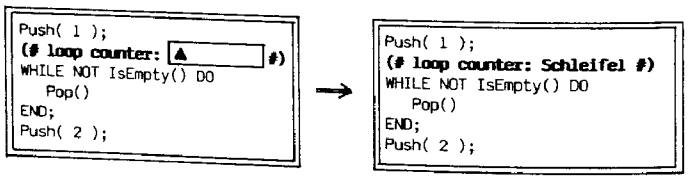


Abb. 2.5.1.6: Quelltext mit Schleifenzähler

Hiernach startet der Benutzer die Ausführung erneut. Um allerdings den Gang der Ausführung besser verfolgen zu können, führt er nun das Programm schrittweise aus: Nach Eingabe des Kommandos "Xs - step" wird zunächst, wie oben erläutert, der Aufruf der Prozedur Init simuliert. Danach wird zweimal das Kommando "Xg - go" eingegeben, so daß sowohl der Aufruf von "Push(1)" als auch die while-Schleife ohne Unterbrechung ausgeführt werden. Nach Beendigung der while-Schleife wird die Ausführung vor dem Aufruf von "Push(2)" unterbrochen. Der Benutzer aktiviert das Kommando "Xv - variable inspection" und läßt sich den Wert des Schleifenzählers ausgeben:

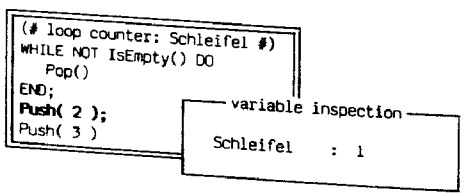


Abb. 2.5.1.7: Ausgabe des aktuellen Wertes eines Schleifenzählers

Da der Schleifenzähler den erwarteten Wert hat, setzt der Benutzer die Ausführung mit dem Kommando "Xs - step" fort. Dies bedeutet, daß auch der Rumpf der Prozedur Push zunächst schrittweise ausgeführt wird. Da die Testumgebung derzeit eingeschaltet ist, wird die bedingte Unterbrechungsanweisung beachtet und in diesem Fall die Ausführung auch unterbrochen. Ein Aufruf des Kommandos "Xv - variable inspection" ergibt, daß ActStack.Last unerwarteterweise den Wert "-1" hat. Damit

der Benutzer beim nächsten Durchlauf dieses Kommando nicht wieder explizit aktivieren muß, wird es auf seinen Wunsch als implizit zu aktivierende Tätigkeit in den Quelltext aufgenommen:

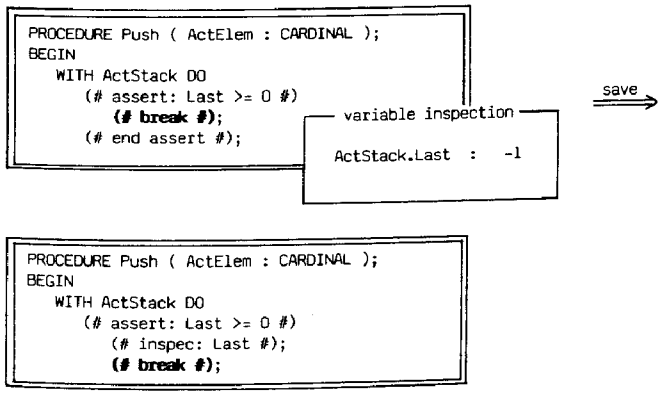


Abb. 2.5.1.8: Aufnahme eines implizit zu aktivierenden Kommandos

Aufgrund des unerwarteten Wertes von "-1" der Variablen ActStack.Last und des einmaligen Durchlaufs der while-Schleife liegt es nahe, einen Fehler in der Realisierung der Prozedur Pop zu vermuten, da nur dort der Wert der Variablen ActStack.Last vermindert wird. Und tatsächlich wurde dort (aus Versehen?) eine 2 anstelle einer 1 bei der Veränderung der Variablen ActStack.Last getippt (vgl. Abb. 2.5.1.1). Mit Hilfe des Editors kann dies leicht korrigiert werden.

In der Hoffnung, daß dies der einzige Fehler war, vervollständigt der IPSEN-Benutzer zunächst den Rumpf der Prozedur Init, löscht mit Hilfe des Kommandos "Td - delete" die implizite Aktivierung der Simulation des Prozeduraufrufs von Init im Hauptprogramm und schaltet mit dem Kommando "To - on/off" die Testumgebung aus. Dies bedeutet, daß alle für Testzwecke im Quelltext eingetragenen Ergänzungen nicht dargestellt werden und während der Ausführung nicht beachtet werden.
Sodann startet der Benutzer die Ausführung des gesamten Moduls erneut mit dem Kommando "Xr - run" und, genau wie im ersten Fall, wird die Ausführung innerhalb der Prozedur Push aufgrund desselben Laufzeitfehlers abgebrochen.
Nun ist der IPSEN-Benutzer aber nicht gezwungen, alle bisher für Testzwecke eingetragenen Ergänzungen des Quelltextes erneut eintragen zu lassen. Durch Aufruf des Kommandos "To - on/off" kann er die **alte Testumgebung** wieder einschalten; hierzu werden im Quelltext die entsprechenden Ergänzungen wieder dargestellt.
Nach einem Neustart der Ausführung mit dem Kommando "Xr - run" wird die Ausführung innerhalb der Prozedur Push unterbrochen, nachdem der Wert von ActStack.Last ausgegeben wurde:

```
PROCEDURE Push ( ActElem : CARDINAL );
BEGIN
   WITH ActStack DO
      (# assert: Last >= 0 #)
         (# inspec: Last #);
         (# break #);
      (# end assert #);
```

variable inspection

ActStack.Last : -1

Abb. 2.5.1.9: Unterbrechung und Ausgabe des Wertes von ActStack.Last

Da der Benutzer sich dieses Verhalten nicht erklären kann, läßt er sich zunächst einen Speicherauszug geben, indem er das Kommando "Xd - dump" aktiviert:

```
PROCEDURE Push ( ActElem : CARDINAL );
BEGIN
   WITH ActStack DO
      (# assert: Last >= 0 #)
         (# inspec: Last #);
         (# break #);
      (# end assert #);
```

dump

MODULE: StackTest	dyn. level: 0
ActStack.Last	: -1
ActStack.CardStack [1]	: 1
ActStack.CardStack [2]	: 0
ActStack.CardStack [3]	: 0
ActStack.CardStack [4]	: 0
ActStack.CardStack [5]	: 0
Schleifel	: 2

Abb. 2.5.1.10: Ausgabe eines Speicherauszugs

Da bei diesem Speicherauszug auch der Wert des Schleifenzählers ausgegeben wird, erkennt der Benutzer, daß die while-Schleife im Hauptprogramm zweimal durchlaufen worden ist. Dies bedeutet, daß zweimal die Operation Pop ausgeführt wurde, obwohl auf dem Stack nur ein Element lag. Also muß die Abbruchbedingung der while-Schleife falsch sein, was sich auch beim Betrachten der Prozedur IsEmpty bestätigt (vgl. Abb. 2.5.1.4). Denn der Ausdruck in der Return-Anweisung muß natürlich "ActStack.Last <= 0" heißen. Nachdem der Benutzer diesen Fehler mit Hilfe des Editors behoben hat, schaltet er mit Hilfe des Kommandos "To - on/off" die Testumgebung wieder aus. Nach einem Neustart mit dem Kommando "Xr - run" wird der Modul dann fehlerfrei ausgeführt.

Das obige Beispiel gibt einen kleinen Einblick, wie der IPSEN-Benutzer die vorgestellten Kommandos der Werkzeuge Editor, Testvorbereitung sowie Ausführung und Test dazu benutzen kann, einen noch unvollständigen Modul zu testen, etwaige Fehler zu beheben und sich hierbei auf verschiedenste Art und Weise vom IPSEN-System unterstützen zu lassen. Hierzu gehört auch die im obigen Beispiel

nicht beschriebene Möglichkeit, Kommandos des Werkzeugs der statischen Analyse zu aktivieren. Charakteristikum des IPSEN-Systems ist hierbei, daß der Benutzer jederzeit ohne Aufwand zwischen den einzelnen Werkzeugen wechseln kann. Weiterhin kann er jederzeit eine Testumgebung ein- bzw. ausschalten, die er im Dialog durch Eingabe entsprechender Kommandos aufgebaut hat und die dadurch auf den aktuell zu testenden Modul zugeschnitten ist.

3 Spezifizieren mit Graph-Ersetzungssystemen

Im letzten Kapitel wurden im Stil einer Anforderungsdefinition eine Reihe von Werkzeugen aus dem Bereich des Programmierens-im-Kleinen des IPSEN-Systems vorgestellt. Hierbei wurde die Funktionalität der einzelnen Kommandos im wesentlichen aus der Sicht eines IPSEN-Benutzers dargestellt. In den folgenden Kapiteln verlassen wir nun die **Sicht eines IPSEN-Benutzers** und wenden uns der Erläuterung der internen Realisierung des IPSEN-Systems zu. In diesem Sinne nehmen wir im folgenden bei der Darstellung die **Position des IPSEN-Entwicklers** ein. Bei der Erläuterung der Aktivitäten des IPSEN-Entwicklers im Rahmen der Realisierung des IPSEN-Systems unterscheiden wir verschiedene Teilaufgaben:
Aufgrund der Komplexität des IPSEN-Systems und der bestehenden Zielsetzung der Adaptabilität und Portabilität des IPSEN-Systems bei geänderten Voraussetzungen beschreiben wir zunächst auf **konzeptioneller Ebene**, welche Datenstrukturen im IPSEN-System gewählt werden und wie sich die entsprechenden Zugriffsoperationen verhalten. Bei einer derartigen **Spezifikation** wird insbesondere Wert auf eine systematische Entwicklung gelegt, um leichte Adaptabilität an geänderte Anforderungen gewährleisten zu können. Es ist Ziel dieses Kapitels, das im Rahmen des IPSEN-Projekts entwickelte Vorgehen beim Aufstellen einer derartigen Spezifikation zu erläutern. Hierbei zeigen wir auf, wie auf systematische Weise ein Erzeugendensystem für eine Klasse von Graphen erstellt werden kann. Diese sogenannten Modulgraphen bilden die zentrale Datenstruktur für alle Werkzeuge des Programmierens-im-Kleinen. In /Sc 86/ wird die hier vorgestellte Spezifikationsmethode angewandt, um das Verhalten des syntaxgestützten Editors zu spezifizieren. In den folgenden Kapiteln dieser Arbeit wird das Verhalten der anderen Werkzeuge bei Benutzung dieser zentralen Datenstruktur Modulgraph weiterhin auf konzeptioneller Ebene beschrieben.
Abschließend folgt dann die Erläuterung, wie eine derartige Spezifikation auf konzeptioneller Ebene umgesetzt werden kann in eine zugehörige **Software-Architektur.** In einer solchen Software-Architektur wird beschrieben, welche Moduln zur Implementierung des IPSEN-Systems benötigt werden und wie die einzelnen Moduln miteinander verbunden sind.

Wir haben erläutert, daß die in IPSEN vorhandene integrierte, modifreie Benutzerschnittstelle dem IPSEN-Benutzer jederzeit einen Wechsel zwischen den einzelnen Werkzeugen ermöglicht. Aus dieser Anforderung an die Benutzerschnittstelle ergeben sich unmittelbar die beiden folgenden **Anforderungen an die Datenstrukturen** zur Speicherung der werkzeugspezifischen Informationen:

a) Alle werkzeugspezifischen Informationen müssen effizient ermittelbar und manipulierbar sein.

b) Inkonsistenzen zwischen den u.U. mehrfach abgelegten Informationen müssen

vermieden werden.

Bezüglich der in a) angesprochenen Effizienz kann unterschieden werden zwischen häufig benutzten Informationen (z.B. Programmstruktur, d.h. Inkremente an der Benutzerschnittstelle, oder kontextfreie/ kontextsensitive Informationen für den Editor) und seltener benutzten Informationen (z.B. set/use-Ketten bei der statischen Analyse). Während erstere Informationen durch einfache Schreib-/Lesezugriffe auf die internen Datenstrukturen ermittelt bzw. verändert werden können sollten, sind bei seltener benutzten Informationen auch umfangreichere, auf den internen Datenstrukturen arbeitende Algorithmen denkbar.

Die im Punkt b) angesprochenen Inkonsistenzen können am einfachsten verhindert werden, wenn alle Informationen nur ein einziges Mal in einer gemeinsamen, zentralen Datenstruktur abgelegt werden. Dies vermeidet unnötige Redundanzen und aufwendige Aktualisierungen anderer Datenstrukturen, wenn eine Datenstruktur modifiziert wurde.

Ein weiterer wichtiger Aspekt ergibt sich aus der Zielsetzung des IPSEN-Projekts, die Entwicklungsumgebung an geänderte Anforderungen wie z.B. einen Wechsel der zu unterstützenden Programmiersprache adaptabel zu machen. Hierbei sind u.a. die beiden folgenden Vorgehensweisen denkbar:
- Entwicklung einer universellen Datenstruktur für zumindest artverwandte Programmiersprachen,
- Entwicklung einer systematischen, leicht zu übertragenden Vorgehensweise bei der Festlegung der internen Datenstruktur zu einer Programmiersprache.

Aufgrund der Vielzahl an unterschiedlichen Sprachelementen auch bei artverwandten Programmiersprachen (z.B. Modula-2, Ada) und der daraus resultierenden Schwierigkeit, eine derartige universelle Datenstruktur zu finden, haben wir uns in IPSEN zunächst für die zweite Möglichkeit entschieden.

Unter Berücksichtigung all dieser Überlegungen wurde in IPSEN auf konzeptioneller Ebene eine graphartige Datenstruktur, der sogenannte **Modulgraph**, als gemeinsames Datenmodell aller Werkzeuge für das Programmieren-im-Kleinen gewählt. Ziel der folgenden Paragraphen ist es, den systematischen Aufbau eines solchen Modulgraphen zu erläutern. In den nächsten Kapiteln wird dann erläutert, welche Vorteile eine solche graphartige Datenstruktur als gemeinsames Datenmodell aller Werkzeuge hat und wie die einzelnen werkzeugspezifischen Informationen in solch einem Datenmodell verwaltet werden können.

3.1 Normierte Backus-Naur-Form

Programmiersprachen werden im allgemeinen durch eine kontextfreie Gram-

matik und eine Menge von zusätzlichen kontextsensitiven Regeln definiert. Zur Darstellung des kontextfreien Anteils werden dabei häufig Syntaxdiagramme oder Backus-Naur-Systeme benutzt. Im Gegensatz zu der durch die Programmiersprache festgelegten Menge von terminalen Symbolen richtet sich die Menge der benutzten nichtterminalen Symbole und die Art der Strukturierung der einzelnen Syntaxdiagramme bzw. Backus-Naur-Produktionen nach den Intentionen des Autors. So ist zum Beispiel der Entwerfer eines Compilers evtl. gezwungen, eine Reihe von zusätzlichen, technischen nichtterminalen Symbolen einzuführen, um die LL(1)-Eigenschaft der zugrundeliegenden Grammatik zu erhalten.

Um nun eine fundierte Basis für die weiteren Überlegungen zu haben, fordern wir in IPSEN eine bestimmte Gestalt der Ausgangsgrammatik. Bei der Festlegung einer derartigen normierten Form der Grammatik wurden wir von der Erkenntnis geleitet, daß sich die für einen Benutzer üblichen syntaktischen Einheiten beim Edieren eines Programms in der Form und Strukturierung einer derartigen Ausgangsgrammatik widerspiegeln sollten. Wir haben diesen Zusammenhang zwischen Benutzerschnittstelle und Ausgangsgrammatik bereits in /EG 83/ ausführlich erläutert, so daß wir im folgenden an den entsprechenden Stellen nur noch kurz darauf zurückkommen.

Zur Darstellung einer Grammatik benutzen wir im folgenden die übliche erweiterte Backus-Naur-Form (EBNF). Eine **EBNF heißt normiert,** wenn die Menge der nichtterminalen Symbole in drei disjunkte Teilmengen der folgenden Arten zerlegt werden kann:

a) Auswahl-Nonterminal:

Diese Teilmenge zerfällt in zwei weitere disjunkte Teilmengen:

- **Alternativen-Nonterminal:**

  Die rechte Seite der zugehörigen EBNF-Produktion besteht aus einer Reihe von Alternativen, wobei entweder alle Alternativen nichtterminale Symbole oder alle Alternativen terminale Symbole sind.

- **Optionales Nonterminal:**

  Die rechte Seite der zugehörigen EBNF-Produktion besteht aus zwei Alternativen, dem leeren Wort und dem optionalen Teil.

b) Struktur-Nonterminal:

Die rechte Seite der zugehörigen EBNF-Produktion besteht aus einer nicht-leeren Folge von nichtterminalen und terminalen Symbolen.

c) Listen-Nonterminal:

Die rechte Seite der zugehörigen EBNF-Produktion beschreibt eine nicht-leere Liste und die Produktion hat die Gestalt:

```
<elem_list>  ::=  <elem>  { 'delimiter' <elem> }
```

Es ist offensichtlich, daß jede gegebene Programmiersprachengrammatik in diese

normierte Form gebracht werden kann, indem in geeigneter Weise nichtterminale Symbole hinzugefügt oder gestrichen werden und die Produktionen hierarchisiert oder abgeflacht werden. Im Anhang A befindet sich eine derartige normierte EBNF für den Programmieren-im-Kleinen-Anteil von Modula-2, die ausgehend von der in /Wi 82/ vorgefundenen EBNF erstellt wurde. Zur näheren Erläuterung der obigen disjunkten Zerlegung der nichtterminalen Symbole seien die folgenden Beispiele zitiert:

ad a) Alternativen-Nonterminals sind z.B. <statement> und <letter>.
Die entsprechenden EBNF-Produktionen sind:

```
<statement>  ::=  <assignment_statement> | <procedure_call>  |
                  <if_statement>         | <case_statement>  |
                  <while_statement>      | <repeat_statement>|
                  <loop_statement>       | <for_statement>   |
                  <with_statement>       | <exit_statement>  |
                  <return_statement>

<letter>     ::=  A | B | ... | Y | Z
```

Zur Kennzeichnung eines optionalen Nonterminals beginnt der Name eines solchen nichtterminalen Symbols stets mit 'opt_', z.B.:

```
<opt_statement_list>  ::=  ε | <statement_list>
```

Die eigentliche Struktur eines Programms wird durch Struktur- und Listen-Nonterminals beschrieben:

ad b) Ein Struktur-Nonterminal ist z.B. <if_statement>, dessen zugehörige EBNF-Produktion lautet:

```
<if_statement>  ::=  IF  <expression>  THEN
                         <opt_statement_list>
                     <opt_elsif_part_list>
                     <opt_else_part>
                     END
```

ad c) Zur Kennzeichnung eines Listen-Nonterminals endet der Name eines solchen nichtterminalen Symbols stets mit '_list', z.B.:

```
<statement_list>  ::=  <statement> { ; <statement> }
```

Die Forderung nach einer disjunkten Zerlegung der Menge der nichtterminalen Symbole impliziert **keine eindeutige** normierte EBNF. Dies gilt vor allem für die Frage, an welcher Stelle in der EBNF eine Entscheidung zwischen mehreren Alternativen verkapselt wird. Als Beispiel seien hier zwei unterschiedliche Möglichkeiten für die Beschreibung einer if-Anweisung ohne elsif-part-Liste und mit bzw. ohne else-Teil diskutiert: In den oben zitierten EBNF-Produktionen ist auf der Ebene eines "statements" nur eine beliebige if-Anweisung bekannt, die auf der if-Anweisungsebene aufgrund der optionalen Nonterminals <opt_else_part> und

<opt_elsif_part_list> dann mit einem else-Teil bzw. einer elsif-part-Liste versehen werden kann oder nicht. Eine zweite Möglichkeit wäre gewesen, auf der Ebene eines "statements" bereits zwischen den Alternativen "if_statement", "if_then_statement" und "if_then_else_statement" zu unterscheiden. Die beiden zusätzlichen EBNF-Produktionen würden dann wie folgt aussehen:

```
<if_then_statement>  ::= IF  <expression>  THEN
                             <opt_statement_list>
                         END
bzw.
<if_then_else_statement> ::= IF  <expression>  THEN
                                 <opt_statement_list>
                             ELSE
                                 <opt_statement_list>
                             END
```

Im nächsten Paragraphen werden wir erläutern, wie aus einer solchen normierten EBNF die Gestalt des Modulgraphen abgeleitet wird. Daran und an der anschließenden Erläuterung, wie weitere werkzeugspezifische Informationen in diesem Modulgraphen abgelegt werden können, wird deutlich werden, welche Auswirkung die gewählte normierte EBNF auf das Aussehen des Modulgraphen hat.

3.2 Systematische Entwicklung eines Erzeugendensystems

Ein wesentliches Entwurfsziel des IPSEN-Projekts ist, dem Benutzer eine syntaxgestützte Bearbeitung seiner Dokumente zu ermöglichen. Um dies effizient realisieren zu können, muß aus der internen Darstellung des aktuellen Dokuments die syntaktische Struktur leicht ermittelbar sein.
Die bei Texteditoren übliche Darstellung eines Dokuments als Zeichenstrom kann deshalb in IPSEN nicht geeignet sein. Bei der Ermittlung der syntaktischen Struktur eines Dokuments, in der Regel eines Programms, wird bei in Compilern enthaltenen Parsern gewöhnlich der Ableitungsbaum (vgl. /AU 79/) als interne Datenstruktur benutzt. Die darin enthaltene, vollständige Beschreibung des Ableitungsprozesses ist jedoch zur Beschreibung der reinen syntaktischen Struktur überflüssig. Aus diesem Grunde wird in den meisten Softwareentwicklungsumgebungen ein sogenannter **abstrakter Syntaxbaum** als interne Datenstruktur benutzt. Diese Vorgehensweise ist vor allem im Projekt MENTOR (vgl. /DG 80/) formalisiert worden. Die dabei entstehenden abstrakten Syntaxbäume unterscheiden sich in zwei wesentlichen Punkten von üblichen Ableitungsbäumen:

a) Die in Ableitungsbäumen enthaltene, vollständige Beschreibung des Ableitungsprozesses über eine Folge von z.T. rein technischen Produktionsanwendungen wird abgeflacht auf eine Beschreibung der eigentlichen syntaktischen Struktur.
b) Sämtliche in Ableitungsbäumen enthaltene konkrete Syntax wird eliminiert.

Um eine externe Repräsentation des durch den abstrakten Syntaxbaum beschriebenen Dokuments zu erhalten, muß in einem Tranformationsprozeß, auch **Unparsing** genannt, die fehlende konkrete Syntax sowie ein entsprechendes Layout erzeugt werden. Hierauf gehen wir im Kapitel 6 ausführlich ein.
Alle weiteren Informationen zu einem Dokument werden in zusätzlichen Knoten- bzw. Kantenattributen zu einem solchen abstrakten Syntaxbaum abgelegt und verwaltet. Dies sind bei einem Programm z.B. kontextsensitive Beziehungen wie Deklaration zum angewandten Auftreten eines Datenobjekts oder andere werkzeugspezifische Informationen.

3.2.1 Graphinkremente

Auch in IPSEN liegt der internen Darstellung eines Programms, dem Modulgraphen, im Prinzip ein abstrakter Syntaxbaum zugrunde. Da wir jedoch nicht nur kontextfreie, sondern sämtliche strukturellen Beziehungen in der internen Datenstruktur durch Knoten und Kanten ausdrücken wollen, verzichten wir darauf, zunächst eine abstrakte Syntax anzugeben, und wählen die folgende Vorgehensweise:

Im Kapitel 2 haben wir erläutert, daß sich alle Werkzeugaktivitäten an der Inkrementeinteilung eines Programms orientieren. Diese inkrementorientierte Arbeitsweise wird am deutlichsten bei der Arbeit mit dem syntaxgestützten Editor, da jede Veränderung eines Programms die Veränderung einer syntaktischen Einheit, d.h. eines **Edierinkrements,** bedeutet (vgl. hierzu /Sc 86/). Somit liegt es nahe, auch in der internen Darstellung eine Inkrementeinteilung zu kennen, um so die inkrementorientierte Arbeitsweise einzelner Werkzeuge geeignet zu unterstützen. Ausgehend von der oben eingeführten normierten EBNF führen wir drei verschiedene Arten von Graphinkrementen ein, nämlich atomare, Struktur- und Listen-Graphinkremente:

In der EBNF befindet sich das unterste Strukturierungsniveau auf Zeichen- bzw. Ziffernebene. Da die Feinstruktur von Bezeichnern und Literalen nur bei der Eingabe zur Überprüfung auf syntaktische Korrektheit bekannt sein muß, aber später von keinem auf dem Modulgraphen arbeitenden Werkzeug benötigt wird, wird im Modulgraphen diese Feinstruktur nicht dargestellt. Dies bedeutet, daß im Modulgraph Bezeichner und Literale als nicht weiter strukturierte Inkremente, also **atomare Graphinkremente,** angesehen werden. Die Darstellung eines solchen atomaren Graphinkrements besteht aus einem einzelnen Knoten. Zur Markierung der Knoten in den Graphinkrementen benutzen wir im folgenden die nichtterminalen Symbole der normierten EBNF bzw. entsprechende Abkürzungen. Der in einem atomaren Graphinkrement enthaltene Knoten wird somit z.B. mit "Ident" oder "String" markiert. Ein dazugehörender Bezeichner bzw. Literal wird in einem Attribut

"Name" an solch einem Knoten abgelegt:

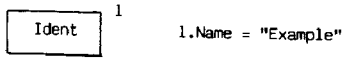


Abb. 3.2.1.1: Darstellung eines atomaren Graphinkrements

(Erläuterung: Der Knoten mit der Knotenbezeichnung 1 hat die Markierung "Ident" und das Attribut "Name" mit dem Attributwert "Example". Falls der zu einem Attribut gehörende Knoten offensichtlich ist, schreiben wir nur "Name" anstatt "1.Name".)

Alle nichtterminalen Symbole, die in einer im üblichen Grammatik-Sinne mit den nichtterminalen Symbolen <ident>, <natural>, <rational> oder <string> startenden Ableitung auftreten und damit zur Beschreibung der Feinstruktur lexikalischer Einheiten dienen, brauchen dann bei der Graphinkrementfestlegung nicht weiter berücksichtigt zu werden.

Allen übrigen in der normierten EBNF enthaltenen Struktur-Nonterminals werden **Struktur-Graphinkremente** zugeordnet. Diese Graphinkremente werden durch einen Baum dargestellt, dessen Wurzel mit diesem Struktur-Nonterminal (bzw. einer entsprechenden Abkürzung) markiert ist. Entsprechend der Anzahl der nichtterminalen Symbole auf der rechten Seite der zu diesem Struktur-Nonterminal gehörenden EBNF-Produktion besitzt diese Wurzel entsprechend viele mit diesen nichtterminalen Symbolen markierte Söhne. Zusätzlich werden die auftretenden Kanten mit geeigneten, der Bedeutung eines Sohnes entsprechenden Kantenmarkierungen versehen. Zur Unterscheidung von Knoten- und Kantenmarkierungen beginnen die Kantenmarkierungen mit "E" (für Edge).

```
EBNF-Produktion:  <while_statement> ::= WHILE <expression> DO
                                            <opt_statement_list>
                                        END
```

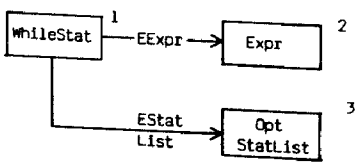


Abb. 3.2.1.2: Darstellung eines Struktur-Graphinkrements

(Bem.: Entgegen der üblichen, vertikalen Schreibweise notieren wir Bäume und Graphen horizontal.)

Allen in der normierten EBNF enthaltenen Listen-Nonterminals werden **Listen-Graphinkremente** zugeordnet. Diese Listen-Graphinkremente werden durch Wurzelgraphen dargestellt, wobei der Wurzelknoten mit dem Listen- Nonterminal markiert

ist und die einzelnen Listenelemente als Söhne an diesen Wurzelknoten angehängt werden. Während bei abstrakten Syntaxbäumen durch die Benutzung geordneter Bäume eine Reihenfolge zwischen den Söhnen eines Knotens festgelegt ist, verzichten wir auf diese zusätzliche formale Forderung eines Ordnungsbegriffs. Bei Struktur-Graphinkrementen ist durch die eindeutige Markierung der Kanten zu den Söhnen eine implizite Ordnung der Söhne gegeben. Bei Listen-Graphinkrementen wird die Reihenfolge zwischen Listenelementen durch zusätzliche, hier mit "ENext" markierte Kanten beschrieben. Außerdem wird das erste bzw. letzte Listenelement durch eine mit "EFirst" bzw. "ELast" markierte Kante mit dem Wurzelknoten verbunden.

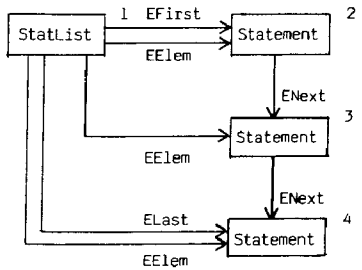


Abb. 3.2.1.3: Darstellung eines Listen-Graphinkrements mit drei Listenelementen

Diese drei Arten von Graphinkrementen bilden die Bausteine, aus denen der Modulgraph zusammengesetzt wird. Dieses Zusammensetzen von Graphinkrementen geschieht durch Ersetzen eines Knotens (genauer eines Blatts bzw. einer Senke) in einem Graphinkrement durch ein an dieser Stelle erlaubtes anderes Graphinkrement. Durch wiederholte Ausführung derartiger Ersetzungsschritte werden die oben eingeführten Graphinkremente zu größeren Graphen expandiert. Im Gegensatz zu den oben erläuterten Graphinkrementen nennen wir derartige Graphinkremente dann auch **expandierte Graphinkremente.** Bevor wir im Abschnitt 3.2.3 diesen Ersetzungsmechanismus präzisieren und ein Erzeugendensystem für die Klasse der Modulgraphen angeben, betrachten wir in Abb. 3.2.1.4 ein kleines Beispiel eines Modulgraphen als Darstellung eines kurzen, z.T. noch unvollständigen Programmausschnitts.

Die in einem Modulgraphen auftretenden Knotenmarkierungen sind neben den Markierungen zur Kennzeichnung atomarer Graphinkremente (z.B. Ident, Natural) in erster Linie (Abkürzungen von) Struktur-Nonterminals (z.B. WhileStat, MinSimpExpr) und Listen-Nonterminals (z.B. StatList) der normierten EBNF. Zur Kennzeichnung noch existierender Quelltextlücken sind die entsprechenden Modulgraphknoten mit (Abkürzungen von) Alternativen- (z.B. Expr) oder optionalen Nonterminals (z.B.

OptStatList) markiert. Am Ende der Arbeit befindet sich ein Verzeichnis der in den Modulgraphbeispielen verwandten Abkürzungen für Knoten- bzw. Kantenmarkierungen.

```
WHILE X > 0 DO
   REPEAT
      (< Statement >)
   UNTIL (< Expression >);
   X := X - 1
END
```

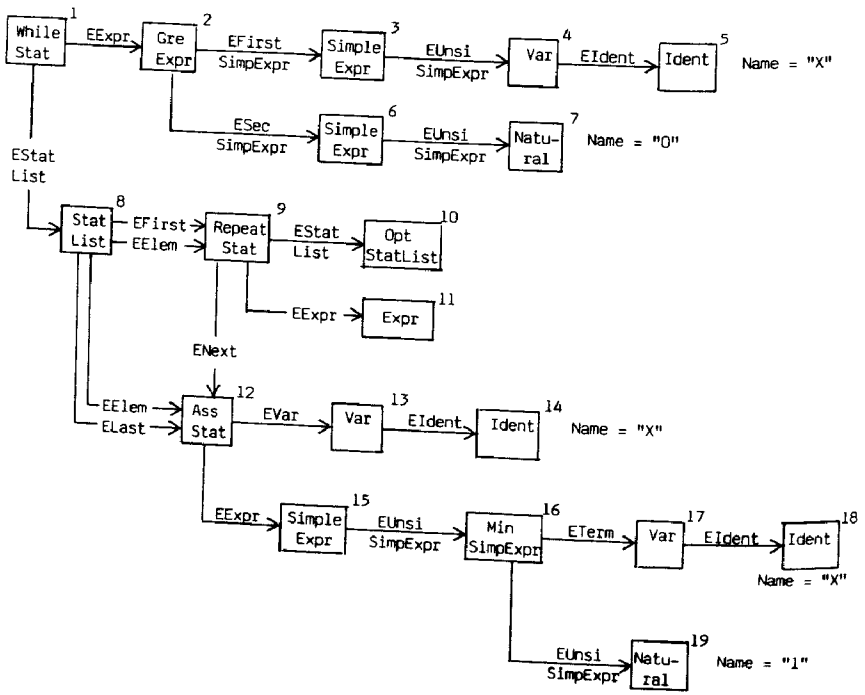


Abb. 3.2.1.4: Beispiel eines Modulgraphausschnitts

3.2.2 Graphentheoretische Grundlagen

In diesem Abschnitt werden einige aus der Graphentheorie bekannte Definitionen zitiert, die im folgenden wiederholt benutzt werden. Weitere, hier nicht aufgenommene Definitionen befinden sich in der entsprechenden Literatur, wobei insbesondere auf Abschnitt I.1 in /Na 79/ hingewiesen sei.

Im folgenden bezeichne **Nodelabels** eine endliche Menge von Knotenmarkierungen,

Edgelabels eine endliche Menge von Kantenmarkierungen, **Attributes** eine endliche Menge von Attributen und die Funktion **nodeatt :** Nodelabels -> $\mathcal{P}$(Attributes) eine Attributzuordnungsfunktion.

Def. 3.2.2.1: (gakk-Graph)

Ein **gerichteter, attributierter, knoten- und kantenmarkierter Graph** (gakk-Graph) über Nodelabels, Edgelabels, Attributes, nodeatt ist ein Tripel
G = (Node, nodelab, Edges) mit
- Nodes endliche Menge von Knotenbezeichnungen,
- nodelab : Nodes -> Nodelabels Knotenmarkierungsfunktion,
- Edges $\subseteq$ Nodes x Nodes x Edgelabels markierte Kantenmenge.

Im folgenden wird als Knotenbezeichnungsmenge Nodes in der Regel {1,...,m} gewählt. Ein Element (k1,k2,el) $\in$Edges wird als gerichtete Kante vom Knoten k1 zum Knoten k2 aufgefaßt, die mit el markiert ist. Mit **G (Nodelabels, Edgelabels, Attributes, nodeatt)** bezeichnen wir die Menge aller gakk-Graphen über Nodelabels, Edgelabels, Attributes und nodeatt.

Def. 3.2.2.2: (Attributierung)

Sei nl $\in$ Nodelabels eine Knotenmarkierung mit einer Attributzuordnung nodeatt(nl) = {A1, ..., An}. Der zu einem Attribut Ai gehörende Wertebereich sei mit Ai-Werte bezeichnet. Eine **aktuelle Attributierung** eines Knotens node mit der Markierung nl ist ein n-Tupel (a1_wert,..., an_wert) $\in$ A1_Werte x ... x An_Werte und der aktuelle Wert eines Attributs Ai ist ai_wert.
(Schreibweise: node.Ai = ai_wert bzw. Ai = ai-wert)

Die in IPSEN benutzten Modulgraphen als interne Darstellung eines Programms sind gerichtete, attributierte, knoten- und kantenmarkierte Graphen mit einer aktuellen Attributierung der einzelnen Knoten (vgl. Abb. 3.2.1.4). Hierbei ist durch die Attributzuordnungsfunktion nodeatt festgelegt, daß alle Knoten mit derselben Markierung dieselben Attribute besitzen, wobei die aktuelle Attributierung verschiedener Knoten mit derselben Markierung natürlich unterschiedlich sein kann. Wir werden im folgenden an verschiedenen Stellen darauf zurückkommen, welche Informationen in einem Modulgraphen in derartigen Attributen abgelegt werden. Als Wertebereich für Attribute sind bisher die Wertebereiche der in Modula-2 vorhandenen Standarddatentypen (Cardinal, Integer, Real, Char, Boolean) sowie der Typ String als Folge von Zeichen (Char) vorgesehen. Das oben eingeführte Attribut Name bei atomaren Graphinkrementen hat z.B. als Wertebereich den Typ String. Zur Darstellung von Attributwerten sind die in Modula-2 erlaubten konstanten Ausdrücke auch hier erlaubt (z.B. node.Size = ((4+2) DIV 2)).

Def. 3.2.2.3: (Untergraph)

Seien U und G zwei gakk-Graphen aus G (Nodelabels, Edgelabels, Attributes, nodeatt).

U = (NU, nlU, EU) ist **Untergraph** von G = (NG, nlG, EG), falls

$NU \subseteq NG$, $nlG|_{NU} = nlU$ und $EG|_{NUxNUxEdgelabels} = EU$

Das heißt insbesondere, daß alle Kanten in U mit derselben Markierung in G enthalten sind.

Zwei gakk-Graphen heißen **strukturäquivalent**, wenn lediglich ihre Knoten unterschiedlich bezeichnet sind bzw. die aktuelle Attributierung entsprechender Knoten unterschiedlich ist.

Def. 3.2.2.4: (hinreichend äquivalent)

Ein gakk-Graph U zusammen mit einer Menge aktueller Attributwerte AU ist **enthalten** in einem gakk-Graphen G, wenn
- U strukturäquivalent zu einem Untergraphen GU von G ist,
- die durch Au gegebene aktuelle Attributierung einiger Knoten in U übereinstimmt mit der aktuellen Attributierung der entsprechenden Knoten in GU.

U und GU heißen dann auch **hinreichend äquivalent.**

In diesem Fall gilt für den Graphen U, daß neben der Strukturäquivalenz zu einem Untergraphen GU von G bestimmte aktuelle Attributwerte mit den entsprechenden Attributwerten in GU übereinstimmen. Die restliche Attributierung der Graphen U und GU kann dann u.U. verschieden sein.

3.2.3 Erzeugendensystem

Eine EBNF ist eine mögliche, im Programmiersprachenbereich häufig angewandte Darstellungsform einer kontextfreien Zeichenketten-Grammatik. Zusammen mit einem dazugehörenden Ableitungsbegriff wird durch eine EBNF festgelegt, wie aus einem nichtterminalen Startsymbol unter Benutzung der EBNF-Produktionen ein terminales Wort abgeleitet werden kann. Im Abschnitt 3.2.1 haben wir bisher nur erläutert, wie aus den Produktionen einer normierten EBNF unmittelbar drei verschiedene Arten von Graphinkrementen hergeleitet werden können. Aus diesem Grunde muß nun auch der Ableitungsbegriff von Zeichenketten auf Graphen verallgemeinert werden, um beschreiben zu können, wie die Graphinkremente zu einem Modulgraphen zusammengesetzt werden können, so daß die durch die EBNF festgelegte kontextfreie Struktur beachtet wird.

Ziel dieses Abschnitts ist es deshalb zu erläutern, wie aus einer vorliegenden

normierten EBNF, also Zeichenketten-Grammatik, eine entsprechende Graph-Grammatik gewonnen werden kann. Eine umfassende Behandlung der theoretischen Grundlagen von Graph-Grammatiken findet man in /Na 79/. Wir werden im nächsten Abschnitt 3.2.4 die wichtigsten Grundlagen für das Verständnis dieser Arbeit zusammenfassen bzw. neu einführen.

Wir haben bei der Erläuterung der Graphinkremente gesehen, daß alle nichtterminalen Symbole der normierten EBNF oberhalb der Ebene zur Beschreibung lexikalischer Einheiten als Knotenmarkierung in den Graphinkrementen stehen können. Damit besteht die Menge der Knotenmarkierungen Nodelabels aus fünf disjunkten Teilmengen, nämlich:

- Atom - die Menge der Nonterminals zur Kennzeichnung atomarer Graphinkremente,
- Alter - die Menge der Alternativen-Nonterminals
- Opt - die Menge der optionalen Nonterminals
- List - die Menge der Listen-Nonterminals
- Struct - die Menge der Struktur-Nonterminals

} der normierten EBNF, jeweils bis zur Ebene der in Atom enthaltenen Nonterminals

Analog zur Einteilung in nichtterminale und terminale Symbole bei Zeichenketten-Grammatiken unterscheiden wir nun auch hier zwischen **terminalen** und **nichtterminalen** Knotenmarkierungen, um zu kennzeichnen, welche Knoten in einem Graphen noch weiter ersetzt werden können:

Alle mit einem in der Menge Alter enthaltenen Alternativen-Nonterminal (z.B. Statement) markierten Knoten können durch das zu einer Alternative (z.B. WhileStat) gehörende Graphinkrement ersetzt werden. Formal wird das durch die folgende Graph-Produktion beschrieben:

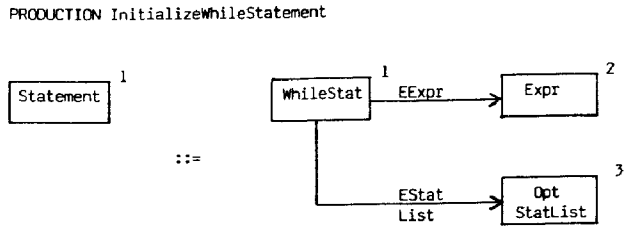


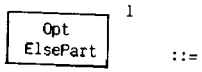
Abb. 3.2.3.1: Graph-Produktion zum Einfügen eines Struktur-Graphinkrements

Bei Anwendung dieser **Graph-Produktion** auf einen Graphen wird ein mit "Statement" markierter Knoten durch das Graphinkrement auf der rechten Seite der Produktion ersetzt. Zusätzlich muß bei der Anwendung einer Graph-Produktion festgelegt

werden, wie der eingesetzte Graph mit dem restlichen Graph verbunden wird. Dies kann im allgemeinen Fall durch eine sogenannte "**Einbettungsüberführung**" festgelegt werden, die z.B. in /Na 79/ formal eingeführt wird. Wir benötigen hier zunächst stets nur die identische Einbettung, d.h. alle ein- und auslaufenden Kanten in den (bzw. die) ersetzten Knoten bleiben bei den eingesetzten Knoten mit derselben Knotenbezeichnung erhalten.

Weiterhin gehören alle in der Menge Opt enthaltenen optionalen Nonterminals (z.B. OptElsePart) zur Menge der nichtterminalen Knotenmarkierungen. Zu jedem so markierten Knoten gibt es zwei Graph-Produktionen, zum Löschen dieses Knotens bzw. zum Eintragen des optionalen Teils (z.B. else_part):

```
PRODUCTION DeleteOptElsePart
```




```
PRODUCTION InitializeElsePart
```

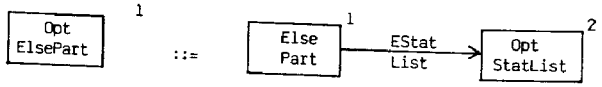
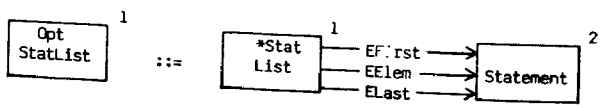


Abb. 3.2.3.2: Graph-Produktionen für optionale Teile

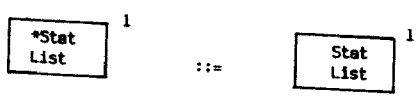
Beim Löschen des Knotens in der Produktion DeleteOptElsePart werden auch alle ein- bzw. auslaufenden Kanten gelöscht.

Bei Listen-Graphinkrementen muß es möglich sein, Listen mit beliebig vielen Elementen erzeugen zu können. Hierzu werden bei jedem Listen-Nonterminal (z.B. StatList) vier Graph-Produktionen benötigt und eine weitere Knotenmarkierung (z.B. *StatList) als nichtterminale Knotenmarkierung. Alle diese zusätzlichen Knotenmarkierungen werden in einer Menge *_List zusammengefaßt.

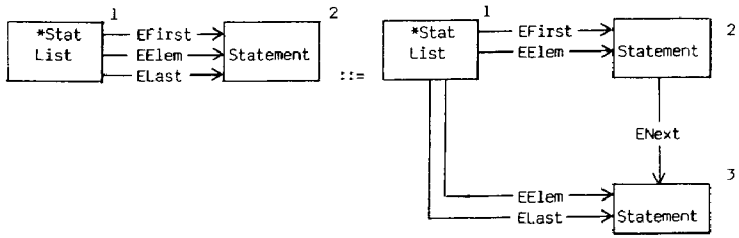
```
PRODUCTION InitializeStatementList
```




```
PRODUCTION TerminateStatementList
```




```
PRODUCTION AppendSecondStatement
```




```
PRODUCTION AppendAnotherStatement
```

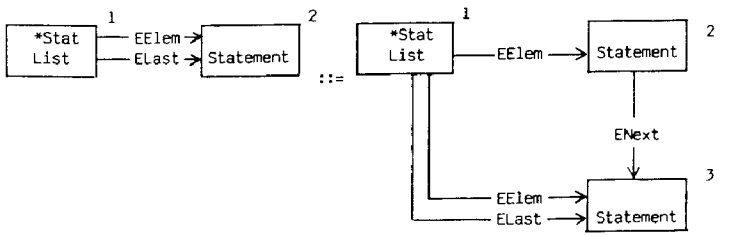


Abb. 3.2.3.3: Produktionen zur Erzeugung einer Liste von statements

Bei atomaren Graphinkrementen (z.B. Bezeichnern) muß zusätzlich der Wert des Attributs Name mit einem Objekt des Typs String gesetzt werden. Um nun nicht für jeden erlaubten Bezeichner eine eigene Graph-Produktion angeben zu müssen, führen wir **parametrisierte Graph-Produktionen** ein. Diese enthalten zusätzlich eine Liste von call-by-value-Parametern, die bei Anwendung der Graph-Produktion durch aktuelle Parameter, d.h. aktuelle Attributwerte, zu ersetzen sind. Zur Unterscheidung von bereits attributierten bzw. noch zu attributierenden atomaren Graphinkrementen wird auch hier eine weitere nichtterminale Knotenmarkierung (z.B. *Ident) benötigt. Alle diese zusätzlichen Knotenmarkierungen werden in einer Menge *_Atom zusammengefaßt.

```
PRODUCTION InitializeIdent (ActIdent : String))
```

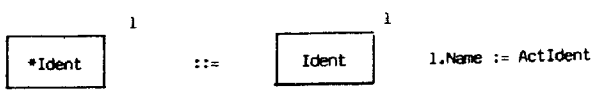


Abb. 3.2.3.4: Beispiel einer parametrisierten Graph-Produktion

Die Zuordnung eines neuen Attributwerts wird in Form einer Modula-2 Zuweisungsanweisung notiert.

Analog zu den oben beispielhaft angegebenen Graph-Produktionen kann von einer gegebenen normierten EBNF systematisch eine Menge von Graph-Produktionen abgeleitet werden. Zusammen mit einem Startgraphen bildet diese Menge von Graph-Produktionen dann eine Graph-Grammatik, durch die eine Menge von Graphen, die von dieser Graph-Grammatik erzeugte Sprache, festgelegt ist. Diese Begriffe werden im nächsten Abschnitt präzisiert.

3.2.4 Graph-Grammatik-Grundlagen

In diesem Abschnitt werden einige Begriffe aus der Theorie der Graph-Grammatiken zitiert; für weitergehende Definitionen sei auch hier insbesondere auf Abschnitt I.2 in /Na 79/ hingewiesen.

Def. 3.2.4.1: (Graph-Produktion)

Eine **Graph-Produktion** über Nodelabels, Edgelabels, Attributes, nodeatt ist ein Quintupel p = (Gl, Al, Gr, E, att) mit

- Gl aus G (Nodelabels, Edgelabels, Attributes, nodeatt), linke Seite der Produktion,
- Al eine Menge aktueller Attributwerte von Attributen an Knoten aus Gl,
- Gr = (Nr, nlr, Er) aus G (Nodelabels, Edgelabels, Attributes, nodeatt), rechte Seite der Produktion,
- E Einbettungsüberführung, die beschreibt, welche Knoten von Gr über welche Kanten mit bestimmten Knoten des Wirtgraphen bei einer Anwendung dieser Produktion zu verbinden sind (näheres siehe /Na 79/)
- att : Nr x Attributes -> Attributes-Values partielle Funktion mit Attributes-Values Menge aller Attributwerte von Attributes, die Knoten node $\in$ Nr und einem zugeordneten Attribut Ai $\in$ nodeatt(nlr(node)) einen neuen Wert ai_wert $\in$ Ai_Werte zuweist.

Da wir in der Regel als Einbettungsüberführung nur die Identität benötigen, verzichten wir hier auf eine Festlegung einer formalen Notation. Die partielle Funktion att legt fest, welche aktuellen Attributwerte bei Anwendung dieser Graph-Produktion zu modifizieren sind. Sie wird in der Gestalt einer Folge von Modula-2-Zuweisungsanweisungen notiert (vgl. Abb. 3.2.3.4).

Bei einer **parametrisierten Graph-Produktion** wird eine Graph-Produktion um eine Liste formaler Parameter ergänzt, die dann auch bei der Festlegung der Menge der aktuellen Attributwerte bzw. der partiellen Funktion att auftreten können (vgl.

Abb. 3.2.3.4 bzw. Abb. 3.2.7.6). Vor Anwendung einer solchen parametrisierten Graph-Produktion müssen dann die formalen Parameter durch aktuelle Attributwerte ersetzt werden (call-by-value Übergabemechanismus), so daß eine Graph-Produktion im obigen Sinne entsteht.

Analog zu Zeichenketten-Grammatiken muß auch bei Graph-Grammatiken definiert werden, wie ein vorliegender Graph bei Anwendung einer Graph- Produktion verändert wird.

Def. 3.2.4.2: (Ableitungsbegriff)

Ein gakk-Graph G' wird aus einem gakk-Graphen G mittels einer Graph-Produktion p = (Gl, Al, Gr, E, att) **abgeleitet** (Schreibweise: G -> G'), indem

- ein zu Gl und Al hinreichend äquivalenter Untergraph in G durch einen zu Gr strukturäquivalenten Untergraphen ersetzt wird und gemäß der Einbettungsüberführung E mit dem restlichen Graphen verbunden wird,
- die aktuelle Attributierung der Knoten des zu Gr strukturäquivalenten Untergraphen in G übernommen ist von der aktuellen Attributierung entsprechender Knoten (d.h. mit derselben Knotenbezeichnung) des zu Gl strukturäquivalenten Untergraphen in G,
- die aktuelle Attributierung einiger Knoten des zu Gr strukturäquivalenten Untergraphen gemäß der partiellen Funktion att modifiziert wird.

Damit sind wir in der Lage, eine Graph-Grammatik wie folgt zu definieren:

Def. 3.2.4.3: (attributierte Graph-Grammatik)

Eine **attributierte Graph-Grammatik** ist ein 7-Tupel
AGG = (Nodelabels, Edgelabels, Attributes, nodeatt, P, G0, ->) mit

- Nodelabels, Edgelabels endliche Menge von Knoten- bzw. Kantenmarkierungen,
- Attributes endliche Menge von Attributen,
- nodeatt: Nodelabels -> $\mathcal{P}$(Attributes) Attributzuordnungsfunktion,
- P endliche Menge von (parametrisierten) Graph-Produktionen über Nodelabels, Edgelabels, Attributes und nodeatt,
- G0 aus G (Nodelabels, Edgelabels, Attributes, nodeatt), der Startgraph,
- -> der in Def. 3.2.4.2 eingeführte Ableitungsbegriff.

Zu dieser Definition des Ableitungsbegriffs einige Bemerkungen: Eine Graph-Produktion ist anwendbar, wenn zur linken Seite der Produktion ein Untergraph im aktuellen Graphen existiert, dessen Struktur mit der linken Seite übereinstimmt und einige der aktuellen Attributwerte entsprechender Knoten identisch sind. Durch diese Möglichkeit der Beschränkung ist man bei der Angabe von Graph-Produktionen nicht gezwungen, für alle möglichen Werte der derzeit nicht berücksichtigten Attribute

eine eigene Graph-Produktion anzugeben, um die Anwendbarkeit dieser Graph-Produktion zu gewährleisten. Alle diese in der Graph-Produktion nicht explizit angegebenen Attribute werden bei der Anwendung der Graph-Produktion identisch übernommen. (Dies kann analog zum Vorgehen bei der Definition der Einbettungsüberführung (vgl. /Na 79/) weiter formalisiert werden.) Anschließend werden dann nur einige Attributwerte aktualisiert; dies ist in Form einer Folge von Modula-2-Zuweisungsanweisungen zusammen mit der Graph-Produktion angegeben.

Da wir an späterer Stelle in dieser Arbeit eine derartige Graph-Grammatik nicht nur als Erzeugendensystem sondern auch als Ersetzungssystem auffassen wollen, haben wir die übliche Definition von Graph-Grammatiken etwas verallgemeinert. Insbesondere fordern wir hier für den allgemeinen Fall nicht eine disjunkte Zerlegung in terminale und nichtterminale Knotenmarkierungen und die Existenz einer nichtterminalen Knotenmarkierung auf der linken Seite einer Produktion (vgl. z.B. Def.I.2.15 in /Na 79/).

Sei ->* der übliche reflexive und transitive Abschluß von ->; jeder gakk-Graph G, der aus dem Startgraphen G0 mit Hilfe der Regeln aus AGG abgeleitet werden kann, also G0 ->* G, heißt **Graph-Satzform** bez. der attributierten Graph-Grammatik AGG.

3.2.5 Abstrakter Syntaxgraph

Die im Abschnitt 3.2.3 aus einer normierten EBNF abgeleiteten Graph-Produktionen sind Graph-Produktionen im Sinne von Def. 3.2.4.1. Zusammen mit einem Startgraphen G0, dem Struktur-Graphinkrement "ProgMod" bzw. "ImplMod", liegt somit eine Graph-Grammatik vor, durch die eine Menge von Graph-Satzformen eindeutig festgelegt ist. Im Abschnitt 3.2.3 haben wir weiterhin zwischen terminalen und nichtterminalen Knotenmarkierungen unterschieden. Die Menge der terminalen Knotenmarkierungen besteht dabei aus einer Vereinigung der Menge Atom zur Kennzeichnung atomarer Graphinkremente und der Mengen List bzw. Struct zur Kennzeichnung von Listen- bzw. Struktur-Graphinkrementen. Entsprechend gehören zur Menge der nichtterminalen Knotenmarkierungen die Mengen Alter bzw. Opt zur Kennzeichnung noch zu ersetzender Knoten bzw. die beiden Hilfsmengen *_List bzw. *_Atom. Enthält eine Graph-Satzform nun nur noch terminale Knotenmarkierungen ist, keine der Graph-Produktionen mehr anwendbar. Diese Graph-Satzformen sind somit Darstellungen der kontextfreien Struktur eines Modula-2-Moduls und werden im folgenden, in Anlehnung an den Begriff eines abstrakten Syntaxbaums, **abstrakter Syntaxgraph** genannt.

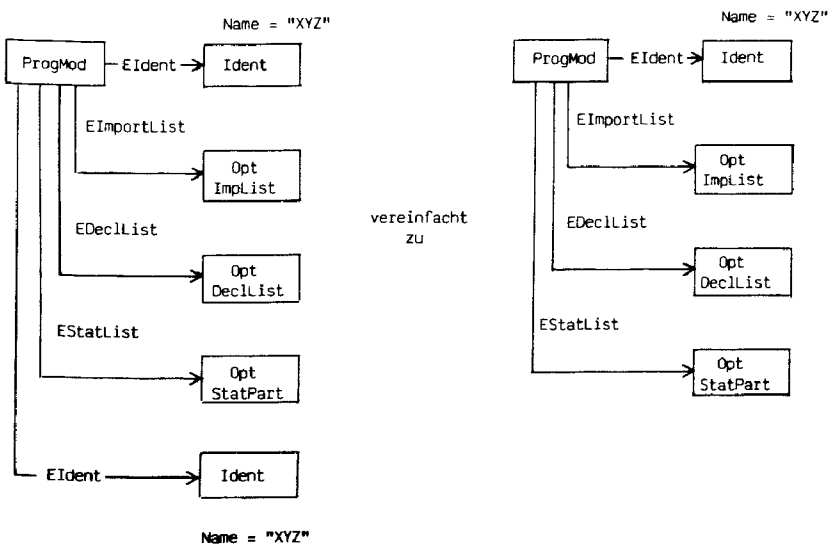
Diese abstrakten Syntaxgraphen bilden im IPSEN-Projekt im Bereich des Programmierens-im-Kleinen das Grundgerüst der Modulgraphen, also der internen

Darstellung eines Modula-2-Moduls und aller werkzeugspezifischen Informationen. Das bisher vorgestellte systematische Vorgehen bei der Entwicklung einer derartigen Graphenklasse durch Angabe einer zugehörigen Graph-Grammatik ist unabhängig von einer konkreten Programmiersprache. Das bisher am Beispiel der Programmiersprache Modula-2 erläuterte Vorgehen kann auf jede beliebige (Programmier-) Sprache übertragen werden, deren Syntax durch eine normierte EBNF beschrieben ist.

Bei einer wie bisher beschriebenen systematischen Vorgehensweise bleibt es nicht aus, daß der abstrakte Syntaxgraph als Darstellung eines Modula-2-Moduls kleinere **Ineffizienzen** und **Redundanzen** enthält. Derartige Stellen können nun per Hand modifiziert werden, um eine kompaktere Darstellung der kontextfreien Struktur zu erhalten. Hierbei kann insbesondere an die beiden folgenden Möglichkeiten gedacht werden:

a) Falls in einem expandierten Struktur-Graphinkrement zum Wurzelknoten zwei atomare Graphinkremente als Söhne mit identischer Markierung und Attributierung existieren, kann einer der Söhne gestrichen werden. (Im folgenden verzichten wir bei der Darstellung von Modulgraphen auf eine explizite Angabe der Knotenbezeichnung.)

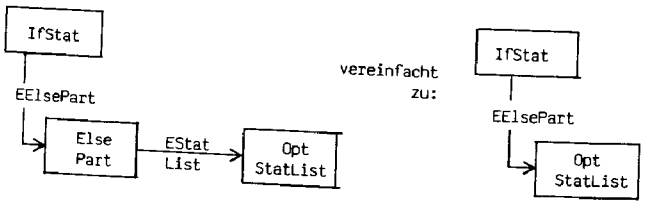
Beispiel:



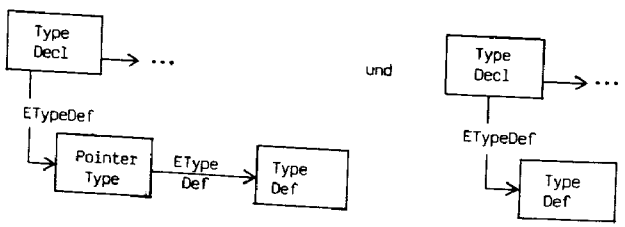
b) Falls in einem Struktur-Graphinkrement (z.B. ElsePart) zum Wurzelknoten nur ein einziger Sohn existiert, können diese beiden Knoten verschmolzen werden, wenn

durch die Markierung der einlaufenden Kante das Struktur-Graphinkrement bereits eindeutig charakterisiert ist.

Beispiel:



Eine derartige kompaktere Darstellung ist jedoch nicht immer möglich. Bei dem Struktur-Graphinkrement PointerType würde dies z.B. bedeuten, daß dieses Struktur-Graphinkrement nicht mehr identifizierbar wäre. Man betrachte hierzu die beiden folgenden Graphausschnitte:



3.2.6 Kontextsensitive Beziehungen

Durch die aus einer normierten EBNF abgeleiteten abstrakten Syntaxgraphen wird die kontextfreie Struktur eines Programms in einer Programmiersprache dargestellt. Neben der formalen Festlegung der kontextfreien Syntax, z.B. durch ein EBNF-System, gehören eine Menge kontextsensitiver Regeln zur Festlegung einer Programmiersprache, die bei jedem syntaktisch korrekten Programm dieser Programmiersprache erfüllt sein müssen.

In den in der Literatur beschriebenen Software-Entwicklungsumgebungen werden verschiedene Möglichkeiten beschrieben, derartige kontextsensitive Regeln beim aktuell erstellten Modul zu überprüfen. Die ersten in der Forschung entwickelten (z.B. Mentor /DH 75/) und die heute industriell vertriebenen Software-Entwicklungsumgebungen (z.B. /IT 85/) bestehen im wesentlichen nur aus einem syntaxgestützten Editor, der darauf beschränkt ist, während der Programmerstellung bereits die kontextfreie Korrektheit zu gewährleisten. Nach Fertigstellung der Eingabe wird das Programm dann einem Compiler übergeben, der wie üblich auch alle

kontextsensitiven Beziehungen überprüft. Da bei diesen Editoren nur die kontextfreie Struktur des aktuell bearbeiteten Programms bekannt sein muß, reichen baumartige Datenstrukturen, wie z.B. abstrakte Syntaxbäume, zur internen Darstellung aus (z.B. /DG 80/).

Mittlerweile haben die meisten Forschungsprojekte mit dem Thema Software-Entwicklungsumgebung das Ziel, bereits während der Programmerstellungszeit neben der kontextfreien Syntax auch alle kontextsensitiven Regeln zu überprüfen. Während die meisten dieser Projekte eine baumartige Datenstruktur zur Darstellung der kontextfreien Struktur des aktuell bearbeiteten Programms verwenden, existieren sehr unterschiedliche Ansätze, kontextsensitive Beziehungen in einem Programm zu beschreiben bzw. überprüfen. In dem von Reps/Teitelbaum entwickelten 'Synthesizer Generator' (/TR 81/, /RT 83/) werden attributierte Grammatiken eingesetzt. Insbesondere wurden Überlegungen angestellt, wie zugehörige Attribute bei einer Modifikation des Systems mit minimalem Aufwand aktualisiert werden können (/Re 82a/, /Re 82b/, /JF 82/).
Andere Projekte benutzen zur Beschreibung kontextsensitiver Beziehungen weniger formale Methoden. So werden z.B. im Projekt Gandalf (/Ha 82/) zur Überprüfung kontextsensitiver Zusammenhänge sogenannte 'action routines' benutzt. Eine ähnliche, ebenso prozedurale Vorgehensweise findet man auch im Projekt Pecan (/Re 84/).
Auch im Projekt Mentor existieren mittlerweile Überlegungen, kontextsensitive Beziehungen durch Benutzung der Sprache "Typol" durch bedingte Ersetzungsregeln zu beschreiben (/DK 84/).

Alle diese Vorgehensweisen haben jedoch den Nachteil, daß zur Darstellung der kontextfreien und kontextsensitiven Beziehungen **unterschiedliche Beschreibungsmittel** benutzt werden. Dies wird im Projekt IPSEN durch die Benutzung einer einheitlichen, graphartigen Datenstruktur vermieden. Wir werden dies im folgenden verdeutlichen, indem wir zunächst erläutern, wie der abstrakte Syntaxgraph erweitert wird, um auch kontextsensitive Zusammenhänge darzustellen. Anschließend werden wir erläutern, wie dementsprechend auch das Erzeugendensystem zu erweitern ist.

Kontextsensitive Regeln erzwingen häufig einen Vergleich textuell bzw. im abstrakten Syntaxgraphen weit entfernt liegender Stellen. Als Beispiel sei hier die Existenz einer Deklaration zu einer im Anweisungsteil eines Programms benutzten Variablen genannt. In der bisher eingeführten internen Darstellung eines Programms, dem abstrakten Syntaxgraphen, ist deshalb jede Überprüfung einer kontextsensitiven Regel durch einen mehr oder weniger aufwendigen Suchalgorithmus im abstrakten Syntaxgraphen zu realisieren.
Aufgrund unserer Benutzung eines allgemeinen Graphen als interne Datenstruktur ist

es nun naheliegend, den bei der Überprüfung einer kontextsensitiven Regel ermittelten kontextsensitiven Zusammenhang zwischen zwei Graphinkrementen für spätere Überprüfungen kontextsensitiver Regeln aufzuheben, d.h. durch zusätzliche Kanten im abstrakten Syntaxgraphen abzulegen. Charakteristisches Merkmal dieser **inkrementellen Vorgehensweise** ist also die Abspeicherung aller zu einem Inkrement ermittelten Informationen in der internen Darstellung dieses Inkrements, um eine erneute Bearbeitung dieses Inkrements durch eine Benutzung der abgespeicherten Informationen effizienter realisieren zu können. Wir kommen auf diesen Punkt ausführlich im Kapitel 4 zurück.

Während die kontextfreie Syntax einer Programmiersprache in der Regel durch eine EBNF oder eine Menge von Syntaxdiagrammen dargestellt wird, werden die zugehörigen kontextsensitiven Regeln üblicherweise weniger präzise, d.h. umgangssprachlich formuliert (z.B. /Wi 82/). Aus diesem Grunde ist es an dieser Stelle nicht möglich, analog zur normierten EBNF im Falle der kontextfreien Syntax hier eine ähnlich fundierte Ausgangsbasis einzuführen. Die im folgenden beschriebenen zusätzlichen Kanten im Modulgraphen zur Darstellung kontextsensitiver Zusammenhänge haben sich insbesondere bei der Entwicklung des syntaxgestützten Editors als nützlich erwiesen, um bei Änderungen eines Inkrements effizient zugehörige kontextsensitive Zusammenhänge überprüfen zu können (vgl. /Sc 86/). Eine weitere Motivation für diese Kanten wird im Verlauf dieser Arbeit bei der Vorstellung der Realisierung der einzelnen Werkzeuge gegeben.

Diese zusätzlichen Kanten beschreiben vor allem den Zusammenhang zwischen Objekten im Anweisungsteil und zugehörigen Deklarationen im Deklarationsteil bzw. kontextsensitive Zusammenhänge innerhalb des Deklarationsteils. Es sind im einzelnen folgende Kanten:

- **EUseDec** - Kante: Zwischen jedem (benutzenden) Auftreten eines deklarierten Bezeichners im Anweisungsteil, d.h. Konstanten-, Datenobjekt-, Prozedurbezeichner bzw. Aufzählungs- oder Verbundkomponentenbezeichner, und dem Knoten mit demselben Bezeichner als aktuelle Attributierung im entsprechenden Deklarationsteil wird eine mit EUseDec markierte Kante gezogen.
- **ESetDec** - Kante: Zwischen dem setzenden Auftreten eines Datenobjektbezeichners und dem Knoten mit demselben Bezeichner als aktuelle Attributierung in einem entsprechenden Deklarationsteil wird eine ESetDec-Kante gezogen. (Unter einem setzenden Auftreten wird das Auftreten auf der linken Seite einer Wertzuweisungsanweisung oder als call-by-reference-Parameter verstanden; ansonsten wird von einem benutzenden Auftreten eines Datenobjekts gesprochen.)

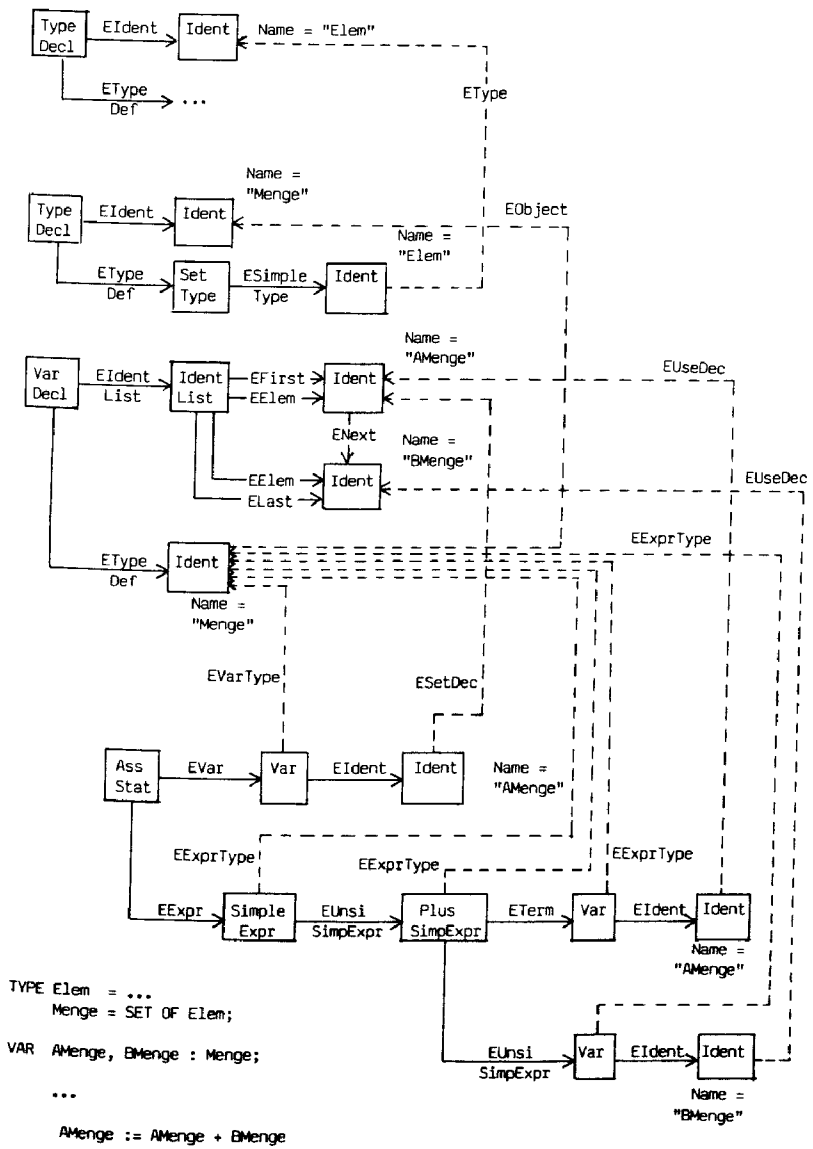


Abb. 3.2.6.1: Abstrakter Syntaxgraph mit zusätzlichen kontextsensitiven Kanten

- **EVarType** - Kante: Variablenbezeichner, die Komponenten komplexerer Datenstrukturen bezeichnen, werden aus mehreren Bezeichnern zusammengesetzt z.B. ActStack.CardStack[1]). Von jeder setzenden Variablen wird eine EVarType-Kante zur entsprechenden Typdefinition gezogen.

- **EExprType** - Kante: von jedem (Teil-)Ausdruck wird eine EExprType-Kante zur entsprechenden Typdefinition gezogen.

- **EObject** - Kante: Von dem Typbezeichner in einer Datenobjektdeklaration zum Typbezeichner der dazugehörenden Typdeklaration wird eine EObject-Kante gezogen.

- **EType** - Kante: Von einem Typbezeichner in einer Typdefinition zum Typbezeichner der dazugehörenden Typdeklaration wird eine EType-Kante gezogen.

Zur Verdeutlichung dieser Kanten geben wir in Abb. 3.2.6.1 einen Ausschnitt eines Modula-2-Moduls und die zugehörige Graphdarstellung an.

Damit diese Kanten zur Darstellung kontextsensitiver Zusammenhänge auch bei der Benutzung von Standarddatentypen und -prozeduren sowie vordefinierten Konstanten korrekt eingetragen werden können, müssen diese Objekte, analog zu den vom Benutzer selbstdefinierten, im Modulgraphen eingetragen werden. Zu diesem Zwecke betten wir den bisherigen Startgraphen, das Struktur-Graphinkrement ProgMod bzw. ImplMod, in einen Graphen ein, der aus einem mit "System" markierten Wurzelknoten besteht, an dem als Sohn eine Liste der Deklarationen aller Standarddatentypen, -prozeduren sowie vordefinierten Konstanten hängt und als zweiter Sohn das Strukturgraphinkrement ProgMod bzw. ImplMod (vgl. Abb. 3.2.6.2). Zur Kennzeichnung, daß es sich bei diesen Deklarationen um vordefinierte Größen handelt, wird der definierende Teil durch einen Knoten mit der Markierung "Standard" dargestellt. Das heißt, daß bei vordefinierten Konstanten im Gegensatz zu den vom Benutzer deklarierten Konstanten der Knoten mit dem konstanten Ausdruck durch einen Knoten mit der Markierung "Standard" ersetzt wird.
Bei vordefinierten Typen (INTEGER, CARDINAL, REAL, BOOLEAN, CHAR, BITSET) steht anstelle der Typdefinition ein mit "Standard" markierter Knoten.
Bei Standardprozeduren wird der Prozedurrumpf durch einen mit "Standard" markierten Knoten ersetzt. Die innerhalb der Identifierliste der formalen Parameter vorkommenden mit "Ident" markierten Knoten erhalten ein leeres Name-Attribut, da bei einem Prozeduraufruf nur der Typ eines formalen Parameters und nicht sein Name für eine kontextsensitive Überprüfung relevant ist.
Abb. 3.2.6.2 zeigt einen Ausschnitt aus einem derartig erweiterten Startgraphen.

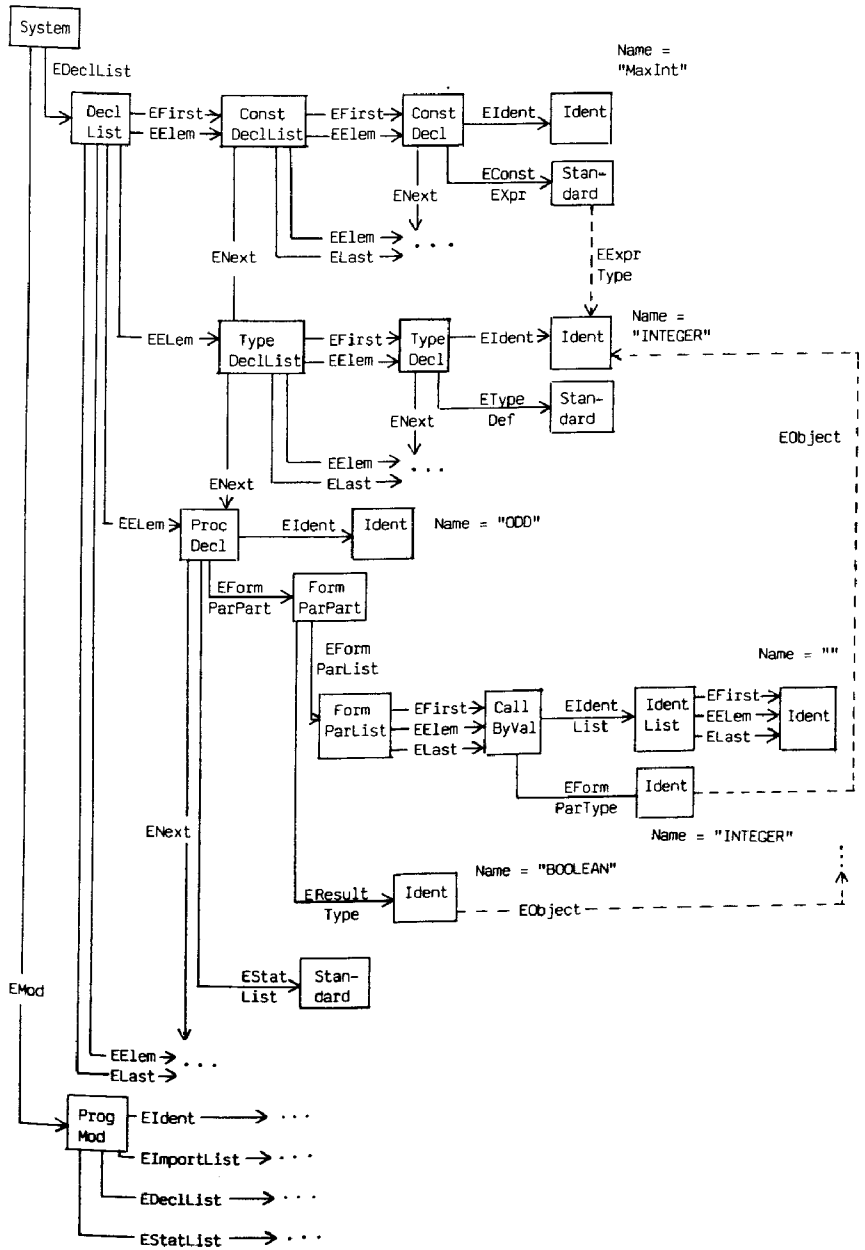


Abb. 3.2.6.2: Ausschnitt eines erweiterten Startgraphen

3.2.7 Erweiterung des Erzeugendensystems

Das bisher vorgestellte Erzeugendensystem für die Klasse der abstrakten Syntaxgraphen ist nun geeignet zu erweitern, damit beim Aufbau eines abstrakten Syntaxgraphen auch diese Kanten zur Darstellung kontextsensitiver Beziehungen in den Graphen eingetragen werden. Wir werden auf diese Weise ein Erzeugendensytem für eine Klasse von Graphen erhalten, die kontextfrei und kontextsensitiv korrekte Modula-2-Moduln darstellen. In diesem Sinne stellt das Erzeugendensystem, also eine Graph-Grammatik, eine formale Beschreibung der kontextfreien und kontextsensitiven Syntax der Programmiersprache Modula-2 dar.

Kontextsensitive Regeln sichern in erster Linie eine korrekte Benutzung von Bezeichnern und Literalen. Dies bedeutet, daß bei Eingabe eines Bezeichners oder Literals im Programmtext überprüft werden muß, ob eine **kontextsensitive Regel** verletzt wurde. Für das oben angegebene Erzeugendensystem für die Klasse der abstrakten Syntaxgraphen bedeutet dies entsprechend, daß beim Eintragen eines atomaren Graphinkrements mit einem Bezeichner oder Literal als aktuelle Attributierung im restlichen, bereits erzeugten Graphen überprüft werden muß, ob dieses atomare Graphinkrement mit dieser Attributierung eingetragen werden darf. Im positiven Fall sind dann Kanten zur Darstellung kontextsensitiver Zusammenhänge einzutragen.

Bei dem bisher vorgestellten Graph-Grammatik-Kalkül ist zu einem bestimmten Zeitpunkt jede Graph-Produktion anwendbar, zu deren linker Seite ein struktur-äquivalenter Untergraph im aktuellen Graph existiert. Um jedoch den oben angedeuteten, zur Überprüfung einer kontextsensitiven Rege nötigen Graph-Algorithmus und das anschließende Eintragen zusätzlicher Kanten mit demselben Formalismus beschreiben zu können, muß es möglich sein, die Anwendung einer bestimmten Folge von Graph-Produktionen festlegen zu können. Hierzu bieten sich zwei unterschiedliche Möglichkeiten an: Bei dem bisher eingeführten Graph-Grammatik-Kalkül ist es denkbar, durch Einführung zusätzlicher Hilfsknoten und Markierungen dafür zu sorgen, daß zu einem bestimmten Zeitpunkt nur eine einzige Produktion anwendbar ist. Eine derartige Verkapselung eines algorithmischen Anteils in der Datenstruktur macht die Graph-Grammatik jedoch sehr schwer lesbar, da strukturverändernde Anteile der Graph-Produktionen mit Hilfsknoten und -kanten für algorithmische Anteile vermischt werden.

Eine zweite, im folgenden gewählte Vorgehensweise besteht darin, den bisherigen Graph-Grammatik-Kalkül um **Kontrollprozeduren** zu erweitern. Diese Kontrollprozeduren haben den syntaktischen Aufbau üblicher **Modula-2-Prozeduren,** in deren Rumpf zusätzlich (parametrisierte) Graph-Produktionen oder weitere Kontrollprozeduren aufgerufen werden können und Untergraphentests als boolesche Ausdrücke benutzt werden. Hierdurch besteht die Möglichkeit, den algorithmischen

Anteil von dem strukturverändernden Anteil in der Graph-Grammatik zu trennen. (Eine ausführliche Diskussion der verschiedenen Möglichkeiten, bei Graph-Grammatiken bestimmte Folgen von Produktionsanwendungen vorzuschreiben, findet sich in Bemerkung III.1.6, S. 195 in /Na 79/). Im Gegensatz zu den in /Na 79/ eingeführten Kontrolldiagrammen benutzen wir Kontrollprozeduren, da durch die Benutzung der uns dadurch zur Verfügung stehenden Kontrollstrukturen und Datenstrukturen die Graph-Grammatik insbesondere leichter lesbar und verständlicher wird. Ein weiterer denkbarer Ansatz zur Formulierung derartiger Graphdurchläufe ist die Benutzung von "Pfadausdrücken", in denen festgelegt wird, über welche Kantenfolgen der Graph traversiert werden soll. Dieser Aspekt wird in der Diplomarbeit (/Qu 86/) untersucht, die aktuell bearbeitet wird.

Wir verdeutlichen diese Benutzung von Kontrollprozeduren an Hand des folgenden Beispiels:
Da Bezeichner an sehr vielen, verschiedenen Stellen im Programmtext stehen können, muß beim Eintrag eines Bezeichners zunächst untersucht werden, in welchem Kontext der Bezeichner auftritt, um die passende kontextsensitive Regel zu überprüfen. Eine mögliche Alternative ist etwa, daß der Bezeichner ein Datenobjektbezeichner innerhalb einer Variablen auf der linken Seite einer Wertzuweisungsanweisung ist (z.B. der Bezeichner "ActStack" in der Variablen "ActStack.CardStack[1]"). In diesem Fall ist zu überprüfen, ob eine entsprechende Deklaration vorliegt. Diese Überprüfung verläuft nun im einzelnen wie folgt:

Schritt 1: Der aktuelle Bezeichner wird eingetragen und der momentan bearbeitete Knoten durch zwei zusätzliche Markierungsknoten gekennzeichnet:

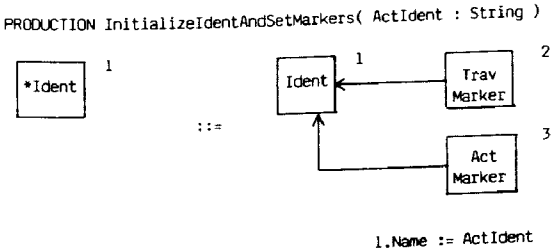


Abb. 3.2.7.1

Schritt 2: Der für die Traversierung eingetragene Knoten TravMarker wird nun solange auf den Vaterknoten gesetzt bis die Wurzel des Strukturgraphinkrements assignment_statement (AssStat) erreicht wird:

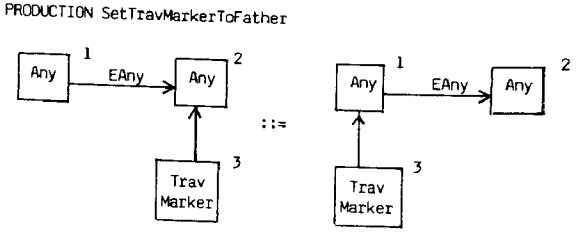


Abb. 3.2.7.2

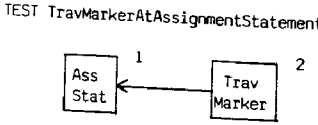


Abb. 3.2.7.3

Bei derartigen Produktionen zum Verschieben eines Markierungsknotens ist häufig in den Graph-Produktionen die konkrete Knoten- bzw. Kantenmarkierung uninteressant. Da Knoten bzw. Kanten je nach Kontext sehr unterschiedlich markiert sein können, müßten eine Menge von Graph-Produktionen angegeben werden, um jede dieser möglichen Situationen zu beschreiben. Aus diesem Grunde sehen wir in diesem Fall vor, daß anstelle eine konkreten Markierung die Markierung "Any" bzw. "EAny" benutzt werden darf. Vor Anwendung dieser Graph-Produktion müssen dann im Stil einer Makroexpansion die auf den beiden Seiten der Produktion auftretenden Any bzw. EAny Markierungen konsistent ersetzt werden. Dies heißt etwas formaler, daß wir **zweistufige Graph-Produktionen** in unserem Graph-Grammatik-Kalkül zulassen.

Schritt 3: Nun muß zunächst die kleinste Prozedurdeklaration bzw. der Modulanfang gefunden werden, um im zugehörigen Deklarationsteil zu überprüfen, ob der untersuchte Bezeichner dort deklariert ist. Hierzu muß der Zeiger TravMarker u.U. über eine Folge von ineinandergeschachtelten Anweisungen nach oben geschoben werden. Hierzu wird er wiederholt jeweils auf den Großvater des aktuellen Knotens gesetzt.

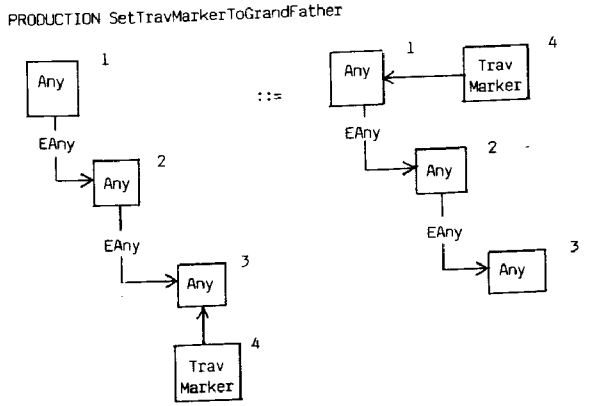


Abb. 3.2.7.4

Schritt 4: Nun sind nacheinander die umgebenden Deklarationsteile zu durchsuchen, ob eine Variablendeklaration mit demselben Bezeichner existiert.

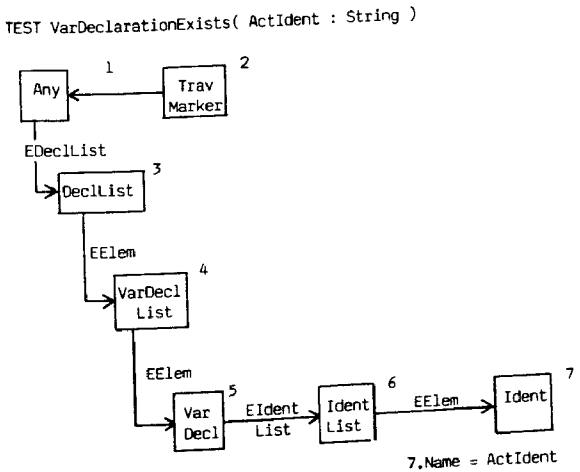


Abb. 3.2.7.5

Bei der Überprüfung eines derartigen Tests wird untersucht, ob zu dem angegebenen Graphen ein hinreichend äquivalenter Untergraph im aktuellen Graphen enthalten ist. Das bedeutet, daß neben der Struktur auch die aktuelle Attributierung des Attributs Name des dem Knoten 7 entsprechenden Knotens mit dem angegebenen Graphen übereinstimmen muß.

Schritt 5: Falls eine Deklaration gefunden wird, können ESetDec- und EVarType-

Kante gezogen werden.

PRODUCTION InsertESetDecAndEVarTypeEdge (ActIdent : String);

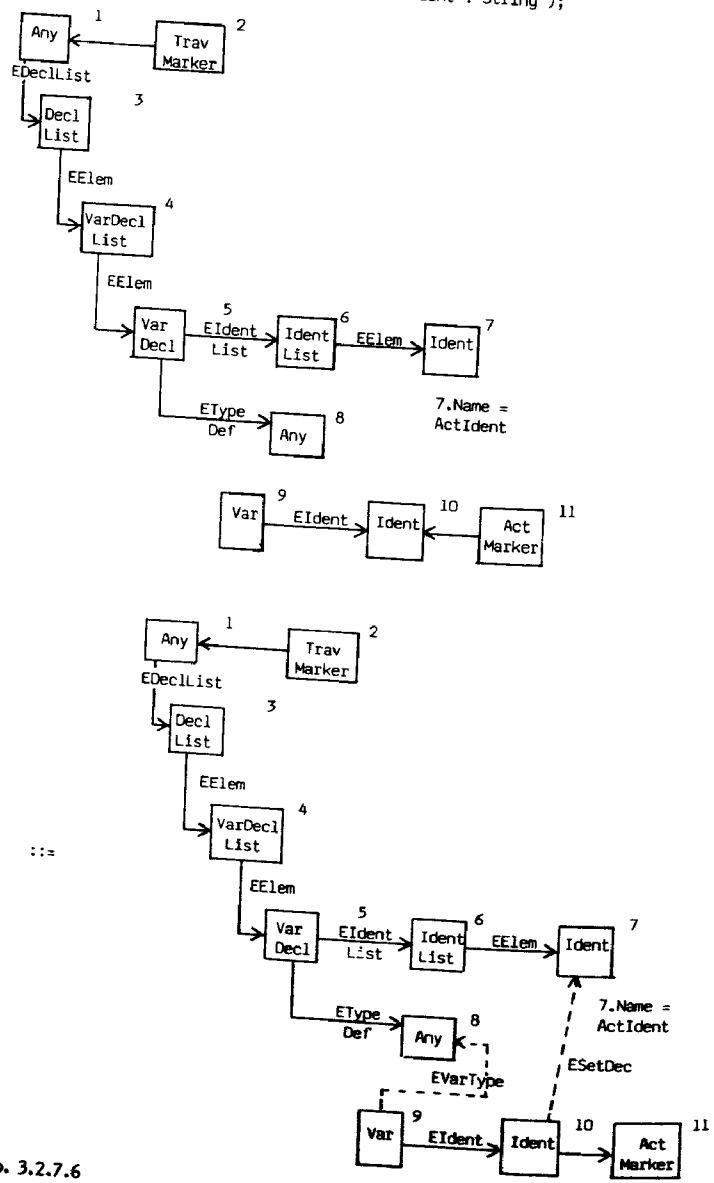


Abb. 3.2.7.6

Schritt 6: Falls der eingetragene Bezeichner Bestandteil eines zusammengesetzten Variablenbezeichners ist, muß die EVarType-Kante zu der Typdefinition gezogen werden, die den Typ der gesamten Variablen und nicht nur des deklarierten Datenobjektbezeichners angibt. Dieses Umhängen der EVar-Type-Kante geschieht in der Kontrollprozedur "ActualizeEVarTypeEdge". Wird dabei ein Verstoß gegen eine kontextsensitive Regel festgestellt, werden alle bei der Ausführung dieser Kontrollprozedur gemachten Veränderungen im Graph wieder rückgängig gemacht. Insbesondere wird dabei die eingetragene aktuelle Attributierung des Ident-Knotens wieder rückgängig gemacht, alle zusätzlich gezogenen Kanten und die beiden Markierungsknoten werden gelöscht. Diese restriktive Behandlung von Inkonsistenzen gewährleistet, daß nach Beendigung einer Kontrollprozedur stets eine Darstellung eines kontextfrei und kontextsensitiv korrekten Programmausschnitts vorliegt.

Schritt 7: Falls keine entsprechende Deklaration gefunden wurde, wird ebenfalls die eingetragene aktuelle Attributierung gelöscht und die beiden zusätzlichen Markierungsknoten werden wieder entfernt.

PRODUCTION DeleteIdent

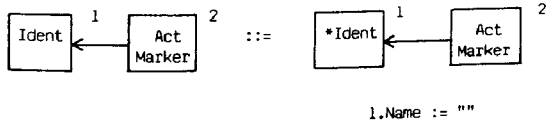


Abb. 3.2.7.7

Die gesamte Kontrollprozedur zum Eintragen eines Bezeichners in der Variablen einer Wertzuweisungsanweisung hat damit die folgende Gestalt:

```
PROCEDURE InsertIdentInVariableInAssignmentStatement( ActIdent : String );

VAR Ready : BOOLEAN;

BEGIN
   (* Schritt 1 *)
   Production( InitializeIdentAndSetMarkers( ActIdent ));

   (* Schritt 2 *)
   REPEAT
      Production( SetTravMarkerToFather )
   UNTIL  Test( TravMarkerAtAssignmentStatement );

   (* Schritt 3 *)
   REPEAT
      Production( SetTravMarkerToGrandFather)
   UNTIL  Test( TravMarkerAtProcedureDeclaration ) OR
          Test( TravMarkerAtProgramModule ) OR
          Test( TravMarkerAtImplementationModule );
```

```
(* Schritt 4 *)
Ready := FALSE;
WHILE NOT Ready DO
   IF  Test( VarDeclarationExists ) THEN

       (* Schritt 5 *)
       Production( InsertESetDecAndEVarTypeEdge );

       (* Schritt 6 *)
       ActualizeEVarTypeEdge;

   ELSIF NOT ( Test( TravMarkerAtModuleHead ) ) THEN
      Production( SetTravMarkerToGrandFather )

   ELSE
      Ready := TRUE
   END;
END;

(* Schritt 7 *)
Production( DeleteIdent )

Production( DeleteMarkers )

END InsertIdentInVariableInAssignmentStatement;
```

Abb. 3.2.7.8: Beispiel für eine Kontrollprozedur

Eine oder mehrere der oben eingeführten Kanten zur Darstellung kontextsensitiver Zusammenhänge ist immer dann im bereits erzeugten Graphen einzutragen bzw. zu aktualisieren, wenn ein konkreter Bezeichner als neue aktuelle Attributierung in den Graphen eingetragen werden soll. Hierbei kann es durchaus vorkommen, daß durch den Eintrag eines solchen Bezeichners eine ganze Reihe derartiger kontextsensitiver Kanten zu aktualisieren sind. Als Beispiel sei hier die Eingabe eines Bezeichners in einem komplexen Ausdruck genannt. Nach der Eingabe müssen u.U. bei einer Reihe von umgebenden Teilausdrücken die zugehörigen EExprType-Kanten modifiziert werden. Wir geben in dieser Arbeit nur das obige kleine Beispiel für eine derartige Kontrollprozedur an. Ausführlicher wird dieser Themenkreis in /Sc 86/ diskutiert. Dort wird auch erläutert, wie diese **Kontrollprozeduren systematisch entwickelt** werden können, in dem ein analoger Aufbau aller Kontrollprozeduren vorgestellt wird. An dieser Stelle sei nur bemerkt, daß alle zur Programmiersprache Modula-2 gehörenden kontextsensitiven Regeln analog zu obiger Vorgehensweise überprüft werden können, indem die zu atomaren Graphinkrementen gehörenden Graph-Produktionen durch Kontrollprozeduren ersetzt werden. Wie bereits oben gesagt, ist durch die restriktive Handhabung auftretender Inkonsistenzen gewährleistet, daß nach Ausführung einer Kontrollprozedur oder einer sonstigen Graph-Produktion der aktuelle Graph eine Darstellung eines kontextfrei und kontextsensitiv korrekten Modula-2-Programmausschnitts ist. Falls keine Kontrollprozedur oder Graph-Produktion mehr anwendbar ist, ist der aktuelle Graph eine Darstellung eines korrekten Modula-2-Programms.

Durch die Erweiterung des in den Abschnitten 3.2.3 bis 3.2.5 vorgestellten Erzeugendensystems um Kontrollprozeduren liegt somit in Form einer programmierten, attributierten Graph-Grammatik eine **formale Beschreibung der kontextfreien und kontextsensitiven Syntax** der Programmiersprache Modula-2 vor. Es ist Ziel des nächsten Abschnitts, diese Erweiterung des Graph-Grammatik-Kalküls etwas zu präzisieren.

3.2.8 Programmierte Graph-Grammatiken

Wir haben oben an einem Beispiel erläutert, wie mit Hilfe von Kontrollprozeduren eine bestimmte Reihenfolge der Anwendung von Graph-Produktionen festgelegt werden konnte. Diese Kontrollprozeduren haben die Gestalt üblicher Modula-2-Prozeduren, in deren Rumpf weitere Kontrollprozeduren aufgerufen werden können, Graph-Produktionen angewandt werden können oder Untergraphentests als boolesche Ausdrücke z.B. in bedingten Anweisungen stehen können. Bei der Menge der Kontrollprozeduren und Graph-Produktionen muß zwischen zwei Arten unterschieden werden. Einerseits existieren Kontrollprozeduren bzw. einzelne Graph-Produktionen, die einen konsistenten Graphen in einen veränderten konsistenten Graphen überführen. Andererseits existieren Hilfskontrollprozeduren und Hilfsproduktionen, die während eines solchen Überführungsschritts ausgeführt bzw. angewendet werden. Bei einer Präzisierung des Ableitungsbegriffs muß nun darauf geachtet werden, daß die innerhalb einer Kontrollprozedur aufgerufenen Hilfskontrollprozeduren und Hilfsproduktionen nicht auch unabhängig von der geforderten Reihenfolge angewandt werden. Aus diesem Grunde wird unterschieden zwischen **allgemein bekannten** (public) und **verborgenen** (hidden) Graph-Produktionen und Kontrollprozeduren.

Def. 3.2.8.1: (programmierte, attributierte Graph-Grammatik)
Eine **programmierte, attributierte Graph-Grammatik** ist ein 9-Tupel
PAGG = (Nodelabels, Edgelabels, Attributes, nodeatt, P, CP, T, G0, ->) mit
- Nodelabels, Edgelabels endliche Mengen von Knoten- bzw. Kantenmarkierungen,
- Attributes endliche Menge von Attributen,
- nodeatt: Nodelabels ->$\mathcal{P}$(Attributes) Attributzuordnungsfunktion,
- P = PP $\cup$ HP disjunkte Vereinigung von endlich vielen allgemein bekannten (parametrisierten) Graph-Produktionen PP (public productions) und endlich vielen verborgenen (parametrisierten) Graph-Produktionen HP (hidden productions),
- CP = PCP $\cup$ HCP disjunkte Vereinigung von endlich vielen allgemein bekannten Kontrollprozeduren PCP (public control procedures) und endlich vielen verborgenen Kontrollprozeduren HCP (hidden control procedures),
- T eine endliche Menge von Untergraphentests,

- G0 aus G (Nodelabels, Edgelabels, Attributes, nodeatt), der Startgraph
- -> der Ableitungsbegriff aus Def. 3.2.8.2.

Eine Graph-Produktion ist anwendbar, wenn zur linken Seite ein hinreichend äquivalenter Untergraph im aktuell bearbeiteten Graphen existiert. Analog hierzu ergänzen wir jede allgemein bekannte Kontrollprozedur PCP um eine sogenannte **Vorbedingung**. Diese Vorbedingung ist ein gakk-Graph zusammen mit einer Menge aktueller Attributwerte, zu dem wie bei Graph-Produktionen ein hinreichend äquivalenter Untergraph im Graphen existieren muß, damit diese Kontrollprozedur gestartet werden kann.

Def. 3.2.8.2: (Ableitungsbegriff)

Ein gakk-Graph G' wird aus einem gakk-Graph G mittels einer allgemein bekannten Graph-Produktion pp oder einer allgemein bekannten Kontrollprozedur pcp **abgeleitet** (Schreibweise: G -> G'), indem

- G' aus G mittels der Produktion pp gemäß Def. 3.2.4.2 abgeleitet wird oder
- die Vorbedingung von pcp in G erfüllt ist, pcp im üblichen Modula-2-Sinn ausgeführt wird, die aufgerufenen Graph-Produktionen gemäß Def.3.2.4.2 angewandt werden, die Ausführung terminiert und G' resultiert.

Durch diese Definition des Ableitungsbegriffs ist sichergestellt, daß verborgene Graph-Produktionen oder Kontrollprozeduren nur innerhalb allgemein bekannter Kontrollprozeduren aufgerufen werden.

Sei ->* der übliche reflexive und transitive Abschluß von ->; jeder gakk-Graph G, der aus dem Startgraphen G0 abgeleitet werden kann, also G0 ->* G, heißt **Graph-Satzform bez. der programmierten, attributierten Graph-Grammatik.**

Bei dem hier vorgestellten Graph-Grammatik-Kalkül ist die Überprüfung der Anwendbarkeit einer Graph-Produktion von der eigentlichen Anwendung losgelöst. Das bedeutet, daß während der Ausführung einer Kontrollprozedur die Anwendbarkeit jeder aufgerufenen Graph-Produktion gesichert sein muß. Der Aufruf einer Graph-Produktion, zu der im aktuellen Graph kein hinreichend äquivalenter Untergraph existiert, führt zu einem Laufzeitfehler und zum Abbruch der Ausführung der Kontrollprozedur.

Das im vorigen Abschnitt erläuterte, erweiterte Erzeugendensystem ist in diesem Sinne eine programmierte, attributierte Graph-Grammatik. Alle ableitbaren Graph-Satzformen sind Darstellungen z.T. noch unvollständiger, aber bez. der Modula-2-Syntax kontextfrei und kontextsensitiv korrekter Moduln. Diese Graph-Satzformen wollen wir deshalb im folgenden **Modulgraphen** nennen. Falls keine Graph-Produktion oder Kontrollprozedur mehr anwendbar ist, sind die Modulgraphen Darstellungen vollständiger Modula-2-Moduln.

3.3 Systematische Entwicklung eines Graph-Ersetzungssystems

Im vorhergehenden Paragraphen haben wir die systematische Entwicklung eines Erzeugendensystems für eine Klasse von Graphen, hier Modulgraphen, erläutert. Jeder Modulgraph stellt dabei eine abstrakte Repräsentation eines (Teils eines) konkreten Modula-2-Moduls dar. In diesem Paragraphen werden wir nun erläutern, inwiefern diese Modulgraphen als Grundlage für eine gemeinsame, zentrale Datenstruktur aller in IPSEN im Bereich des Programmierens-im-Kleinen vorhandenen Werkzeuge benutzt werden können. Hierzu werden wir Beispiele für werkzeugspezifische Informationen angeben, die zusätzlich im Modulgraphen abgelegt werden sollen, und auf konzeptioneller Ebene spezifizieren, wie diese im Modulgraphen abgelegten Informationen durch Werkzeugaktionen manipuliert werden können. In diesem Sinne stellt ein Modulgraph einen **abstrakten Datentyp** dar, für den auf konzeptioneller Ebene die benötigten Zugriffsoperationen zu spezifizieren sind.

Das Spektrum der in der Literatur vorzufindenden **Spezifikationsmethoden** reicht von rein informellen, in natürlicher Sprache gegebenen Spezifikationen bis hin zu sehr formalen, mathematischen Spezifikationsmethoden (vgl. z. B. /Ki 79/, /EM 85/). Zu den mehr formalen Spezifikationsmethoden gehören insbesondere sogenannte **operationelle Spezifikationen.** Diese Spezifikationen bestehen in erster Linie aus einer Menge von Ersetzungsregeln. Die Bedeutung von Zugriffsoperationen auf ein Objekt des zu spezifizierenden Datentyps ist dabei durch die Anwendung einer (Folge von) Ersetzungsregel(n) festgelegt. Ein Beispiel für solche operationellen Spezifikationen sind Term-Ersetzungssysteme, die häufig im Zusammenhang mit algebraischen Spezifikationen (vgl. /GT 78/) zitiert und behandelt worden sind.

Eine Verallgemeinerung dieser Term-Ersetzungssysteme sind Graph-Ersetzungssysteme, bei denen im Gegensatz zu den zugrundeliegenden Termen, d.h. Bäumen, nun **Graphen** die zu beschreibenden Datenstrukturen sind. Da im Projekt IPSEN auf konzeptioneller Ebene Graphen als zentrale Datenstrukturen gewählt wurden, liegt es nahe, Zugriffsoperationen auf diesen Graphen durch Graph-Ersetzungssysteme zu spezifizieren. Rein formal sind Graph-Ersetzungssysteme identisch zu den im letzten Kapitel eingeführten Graph-Grammatiken. Da der Grammatik-Begriff jedoch im allgemeinen nur im Zusammenhang mit einem Erzeugendensystem benutzt wird, werden wir diese Graph-Grammatiken im folgenden **Graph-Ersetzungssysteme** nennen. Es ist Ziel dieses Paragraphen zu erläutern, wie Graph-Ersetzungssysteme für die Spezifikation von Werkzeugaktionen, d.h. von Zugriffsoperationen auf einen Modulgraphen im IPSEN-Projekt eingesetzt werden können.
Dabei legen wir insbesondere Wert auf die Darstellung von ingenieurmäßigen Methoden aus dem Bereich des Software Engineering, die bei der Erstellung einer derartigen Spezifikation angewendet werden sollten. Durch diese unter dem Begriff **Graph Grammar Engineering** (vgl. /ES 85a/) zusammengefaßten Methoden wird eine

modulare Entwicklung eines Graph-Ersetzungssystems unterstützt. Dies wiederum ermöglicht eine leichte Anpassung der Spezifikation bei geänderten Anforderungen oder gewünschten Erweiterungen.

Wir beschränken uns in dieser Arbeit auf eine **Erläuterung der generellen Vorgehensweise** beim Einsatz von Graph-Ersetzungssystemen als Spezifikationsinstrument. Zur Verdeutlichung geben wir eine Reihe von Beispielen an. Eine ausführliche Beschreibung der Vorgehensweise und Erfahrungen bei der Benutzung dieser Methode zur Spezifikation des syntaxgestützten Editors befindet sich in /Sc 86/.

3.3.1 Graphinkrementmodifikationen

Aufgrund der inkrementorientierten Arbeitsweise aller IPSEN-Werkzeuge beziehen sich alle Modifikationen eines Modulgraphen auf ein bestimmtes Graphinkrement, das sogenannte **aktuelle Inkrement**. Dieses aktuelle Inkrement wird bei einer Repräsentation an der Benutzerschnittstelle besonders dargestellt (z.B. durch einen Flächencursor) und auch in dem konzeptionellen Datenmodell Modulgraph durch einen zusätzlichen, mit "Cursor" markierten Knoten und eine auslaufende Kante gekennzeichnet, die im aktuellen Inkrement endet. Während bei einem Erzeugendensystem die aktuell zu bearbeitende Stelle im Graphen nicht festgelegt ist, bedeutet die Existenz eines eindeutigen aktuellen Inkrements für die Spezifikation von Graphinkrementmodifikationen, daß der **Cursorknoten** in alle Graph-Produktionen zusätzlich mit aufgenommen wird. Dadurch wird erreicht, daß stets nur am aktuellen Inkrement eine Graph-Produktion anwendbar ist. Die Graph-Produktion zum Eintragen des Graphinkrements einer while-Anweisung hat damit z.B. die folgende Gestalt (vgl. auch Abb. 3.2.3.1):

PRODUCTION InsertWhileStatementGraphIncrement

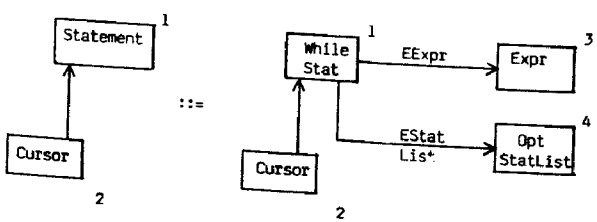


Abb. 3.3.1.1: Graph-Produktion zum Eintragen des Graphinkrements einer while-Anweisung mit Cursorknoten

Mit diesen zusätzlichen Cursorknoten entsprechen die Graph-Produktionen eines

Erzeugendensystems nun unmittelbar den Graph-Produktionen zur Spezifikation des Einfügens von Graphinkrementen in einen Modulgraphen. Neben der Spezifikation der **Einfügeoperationen** muß auch für alle anderen Aktivitäten des syntaxgestützten Editors spezifiziert werden, wie der Modulgraph bei Ausführung einer derartigen Aktivität verändert wird. Hierzu zählen z.B. Aktivitäten wie das **Löschen** von Graphinkrementen, das **Ändern von Bezeichnerattributen** oder das **Verschieben des Cursors**. Als Beispiel sei hier die folgende Graph-Produktion angegeben, die beschreibt, wie der Cursor in einer while-Anweisung auf die zugehörige Schleifenbedingung gesetzt wird.

PRODUCTION SetCursorToWhileStatementCondition

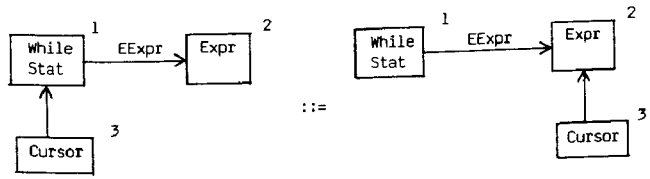


Abb. 3.3.1.2: Graph-Produktion zum Verschieben des Cursors

Alle diese Graphinkrementmodifikationen und Cursorbewegungen können mit dem im letzten Paragraphen eingeführten Graph-Grammatik-Kalkül spezifiziert werden. Hierbei finden insbesondere programmierte Graphersetzungen Anwendung, da z.B. beim Löschen eines expandierten Graphinkrements sukzessive zunächst alle innerhalb liegenden Graphinkremente gelöscht werden müssen, um eventuelle kontextsensitive Fehler zu entdecken. Auf diese Problematik und die **vollständige Spezifikation des syntaxgestützten Editors** wird ausführlich in /Sc 86/ eingegangen.

Alle Kontrollprozeduren und Graph-Produktionen zur Modifikation des Modulgraphen fassen wir in einem Graphersetzungssystem zusammen, das wir im folgenden "**GraphModifications**" nennen. Alle Kontrollprozeduren und Graph-Produktionen zum Verschieben des Cursorknotens fassen wir in dem Graph-Ersetzungssystem "**Cursor-Movements**" zusammen.

3.3.2 Werkzeugspezifische Ergänzungen

Die in einem Modulgraphen enthaltene, durch die Zusammensetzung von Graphinkrementen dargestellte kontextfreie Struktur ist notwendig, um eine inkrementorientierte Arbeitsweise des Editors und aller anderen Werkzeuge effizient realisieren zu können. Darüberhinaus haben wir erläutert, daß bei der Überprüfung kontextsensitiver Regeln ermittelte kontextsensitive Zusammenhänge durch zusätzliche Kanten im Graphen ausgedrückt werden können. Diese so dargestellten

Informationen im Modulgraphen sind in erster Linie für das Werkzeug syntaxgestützter Editor relevant. Aber auch für alle anderen Werkzeuge muß überlegt werden, welche **werkzeugspezifischen Informationen** bereits bei einer Modulgraphveränderung durch eine Aktivität des syntaxgestützten Editors ermittelt und im Modulgraphen abgelegt werden können. Wir werden im Kapitel 4 ausführlich die verschiedenen Möglichkeiten diskutieren, wie werkzeugspezifische Informationen verwaltet werden können. In diesem Abschnitt stellen wir beispielhaft einige Informationen vor, für die im IPSEN-System entschieden wurde, sie analog zu kontextsensitiven Beziehungen bereits bei einer Modulgraphveränderung im Graphen abzulegen bzw. zu aktualisieren.

Neben den bisher vorgestellten Beziehungen ist durch die gegebene Anordnung von Kontrollstrukturen auch bereits die für die Ausführung relevante Struktur im Modulgraphen verkapselt. Hierzu zählt insbesondere die Information, in welcher Reihenfolge die einzelnen Anweisungen auszuführen sind, d.h. welche möglichen **Kontrollflüsse im Anweisungsteil** verkapselt sind. Da die Ausführung im IPSEN-System durch einen Interpreter geschieht, der während der Ausführung von Anweisung zu Anweisung durch den Modulgraphen läuft, muß die nächste auszuführende Anweisung leicht bestimmbar sein. Eine explizite Darstellung des Kontrollflusses durch **zusätzliche Kanten** ist somit hier angebracht. (Auf die Erläuterung der Realisierung der Ausführung kommen wir im Kapitel 7 ausführlich zurück.)

Die in einem Modul enthaltenen Erreichbarkeitsbeziehungen bez. des Kontrollflusses zwischen Inkrementen im Anweisungsteil sind weiterhin Grundlage für die in einem Modul enthaltenen Erreichbarkeitsbeziehungen bez. des **Datenflusses.** Eine explizite Darstellung des Kontrollflusses durch zusätzliche Kanten im Anweisungsteil ist aus diesem Grunde hilfreich für eine effiziente Realisierung der im Rahmen der statischen Analyse durchzuführenden Untersuchungen des Modulgraphen. Wir kommen hierauf im Kapitel 5 bei der Erläuterung der Realsierung der statischen Analyse zurück.

Den in einem Programm enthaltenen Kontrollfluß durch eine graphartige Datenstruktur zu repräsentieren, wird seit vielen Jahren durch die Benutzung von **Flußdiagrammen** bei der Programmentwicklung praktiziert. Eine ähnliche Darstellung findet man auch in sogenannten **Flußgraphen** (vgl. /AU 79/), die als Datenstruktur für die in Compilern während der Optimierung durchgeführte Datenflußanalyse dienen. Im Unterschied zu Flußdiagrammen werden hierbei Folgen von Zuweisungsanweisungen zu sogenannten "basic blocks" zusammengefaßt, die untereinander durch den Kontrollfluß darstellende Kanten verbunden sind.

Diese Darstellungen können nun analog auf unsere Darstellung eines Programms, den Modulgraphen, übertragen werden, indem im Anweisungsteil zusätzliche, sogenannte

Kontrollflußkanten eingetragen werden. Eine Kontrollflußkante zwischen einem Inkrement I und einem Inkrement I' besagt dann, daß das Inkrement I' unmittelbar nach dem Inkrement I ausgeführt werden kann. Da die durch diese Kontrollflußkanten verkapselte Information sehr häufig benötigt wird und, wie wir sehen werden, andererseits ohne großen zusätzlichen Aufwand während des Edierens mitverwaltet werden kann, werden diese Kontrollflußkanten zusätzlich im Modulgraphen abgelegt und bei einer Veränderung des Modulgraphen sofort aktualisiert.

Zur Darstellung des Kontrollflusses unterscheiden wir vier unterschiedlich markierte Kanten:

- EControlFlow
- ETrueControlFlow
- EFalseControlFlow
- EBackControlFlow

Falls der Kontrollfluß eindeutig und unbedingt ist, wird eine mit "**EControlFlow**" markierte Kante zwischen den beiden während der Ausführung direkt nacheinander zu interpretierenden Inkrementen gezogen. Ist der Kontrollfluß abhängig von einer auszuwertenden Bedingung (z.B. bei einer if-Anweisung) wird zu dem Inkrement, das bei zutreffender Bedingung auszuführen ist (z.B. then-Teil), eine mit "**ETrueControlFlow**" markierte Kante gezogen und zu dem Inkrement, das bei nicht zutreffender Bedingung auszuführen ist (z.B. elsif- oder else-Teil), eine mit "**EFalseControlFlow**" markierte Kante gezogen. Von der letzten Anweisung innerhalb eines Schleifenrumpfes wird eine mit "**EBackControlFlow**" markierte Kante zurück zum Schleifenanfang gezogen. Hier reicht eine nur mit "**EControlFlow**" markierte Kante nicht aus, da während des Interpretierens (z.B. einer for-Schleife) erkannt werden muß, ob der Schleifenanfang zum ersten Mal oder zum wiederholten Mal erreicht wird. In Abhängigkeit hiervon muß der Schleifenzähler initialisiert bzw. erhöht oder erniedrigt werden.

Um den durch diese zusätzlichen Kanten im Modulgraphen dargestellten Kontrollfluß übersichtlicher zu machen, ergänzen wir außerdem jedes, eine Kontrollstruktur sowie eine Prozedur- bzw. Moduldeklaration darstellende Graphinkrement um einen sogenannten **Endeknoten.** In diesem Endeknoten werden die Kontrollflußkanten vor Verlassen dieses Graphinkrements zusammengeführt, so daß jedes derartige Graphinkrement (in der Regel) einen eindeutigen Anfangsknoten mit einer einlaufenden Kontrollflußkante und einen eindeutigen Endeknoten mit einer auslaufenden Kontrollflußkante besitzt. Als Beispiele geben wir schematisch die Kontrollflußkanten in der Graphdarstellung einer if- und einer while-Anweisung an:

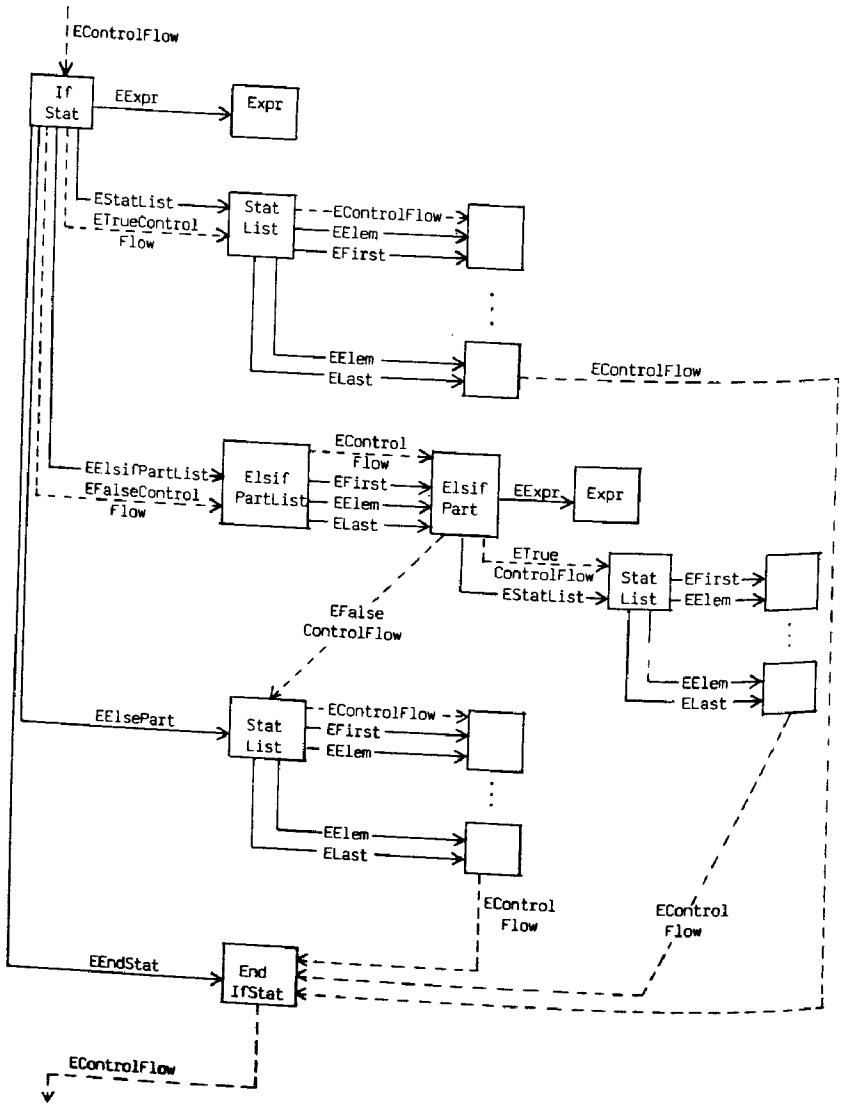


Abb. 3.3.2.1 Teil a): Kontrollflußkanten in einer if-Anweisung

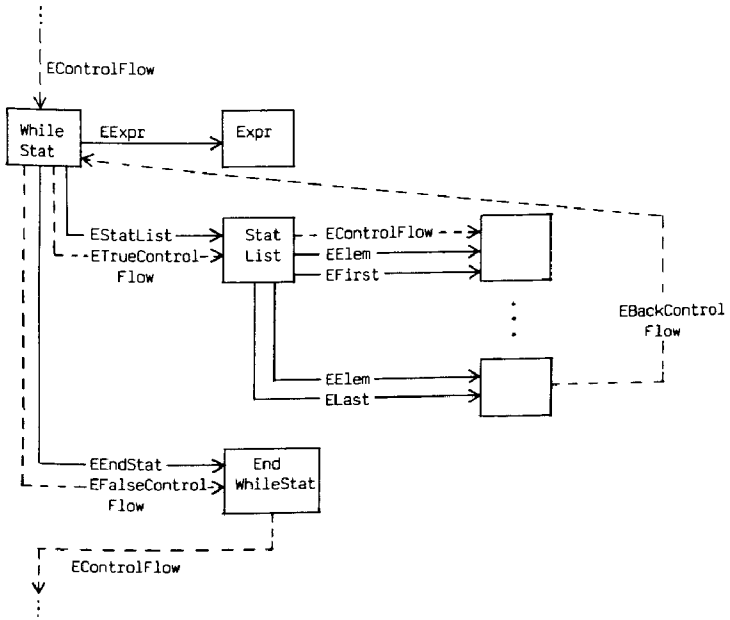


Abb. 3.3.2.1 Teil b): Kontrollflußkanten in einer while-Anweisung

Da es in Modula-2 keine beliebigen Sprunganweisungen gibt, bieten return-Anweisungen in Prozedurrümpfen bzw. exit-Anweisungen im Rumpf von loop-Schleifen die einzige Möglichkeit, eine Kontrollstruktur **vorzeitig** zu verlassen. Die von einer return-Anweisung auslaufende, mit "EControlFlow" markierte Kontrollflußkante endet deshalb in dem Endeknoten der nächsten umgebenden Prozedurdeklaration. Analog endet die von einer exit-Anweisung ausgehende Kontrollflußkante in dem Endeknoten der nächsten umgebenden loop-Anweisung.

Analog zu der im Abschnitt 3.3.1 erläuterten Spezifikation der Graphinkrementmodifikationen mit Hilfe von Graph-Ersetzungssystemen kann die zugehörige Verwaltung des Kontrollflusses durch Graph-Produktionen spezifiziert werden. Als Beispiel geben wir in der nächsten Abbildung die Graph-Produktion zum Eintrag der Kontrollflußkanten und des zusätzlichen Endeknotens in die Graphdarstellung einer while-Anweisung an.
Derartige Graph-Produktionen zur Aktualisierung des im Modulgraphen dargestellten Kontrollflusses sind für alle im Abschnitt 3.3.1 angesprochenen und ebenfalls durch Graph-Produktionen spezifizierten Graphinkrementmodifikationen anzugeben.

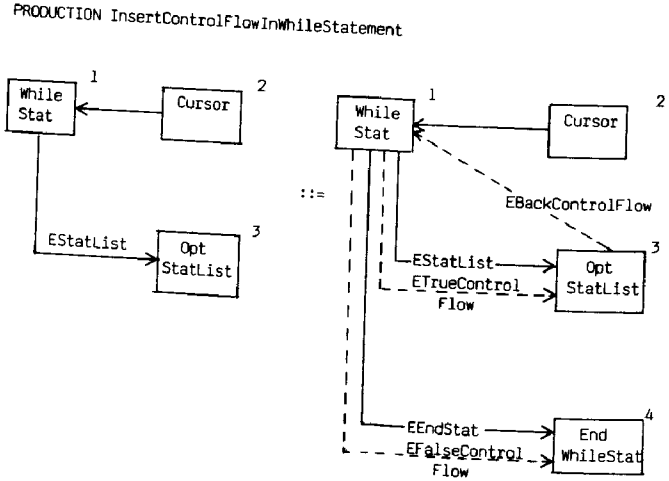


Abb. 3.3.2.2: Graph-Produktion zum Eintrag von Kontrollflußkanten

Das Verwalten der zusätzlichen Kontrollflußkanten kann in der Regel durch eine einzige Graph-Produktion spezifiziert werden. Nur beim Eintrag der auslaufenden Kontrollflußkante von einer return- bzw. exit-Anweisung muß zunächst der zugehörige Endeknoten gesucht werden. Bei dem hier durchzuführenden **Suchalgorithmus** muß analog zu dem im Abschnitt 3.2.7 vorgestellten Algorithmus zum Auffinden der Deklaration eines Datenobjekts solange in der Inkrementstruktur aufgestiegen werden, bis das gesuchte Inkrement, hier die zugehörige Prozedurdeklaration bzw. loop-Anweisung gefunden wird. Analog zu dem im Abschnitt 3.2.7 vorgestellten Vorgehen kann dies durch eine **Kontrollprozedur** spezifiziert werden. Alle diese Kontrollprozeduren und Graph-Produktionen zur Spezifikation der Verwaltung von Kontrollflußkanten werden in dem Graph-Ersetzungssystem "**ControlFlowModifications**" zusammengefaßt.

Neben diesen durch zusätzliche Knoten und Kanten ausgedrückten strukturellen Informationen werden von einigen Werkzeugen auch **nicht-strukturelle Informationen** benötigt. Hiermit sind in der Regel werkzeugspezifische Informationen gemeint, die benutzt werden, um Teile des vorliegenden Modulgraphen in eine andere Darstellung zu transformieren. Diese zusätzlichen Informationen werden in entsprechenden, an Knoten befindlichen **Attributen** abgelegt. Wir werden in den folgenden Kapiteln einige Beispiele für derartige Informationen kennenlernen, einige seien hier bereits erwähnt:

- Bei der Interpretation des Modulgraphen werden einige Teile zuvor in eine maschinennähere Form (P-Code) übersetzt. Hierzu werden Angaben über die Adressen von Datenobjekten im Laufzeitdatenbereich bzw. über den Platzbedarf

von Objekten bestimmter Datenstrukturen benötigt. Diese Angaben werden im Modulgraphen in Attributen mit Namen "Address" bzw. "Size" abgelegt. Der erzeugte maschinennähere Code wird in einem Attribut mit Namen "Code" abgelegt.
- Die abstrakte Modulgraphdarstellung muß vor der Ausgabe auf dem Bildschirm in eine Quelltextdarstellung transformiert werden. Die für diesen Unparsing-Prozeß benötigten Informationen über die Gestalt der Quelltextdarstellung wird in sogenannten "Unparsing-Schemata" abgelegt. Das zugehörige Attribut lautet "Scheme".

Einige dieser Informationen werden analog zu Kontrollflußkanten unmittelbar nach einer Graphinkrementmodifikation in den Modulgraphen eingetragen bzw. aktualisiert. Hierzu gehören z.B. die oben angesprochenen Unparsing-Schemata. Auch diese Attributmodifikation kann durch eine Graph-Produktion spezifiziert werden:

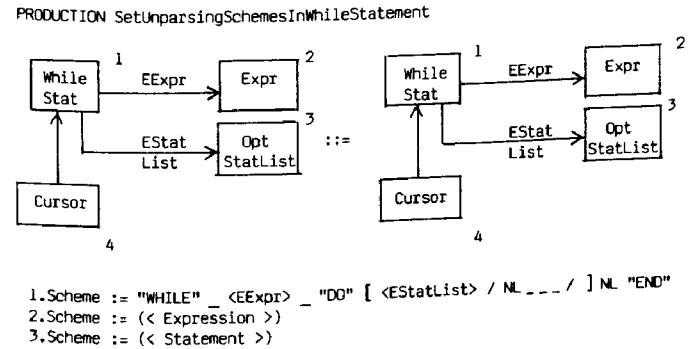


Abb. 3.3.2.3: Graph-Produktion zum Eintrag von Unparsing-Schemata

Alle Graph-Produktionen zur Modifikation des Attributs Scheme werden in dem Graph-Ersetzungssystem "**SchemeModifications**" zusammengefaßt.

3.3.3 Komposition von Graph-Ersetzungssystemen

In den letzten beiden Abschnitten haben wir beispielhaft vier verschiedene Graph-Ersetzungssysteme vorgestellt, durch die für einzelne (werkzeugspezifische) Informationen getrennt spezifiziert wird, wie diese Information bei einer Veränderung des Modulgraphen zu aktualisieren ist. Nach Fertigstellung dieser einzelnen **Teilspezifikationen** muß in einem zweiten Schritt festgelegt werden, in welcher **Reihenfolge** die einzelnen Kontrollprozeduren und Graph-Produktionen anzuwenden sind, damit unmittelbar nach einer strukturellen Veränderung des Modulgraphen alle werkzeugspezifischen Informationen geeignet aktualisiert werden. In dem bisher

erläuterten Beispiel bedeutet dies, daß festgelegt werden muß, daß unmittelbar nach dem Eintrag des Graphinkrements für eine while-Anweisung die zugehörigen Kontrollflußkanten und die zugehörigen Unparsing-Schemata zu ergänzen sind.
Für eine derartige **Komposition von Graph-Ersetzungssystemen** stellen wir im folgenden zwei Möglichkeiten vor, zunächst eine dynamische und anschließend eine statische Komposition von Graph-Ersetzungssystemen.

Bei der **dynamischen Komposition von Graph-Ersetzungssystemen** benutzen wir wiederum Kontrollprozeduren zur Spezifikation einer bestimmten Reihenfolge, in der einzelne Kontrollprozeduren aufzurufen bzw. Graph-Produktionen anzuwenden sind. Da hierbei die Komposition bei Ausführung einer Kontrollprozedur vollzogen wird, nennen wir sie "dynamische Komposition".

Das Einfügen einer while-Anweisung mit allen dazugehörenden werkzeugspezifischen Informationen wird in diesem Sinne durch die folgende Kontrollprozedur spezifiziert.

```
PROCEDURE InsertWhileStatement;

BEGIN
   Production ( InsertWhileStatementGraphIncrement );
   Production ( InsertControlFlowInWhileStatement );
   Production ( SetUnparsingSchemesInWhileStatment )
END InsertWhileStatement;
```

Abb. 3.3.3.1: Kontrollprozedur zum Einfügen einer while-Anweisung

Der große Vorteil dieser Vorgehensweise liegt darin, daß die Spezifikation aller Operationen auf dem Modulgraphen **schrittweise** und für jede werkzeugspezifische Information **unabhängig** erstellt werden kann. Bei Änderung der Anforderungen an ein Werkzeug oder bei Hinzunahme weiterer Werkzeuge kann die Verwaltung zusätzlicher werkzeugspezifischer Informationen zunächst unabhängig von den bereits erstellten Spezifikationen durch ein eigenes Graph-Ersetzungssystem spezifiziert werden. In einem zweiten Schritt sind dann u.U. die oben erläuterten Kontrollprozeduren zu modifizieren. Alle diese Kontrollprozeduren, durch die spezifiziert wird, in welcher Reihenfolge einzelne werkzeugspezifische Informationen im Modulgraphen modifiziert werden, werden in dem Graph-Ersetzungssystem **"ModuleGraphOperations"** zusammengefaßt. Dieses Graph-Ersetzungssystem enthält darüberhinaus auch alle Kontrollprozeduren und Graph-Produktionen der bisher angesprochenen Graph-Ersetzungssysteme GraphModifications, CursorMovements, ControlFlowModifications und SchemeModifications. Hinzu kommen u.U. weitere Graph-Ersetzungssysteme, durch die die Verwaltung weiterer, hier nicht erläuterter Informationen spezifiziert wird. Um sicherzustellen, daß alle Kontrollprozeduren und Graph-Produktionen aus GraphModifications, ControlFlowModifications und SchemeModifications nicht für sich alleine angewandt werden, werden sie zu **verborgenen** Kontrollprozeduren und Graph-Produktionen erklärt (vgl. Def. 3.2.8.1). Das so entstehende Graph-Ersetzungssystem ModuleGraphOperations kann schematisch wie

folgt dargestellt werden:

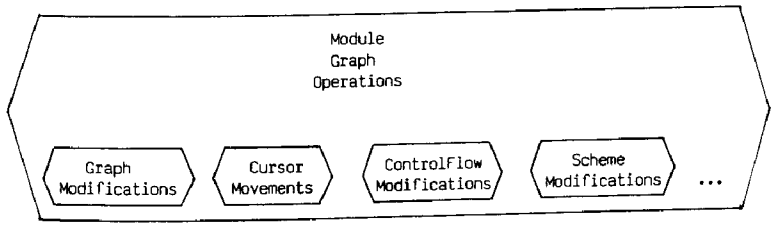


Abb. 3.3.3.2: Feinstruktur des Graph-Ersetzungssystems ModuleGraphOperations

Unsere im Abschnitt 3.2.4 eingeführte Formalisierung des Ableitungsbegriffs erlaubt es, in einer Graph-Produktion nur für einen Teil der an einem eingesetzten Knoten befindlichen Attribute neue Attributwerte zu spezifizieren. Alle sonstigen aktuellen Attributwerte werden bei einer Anwendung einer Graph-Produktion identisch übernommen. Dies ermöglicht es, in einem **unabhängigen** Graph-Ersetzungssystem für ein einzelnes Attribut etwaige Veränderungen der aktuellen Attributwerte zu spezifizieren. Der Spezifikator ist nicht gezwungen, auch alle anderen, an dem ersetzten Knoten befindlichen Attribute zu kennen.
Diese für den nicht-strukturellen Anteil des Modulgraphen mögliche Teilspezifikation ist bei unserer bisherigen Definition des Ableitungsbegriffs für den strukturellen Anteil des Modulgraphen nicht erlaubt. Da für die Anwendbarkeit einer Graph-Produktion stets ein der linken Seite entsprechender **Untergraph** im aktuellen Graph enthalten sein muß, muß jeder Spezifikator alle im Modulgraphen enthaltenen Kanten kennen. Denn in dem in Abb. 3.3.3.1 angegebenen Beispiel bedeutet dies z.B., daß die Produktion zum Eintragen der Unparsing-Schemata nicht anwendbar ist, da aufgrund fehlender Kontrollflußkanten kein entsprechender Untergraph zur linken Seite im aktuellen Modulgraphen existiert. Die im Bereich der nicht-strukturellen Information mögliche Teilspezifikation muß deshalb auch für den Bereich der strukturellen Information erlaubt sein, um die oben angesprochene unabhängige Spezifikation zu ermöglichen. Formal geschieht das dadurch, daß in Def. 3.2.4.2 anstelle eines Untergraphen nur ein hinreichend äquivalenter **Teilgraph** zur linken Seite gefordert wird. Bei einer Ersetzung werden dann alle zusätzlichen vorhandenen Kanten zwischen entsprechenden Knoten identisch übernommen (vgl. hierzu Bemerkung I.2.20 d) in /Na 79/).

Bei der **statischen Komposition von Graph-Ersetzungssystemen** wird die Komposition bereits zur Spezifikationszeit durchgeführt. Betrachtet man nämlich einmal die Graph-Produktionen, die in den Kontrollprozeduren des Graph-Ersetzungssystems ModuleGraphOperations nacheinander aufgerufen werden, erkennt man, daß sich alle Veränderungen auf nahezu **denselben Untergraphen** des aktuellen

Modulgraphen beziehen. Aus diesem Grunde liegt es nahe, derartige Modifikationen des Modulgraphen in einem Schritt durchzuführen. Dies bedeutet in der Regel nur, die **rechten Seiten** der einzelnen Graph-Produktionen **übereinanderzulegen** und dadurch eine einzige Graph-Produktion zu formulieren. Im bisher erläuterten Beispiel ergibt sich z.B. die folgende Graph-Produktion zum Eintragen des Graphinkrements für eine while-Anweisung mit allen zugehörigen Informationen.

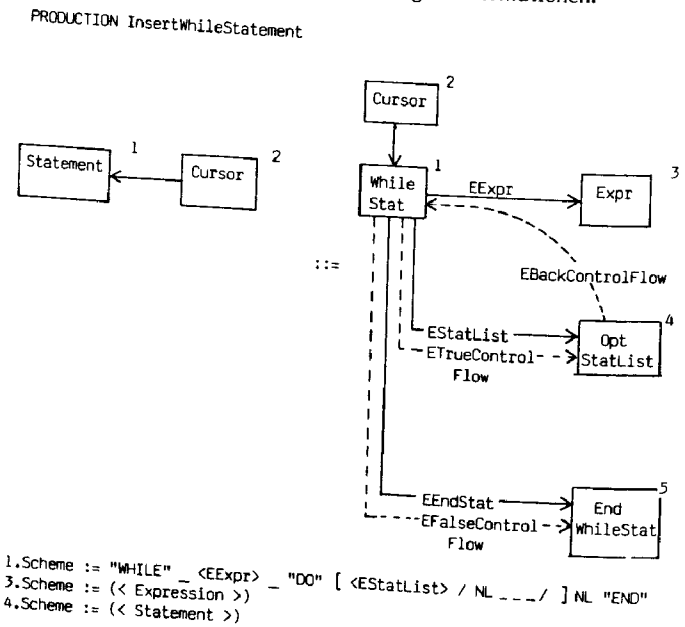


Abb. 3.3.3.3: Zusammengesetzte Graph-Produktion zum Einfügen einer while-Anweisung

Durch das, auf eine der oben vorgestellten Arten zusammengesetzte, Graph-Ersetzungssystem "ModuleGraphOperations" werden formal alle Zugriffsoperationen auf den Modulgraphen spezifiziert. Mit anderen Worten heißt das, daß dieses Graph-Ersetzungssystem eine **formale Spezifikation des abstrakten Datentyps Modulgraph** darstellt. Da alle Werkzeuge im IPSEN-System im Bereich des Programmierens-im-Kleinen einen derartigen Modulgraphen als gemeinsame, zentrale Datenstruktur benutzen, dient diese Spezifikation des Datentyps Modulgraph nun als Grundlage für eine Spezifikation der einzelnen Werkzeuge und des gesamten Systems. Eine derartige Spezifikation für das Werkzeug "syntaxgestützter Editor" wird beispielhaft in /Sc 86/ vorgestellt. In analoger Weise können auch die Werkzeuge "statische Analyse" und "Testvorbereitung" spezifiziert werden. Während der

Ausführung eines Modula-2-Programm-Moduls bzw. einzelner Ressourcen wird neben dem Modulgraphen eine zweite Datenstruktur für die Verwaltung der **Laufzeitdaten** benötigt. Die hierbei üblicherweise benötigten Komponenten Laufzeitkeller und Laufzeitheap können prinzipiell auch als **graphartige Datenstruktur** aufgefaßt werden, deren Zugriffsoperationen dann ebenfalls durch ein Graph-Ersetzungssystem spezifiziert werden können. Dieses Graph-Ersetzungssystem nennen wir im folgenden **"RuntimeDataOperations"**.

Die **Spezifikation der einzelnen Werkzeuge** besteht dann im wesentlichen aus einer Menge von Kontrollprozeduren, in denen Reihenfolge und Art der Zugriffsoperationen auf die beiden Datenstrukturen Modulgraph und Laufzeitdaten geeignet koordiniert wird. Aufgrund der Benutzung der Programmiersprache Modula-2 als Spezifikationssprache besteht außerdem die Möglichkeit, eine einfache **Ein-/Ausgabe** durch Aufruf von Ein-/Ausgabeprozeduren mit zu spezifizieren. Der **Gesamtablauf** kann dann durch eine weitere Kontrollprozedur gesteuert werden, in der Ein-/Ausgabeprozeduren und Kontrollprozeduren der Spezifikation für die einzelnen Werkzeuge aufgerufen werden. Insgesamt hat die Spezifikation des IPSEN-Systems die folgende Struktur:

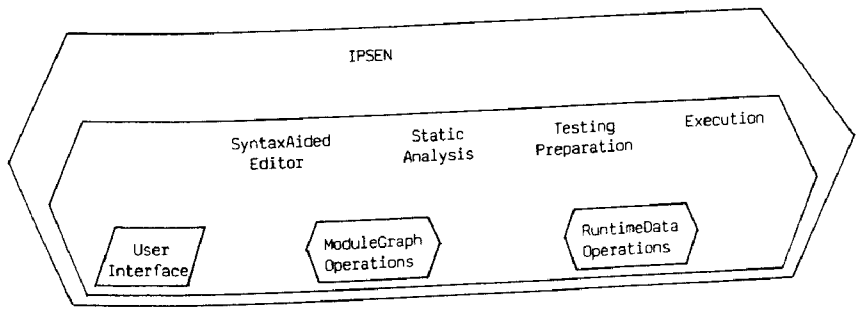


Abb. 3.3.3.4: Struktur der Spezifikation des IPSEN-Systems

Zum Abschluß wollen wir noch einmal die wichtigsten Schritte beim Erstellen einer derartigen Spezifikation zusammenfassen:

- Ausgehend von einer **normierten EBNF** haben wir drei verschiedene Arten von Graphinkrementen eingeführt. Die zugehörigen Graph-Produktionen legen fest, wie diese Graphinkremente zusammengesetzt werden können, so daß ein **abstrakter Syntaxgraph** entsteht.
- In einem zweiten Schritt haben wir erläutert, wie die Spezifikation und damit der abstrakte Syntaxgraph ergänzt werden kann, so daß zusätzlich zur kontextfreien Struktur auch **kontextsensitive Beziehungen** in dem entstehenden **Modulgraphen** dargestellt werden.

- Im nächsten Schritt wird **unabhängig** für jede einzelne werkzeugspezifische Information spezifiziert, wie diese Information bei einer Veränderung des Modulgraphen zu aktualisieren ist.
- Die auf diese Weise entstehenden Graph-Ersetzungssysteme werden zu einem Graph-Ersetzungssystem **zusammengesetzt**. Dieses Graph-Ersetzungssystem stellte dann eine formale Spezifikation des **abstrakten Datentyps** Modulgraph dar.
- Diese Vorgehensweise kann in analoger Weise auch für andere benötigte Datenstrukturen angewendet werden (z.B. Laufzeitdaten). Das Ergebnis ist eine **Schicht von Spezifikationen abstrakter Datentypen**, die die Grundlage für die weitere Spezifikation bildet.
- In der nächsten Schicht können **unabhängig** voneinander die Spezifikationen für die einzelnen **Werkzeuge** erstellt werden. Diese bestehen in der Regel aus Kontrollprozeduren, die die Zugriffe auf die in der darunterliegenden Schicht befindlichen abstrakten Datentypen koordinieren.
- In einem letzten Schritt werden dann alle Werkzeugaktivitäten koordiniert und es entsteht die **Gesamtspezifikation.**

Dieses ingenieurmäßige Vorgehen beim Erstellen der Spezifikation ist stark beeinflußt durch Methoden aus dem Bereich des Software-Engineering. Aus diesem Grunde nennen wir diese Spezifikationsmethode auch "**Graph Grammar Engineering**" (vgl. /ES 85a/).

Wir haben im Projekt IPSEN diese Spezifikationsmethode bisher in erster Linie zur Spezifikation einer zugrundeliegenden gemeinsamen **Datenstruktur** und des zugehörigen **syntaxgestützten Editors** angewandt. Über die dabei erzielten Erfahrungen und Erkenntnisse bez. des syntaxgestützten Editors im Bereich des Programmierens-im-Kleinen wird in /Sc 86/ berichtet. In /Do 84/ wird diese Vorgehensweise bei der Spezifikation eines Editors für die Programmiersprache Ada angewandt. Auch der syntaxgestützte Editor für den Bereich des Programmierens-im-Großen im IPSEN-System wird derartig spezifiziert (/LN 84/).
Da wir die Laufzeitdaten aus Effizienzgründen nicht in einer graphartigen Datenstruktur, sondern wie üblich als Feld im Hauptspeicher realisiert haben, haben wir an dieser Stelle auf eine formale Spezifikation dieses Datentyps verzichtet (vgl. Kapitel 7 bzw. /Sa 86/). Dies hat zur Folge, daß wir auch die gesamte Ausführung bisher nicht durch Graph-Ersetzungssysteme spezifiziert haben. Ebenso wurden bisher auch die Werkzeuge statische Analyse und Testvorbereitung nicht derartig formal spezifiziert. Speziell für eine Spezifikation der statischen Analyse ist es erstrebenswert, die Spezifikationsmethode weiterzuentwicklen, so daß **Graph-traversierungen** leicht spezifiziert werden können. Außerdem hätte auch eine vollständige Spezifikation aller dieser Werkzeuge den Rahmen dieser Arbeit gesprengt.

Die von uns gewählte Sprache Modula-2 als **Spezifikationssprache** für die Kontrollprozeduren ist nur als erster Ansatz zu verstehen. Es sollte Ziel der zukünftigen Forschung sein, an dieser Stelle eine Sprache zu wählen, deren Kontrollstrukturen auf die zu bearbeitenden, graphartigen Datenstrukturen zugeschnitten sind. Eine zu untersuchende Alternative könnte die in /Na 80/ erläuterte Sprache GRAPL sein.

Die bisher vorgestellte Vorgehensweise erlaubt bereits eine **modulare Spezifikation**, in der die Werkzeuge bzw. die Veränderung einzelner werkzeugspezifischer Informationen im Modulgraphen in getrennten Graph-Ersetzungssystemen spezifiziert werden können. Hier ist in Zukunft zu untersuchen, inwieweit die von anderen Spezifikationsmethoden bekannten Konzepte (z.B. Parametrisierung von Spezifikationen (vgl. /EM 85/)) auch bei der hier vorgestellten Spezifikationsmethode sinnvoll sind.

3.4 Realisierung von Graph-Ersetzungssystemen

Die Spezifikationsphase im Softwarelebenszyklus dient dazu, auf konzeptioneller Ebene, d.h. zunächst unabhängig von einer konkreten Implementierung, die Funktionalität des zu realisierenden Systems festzulegen. In einem zweiten Schritt muß eine derartige Spezifikation umgesetzt werden in eine passende Software-Architektur, die dann den Ausgangspunkt für die konkrete Implementierung darstellt. Dies bedeutet, daß erst in der darauffolgenden Phase überprüft werden kann, ob die durch die Spezifikation festgelegte Funktionalität tatsächlich den gewünschten Anforderungen an das System entspricht.
Benutzt man jedoch zur Festlegung der Funktionalität auf konzeptioneller Ebene eine **operationelle Spezifikationsmethode** hat man aufgrund der **Ausführbarkeit** einer solchen Spezifikation bereits zur Spezifikationszeit die Möglichkeit, die spezifizierte Funktionalität zu überprüfen ("rapid prototyping"). Dies gilt somit insbesondere auch für die von uns gewählten Graph-Ersetzungssysteme. Bei Benutzung eines entsprechenden **Interpreters** für derartige Graph-Ersetzungssysteme kann bereits zur Spezifikationszeit die bis dahin erstellte Spezifikation getestet werden. Wir wollen in diesem Paragraphen kurz die Architektur eines entsprechenden Systems zur Ausführung von Graph-Ersetzungssystemen vorstellen. Teile dieses Systems werden derzeit in der Diplomarbeit /Qu 86/ untersucht.

Da ein derartiger Interpreter für Graph-Ersetzungssysteme bisher noch nicht realisiert ist, dienten uns die Graph-Ersetzungssysteme bisher nur als Mittel, um auf konzeptioneller Ebene beschreiben zu können, wie die einzelnen Werkzeuge den Modulgraphen als gemeinsame Datenstruktur benutzen. Darüberhinaus stellte sich jedoch heraus, daß aus der modularen Spezifikation mit Hilfe von Graph-Erset-

zungssystemen unmittelbar ein Teil der Software-Architektur für das IPSEN-System abgeleitet werden konnte. Dieser **Zusammenhang zwischen Graph-Ersetzungssystemen und Software-Architektur** wird am Beispiel des syntaxgestützten Editors in /Sc 86/ ausführlich diskutiert.

Kommen wir nun zur Vorstellung der Architektur eines Systems zur Ausführung von Graph-Ersetzungssystemen:
Da zu solch einem System neben dem bereits erwähnten Interpreter natürlich ein Editor gehören muß, mit dem man Graph-Ersetzungssysteme eingeben und modifizieren kann, ist ein solches System nichts anderes als eine **Entwicklungsumgebung für Graph-Ersetzungssysteme.** Das bedeutet, daß viele der Überlegungen im Zusammenhang mit dem IPSEN-System unmittelbar auf diese Entwicklungsumgebung übertragen werden können. Der wesentliche Unterschied ist nur, daß nicht Modula-2-Programme sondern Graph-Ersetzungssysteme bearbeitet werden. Somit ist natürlich auch die Architektur einer solchen Umgebung sehr ähnlich zur Architektur des IPSEN-Systems. Eine Grobarchitektur des IPSEN-Systems hatten wir bereits in der Einleitung vorgestellt (vgl. Abb. 1.1). Eine Grobarchitektur für eine Entwicklungsumgebung für Graph-Ersetzungssysteme hat die folgende Form:

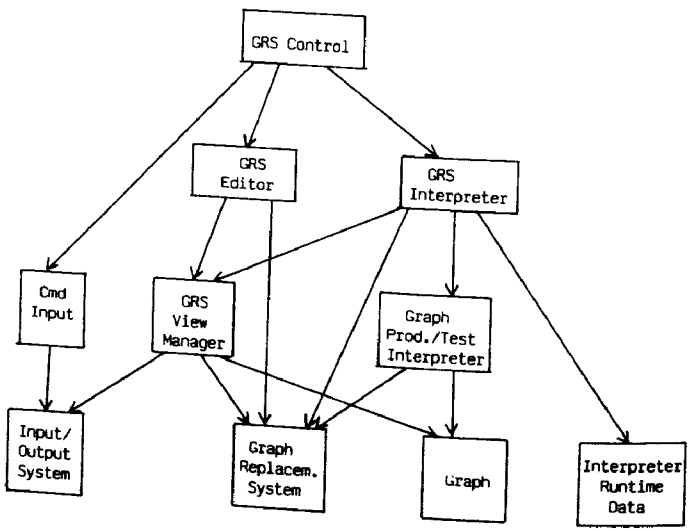


Abb. 3.4.1: Grobarchitektur einer Entwicklungsumgebung für Graph-Ersetzungssysteme

Die einzelnen Bestandteile stellen Komponenten dar, die bei einer konkreten Implementierung aus mehreren Moduln zusammengesetzt sind. Die Kanten zwischen

den einzelnen Komponenten beschreiben, welche Komponente auf welche anderen Komponenten Zugriff hat.

Mit Hilfe des kommandogesteuerten, syntaxgestützten "GRS-Editors" (GRS Abkürzung für Graph Replacement System) können Graph-Ersetzungssysteme **eingegeben und modifiziert** werden. Alle erstellten Graph-Ersetzungssysteme werden in der Komponente "GraphReplacementSystem" abgespeichert. Hierbei wird es sich um ein Datenbanksystem handeln, in dem Kontrollprozeduren und Graph-Produktionen mit effizienter Speicher- und Zugriffstechnik abgelegt werden können. Zur Darstellung auf dem Bildschirm in der Komponente "Input/OutputSystem" wird eine interne Darstellung von Kontrollprozeduren und Graph-Produktionen durch die Komponente "GRSViewManager" zunächst in eine externe Darstellung transformiert ("Unparsing-Prozeß").

Ein erstelltes Graph-Ersetzungssystem kann dann vom "GRSInterpreter" **interpretiert** werden. Da derzeit die Kontrollprozeduren in Modula-2 geschrieben sind, ist ein derartiger Interpreter im wesentlichen ein Modula-2-Interpreter. Für die Interpretation nötige Laufzeitdaten werden in der Komponente "InterpreterRuntimeData" abgelegt. Beim Aufruf von Graph-Produktionen und Unter- bzw. Teilgraphentests wird der "GraphProduction/TestInterpreter" aktiviert, der Tests und Veränderungen auf dem "Graph" durchführt. Dieser Graph kann z.B. der Modulgraph sein, oder für den Fall, daß der Interpreter im IPSEN-System auch mit Graph-Ersetzungssystemen spezifiziert wurde, z.B. der Graph für die Verwaltung der Laufzeitdaten. Der GraphProduction/TestInterpreter kann effizient arbeiten, da die aktuelle Bearbeitungsposition im Graphen aufgrund der Existenz eines **eindeutigen Cursorknotens** direkt ermittelt werden kann. Darüberhinaus kann der Spezifikator die Effizienz des Interpretationsvorgangs steigern, indem er die im letzten Abschnitt vorgestellte statische Komposition von Graph-Ersetzungssystemen im Gegensatz zur dynamischen Komposition verwendet.

Die gesamte Verarbeitung wird von der Komponente "GRSControl" gesteuert, die Kommandos von "CmdInput" einliest und den Editor bzw. Interpreter entsprechend aktiviert.

4 Klassifikation von Zugriffsoperationsrealisierungen

Im letzten Kapitel haben wir erläutert, wie auf formale Art die Gestalt der internen Datenstruktur Modulgraph festgelegt werden kann. Dieser Modulgraph wird als gemeinsame, zentrale Datenstruktur für die Realisierung aller zum Programmieren-im-Kleinen gehörenden Werkzeuge im IPSEN-System verwandt. Ziel der nächsten Kapitel ist, zu jedem im Kapitel 2 vorgestellten Kommando zu beschreiben, wie dieses Kommando realisiert wird. Dies wird weiterhin auf konzeptioneller Ebene geschehen; allerdings verzichten wir im folgenden auf eine präzise Spezifikation z.B. mit Graph-Ersetzungssystemen zugunsten einer mehr informellen Darstellung.

Jede dieser Realisierungen eines Kommandos besteht aus einer Folge von Aufrufen von **Zugriffsoperationen** auf die gemeinsame Datenstruktur Modulgraph. Hierbei kann noch einmal unterschieden werden zwischen **schreibenden** und **lesenden** Zugriffsoperationen. Schreibende Zugriffsoperationen werden vor allem bei der Realisierung von Editorkommandos benutzt, um den Modulgraphen geeignet zu modifizieren. Lesende Operationen werden z.B. für die Realisierung des Werkzeugs statische Analyse benötigt, um bestimmte Eigenschaften des Modulgraphen zu überprüfen. Hierauf werden wir im Kapitel 5 ausführlich eingehen.

Daneben gibt es eine Reihe von Aktivitäten, bei denen die im Modulgraphen verkapselte Information in eine andere Darstellung transformiert wird. Ein Beispiel für ein derartiges **Transformationsproblem** ist der Unparser, der aus der Modulgraphdarstellung die zugehörige textuelle Darstellung erzeugt. Im Gegensatz zu obigen Zugriffsoperationen auf eine gemeinsame Datenstruktur besteht hier die Realisierung aus der abwechselnden Aktivierung von Zugriffsoperationen von (mindestens) zwei Datenstrukturen. Bei der Realisierung des Unparsers ist z.B. eine Folge von lesenden Zugriffen auf den Modulgraph vermischt mit einer Folge von schreibenden Zugriffen auf eine weitere Datenstruktur, die den erzeugten Quelltext repräsentiert. Wir kommen hierauf im Kapitel 6 ausführlich zurück.

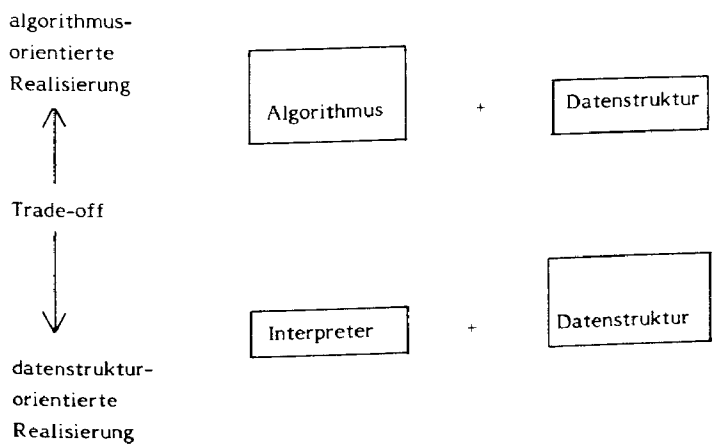
Betrachtet man diese verschiedenen Zugriffsoperationen auf Datenstrukturen einmal etwas genauer, erkennt man eine Reihe von Gemeinsamkeiten, die zunächst einmal unabhängig von einer konkreten Datenstruktur bzw. einer konkreten Anwendung sind. Wir wollen in diesem Kapitel einige dieser **allgemeinen Charakteristika** von Zugriffsoperationen herausarbeiten. In den nächsten Kapiteln werden wir dann detailliert erläutern, welche Informationen im einzelnen im Modulgraphen abgelegt werden, wie zugehörige lesende und schreibende Zugriffsoperationen aussehen und welche zusätzlichen Datenstrukturen neben dem Modulgraph für die einzelnen Transformationsprobleme benötigt werden.

Jede Zugriffsoperation auf eine Datenstruktur wird durch einen mehr oder weniger umfangreichen **Algorithmus** realisiert. Die verschiedenen Charakteristika

eines derartigen Algorithmus erörtern wir an Hand von drei Fragen:

1) Wie ist die Information verkapselt, die den Algorithmus beschreibt?

Diese Frage stellt sich stets zu einem frühen Zeitpunkt und ist von dem Entwerfer der Realisierung einer Zugriffsoperation zu entscheiden. Gemeint ist hiermit der häufig auftretende **"Trade-off zwischen Algorithmus und Datenstruktur"**. Für eine stärker algorithmus-orientierte Realisierung einer Zugriffsoperation bedeutet dies, daß man sämtliche Informationen fest im Programmtext verkapselt und nur sehr wenige Informationen in zusätzlichen Datenstrukturen ablegt. Für eine mehr datenstruktur-orientierte Realisierung einer Zugriffsoperation bedeutet dies hingegen, daß möglichst viele Informationen in einer oder mehreren zusätzlichen Datenstrukturen (z.B. Tabellen) abgelegt werden, so daß der Algorithmus zu einem reinen Interpreter dieser Datenstrukturen (also z.B. Tabelleneinträge) entartet. Schematisch sieht das etwa folgendermaßen aus:



Zwischen diesen beiden Extrema sind natürlich beliebig viele Mischformen denkbar.

Bei der Abwägung der Vor- bzw. Nachteile dieser beiden Alternativen sind vor allem die beiden Kriterien a) Laufzeiteffizienz und b) Adaptabilität zu betrachten:

ad a) **Laufzeiteffizienz:**

Da bei einer datenstruktur-orientierten, interpretativen Lösung im Interpreter alle möglichen verschiedenen Tabelleneinträge berücksichtigt und bei einer Interpretation zunächst erkannt werden müssen, ist dieser Lösungsweg in der Regel erheblich laufzeitaufwendiger als rein algorithmische Lösungen.

ad b) **Adaptabilität:**

Vor der Entscheidung für einen bestimmten Lösungsweg ist zu bedenken, welche geänderten Anforderungen zu erwarten sind und wie diese realisiert werden könnten. Grundsätzlich erhöht die Verkapselung von Information in Datenstrukturen (z.B. Tabellen) die Adaptabilität, da bei geänderten Anforderungen nur die Datenstrukturen (bzw. Teile davon) auszutauschen sind. Ist die Information im Algorithmus verkapselt, bedeutet die Anpassung an geänderte Anforderungen einen vollständig neuen Edier-/Compilationszyklus. Die datenstruktur-orientierte Vorgehensweise ist jedoch nur dann von wirklichem Vorteil, wenn z.B. die Tabelleneinträge einfach modifizierbar sind oder die zu ändernden Stellen im Algorithmus schwer zu lokalisieren sind.

Wir werden bei den im folgenden erläuterten Zugriffsoperationen Beispiele für beide möglichen Vorgehensweisen kennenlernen. Auf eine interpretative, tabellengesteuerte Vorgehensweise gehen wir dabei insbesondere bei der Darstellung des Unparsers im Kapitel 6 ein. Beispiele für eine algorithmische Lösung findet man u.a bei der Darstellung der Analysealgorithmen im Kapitel 5.

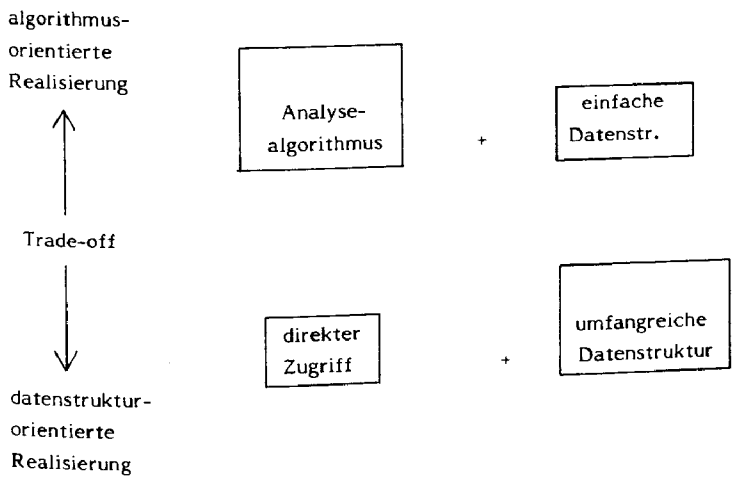
Ein weiteres, in dieser Arbeit nicht weiter vertieftes Charakteristikum für die Realisierung von Zugriffsoperationen ergibt sich aus der Unterscheidung, auf welche Art und Weise eine mehr datenstruktur-orientierte oder mehr algorithmus-orientierte Realisierung erstellt wird. Hierbei erstreckt sich die Bandbreite von reinen "**per Hand**" erstellten Realisierungen bis hin zu "**automatisch generierten**" Realisierungen. Bei automatisch generierten Realisierungen muß die Eingabe für den Generator in entsprechend formalisierter Form vorliegen. Wir werden im Kapitel 6 bei der Darstellung des Unparsers ein Beispiel für einen derartigen Generator kennenlernen.

Während die oben erläuterte Frage bereits vor der Erstellung der Realisierung einer Zugriffsoperation zu entscheiden ist, sind die nächsten beiden Fragen relevant für den Zeitpunkt der Ausführung einer Zugriffsoperation:

2) Wie ist die Information verkapselt, die durch den Algorithmus bearbeitet werden soll?

Mit dieser Frage ist gemeint, ob die während der Ausführung benötigte Information aus einer Datenstruktur durch einen **einfachen, direkt lesenden Zugriff** ermittelt werden kann oder zunächst durch einen **umfangreichen Analysealgorithmus** aufgebaut werden muß. Als Beispiel haben wir bereits oben erläutert, daß durch die graphartige Darstellung eines Programms die kontextfreie Struktur und einige kontextsensitive Beziehungen direkt ausgedrückt werden. Dies bedeutet z.B. für die Realisierung des syntaxgestützten Editors, daß ein direkter lesender Zugriff ausreicht, um entscheiden zu können, ob ein bestimmtes Inkrement eingefügt werden darf. Hätte man als Datenstruktur einen einfachen Zeichenstrom gewählt, hätte diese Entscheidung

zunächst einen aufwendigen Analysealgorithmus erfordert. Dieser beim Editor zugunsten einer umfangreichen Datenstruktur entschiedene Trade-off zwischen einem (aufwendigen Analyse-)Algorithmus und einer (graphartigen) Datenstruktur, ist für jede Realisierung einer Zugriffsoperation abzuwägen.
Schematisch sieht dieser Trade-off zwischen Algorithmus und Datenstruktur hier wie folgt aus:



Auch hier sind wieder beliebig viele Mischformen zwischen den beiden Extrema denkbar.

Bei der Entscheidung für eine bestimmte Vorgehensweise ist hier vor allem die Frage der **Laufzeiteffizienz** zu diskutieren: Ein direkter lesender Zugriff auf eine Datenstruktur ist sicherlich erheblich effizienter als ein aufwendiger Analysealgorithmus. Andererseits muß man jedoch bedenken, daß jede durch einen direkten lesenden Zugriff ermittelbare Information zuvor in der Datenstruktur abgelegt worden sein muß. Außerdem bedeutet eine umfangreichere Datenstruktur in der Regel einen erhöhten Verwaltungsaufwand bei einer Modifikation der Datenstruktur, um die Konsistenz der abgelegten Informationen zu gewährleisten.

Jede direkt lesbare Information muß zu einem früheren Zeitpunkt durch einen schreibenden Zugriff in der Datenstruktur abgelegt worden sein. Dies führt uns bei der Charakterisierung von Zugriffsoperationen zur dritten Frage:

3) Wann wird eine direkt lesbare Information in der Datenstruktur abgelegt?

Hierbei sind die folgenden Vorgehensweisen denkbar:

a) sofortige Aktualisierung:

Bei dieser Strategie wird bei jeder Veränderung eines Teils der Datenstruktur die dort abgelegte Information unmittelbar anschließend auf den aktuellen Stand gebracht.

b) Aktualisierung in einem Vorlauf:

Bei dieser Strategie wird bei einer Veränderung eines Teils der Datenstruktur die dort abgelegte Information nicht sofort aktualisiert, jedoch u.U. gelöscht oder als "nicht aktuell" gekennzeichnet. Bei Aufruf einer lesenden Zugriffsoperation auf diese Information wird dann in einem Vorlauf sämtliche Information bzw. nur die gelöschten oder gekennzeichneten Teile der Information auf den aktuellen Stand gebracht, so daß die anschließende Ausführung der Zugriffsoperation ein rein lesender Zugriff ist.

c) Aktualisierung auf Anforderung:

Bei dieser Strategie wird bei einer Veränderung eines Teils der Datenstruktur wie im Fall b) verfahren. Bei Aufruf einer Zugriffsoperation wird dann jedoch nur dann an einer bestimmten Position in der Datenstruktur die Information aktualisiert, wenn für die Ausführung dieser Zugriffsoperation diese Information benötigt wird.

Diese drei Strategien haben für die im IPSEN-System benutzte Datenstruktur Modulgraph folgende Bedeutung: Wir haben bereits an Beispielen erläutert, wie werkzeugspezifische Informationen durch zusätzliche Knoten, Kanten oder Attribute im Modulgraphen abgelegt werden können. Für jede dieser Informationen muß entschieden werden, zu **welchem Zeitpunkt** die derart abgelegten Informationen bei einer Veränderung des Modulgraphen zu **aktualisieren** sind. Als Beispiel für die in Strategie a) skizzierte Vorgehensweise haben wir erläutert, daß sämtliche Kanten zur Darstellung kontextsensitiver Zusammenhänge unmittelbar nach einer Veränderung eines Inkrements aktualisiert werden. Dasselbe gilt für die im Modulgraphen abgelegten Kontrollflußkanten bzw. für die für den Unparser benötigten Unparsing-Schemata. Bei der Vorstellung der Realisierung der einzelnen Werkzeuge werden wir auch Beispiele für die unter b) und c) beschriebenen Vorgehensweisen kennenlernen (vgl. z.B. Kapitel 7).

In den folgenden Kapiteln werden wir bei der Vorstellung der Realisierung der einzelnen Kommandos bei jeder benötigten Information die unter 2) gestellte Frage diskutieren, wie diese Information am effizientesten ermittelt werden kann. Bei den Informationen, die zusätzlich im Modulgraphen abzulegen sind, ist dann zusätzlich zu diskutieren, welche der obigen drei Vorgehensweisen gewählt werden sollen. Insgesamt sind bei diesen Entscheidungen u.a. folgende Kriterien zu beachten:

Jede weitere Information sollte nur dann im Modulgraphen abgelegt werden, wenn

auf sie relativ **häufig zugegriffen** wird.
Soll die Information im Modulgraphen abgelegt werden, ist zu untersuchen, wie groß der Mehraufwand ist, diese Information **sofort** bei jeder Modifikation eines Inkrements zu **aktualisieren.** Bei geringem Mehraufwand kann Strategie a) gewählt werden.
Falls dieser Mehraufwand hoch ist und der lesende Zugriff auf diese Information im Vergleich zu Inkrementmodifikationen relativ selten ist, sollte Strategie b) oder c) für die Aktualisierung dieser Information gewählt werden.

5 Realisierung der statischen Analyse

In diesem Kapitel erläutern wir, wie die im Rahmen der Anforderungsdefinition im Paragraphen 2.2 vorgestellten Kommandos zur statischen Analyse realisiert werden. Gemäß der im vorigen Kapitel vorgestellten Kriterien diskutieren wir, welche Informationen zusätzlich im Modulgraph abgelegt bzw. welche Informationen algorithmisch ermittelt werden sollen. Hierbei werden wir zeigen, daß für die Realisierung einiger Kommandos die bisher erläuterten, im Modulgraphen abgelegten Informationen ausreichen. Bei anderen Kommandos muß diese Information verfeinert werden, damit durch einen direkten lesenden Zugriff die Analyse realisiert werden kann. Schließlich werden wir auch Beispiele kennenlernen, bei denen zusätzlich zu den im Modulgraphen abgelegten Informationen weitere Informationen durch einen Analysealgorithmus ermittelt werden müssen.

Die meisten der im Modulgraphen abgelegten Informationen werden unmittelbar nach jeder Veränderung eines Inkrements des Modulgraphen durch eine Editoraktion aktualisiert. Aufgrund dieser **inkrementellen Vorgehensweise** bei der Aktualisierung solcher, u. a. auch für die statische Analyse benötigten Informationen sprechen wir bei diesem Werkzeug auch von einer **inkrementellen statischen Analyse**. Diese Inkrementalität bei der Realisierung eines Werkzeugs ist nicht zu verwechseln mit der **inkrementorientierten** Arbeitsweise eines Werkzeugs an der Benutzerschnittstelle.

5.1 Ermittlung nicht angewandter Deklarationen

Im Abschnitt 3.2.6 haben wir erläutert, daß die kontextsensitive Beziehung zwischen einem angewandten Auftreten eines Bezeichners und der dazugehörenden Deklaration durch eine entsprechend markierte Kante im Modulgraphen dargestellt wird. Als Beispiel seien hier noch einmal die mit "EUseDec" bzw. "ESetDec" markierten Kanten erwähnt, die von einem benutzenden bzw. setzenden Auftreten eines Datenobjektbezeichners zur zugehörigen Deklaration gezogen werden. Diese Kanten zwischen einem angewandten Auftreten eines Bezeichners und einer Deklaration können nun auch dazu benutzt werden zu überprüfen, ob ein deklarierter Bezeichner irgendwo angewandt wird. Denn alle die Knoten von deklarierten Bezeichnern, die keine derartige einlaufende Kante besitzen, werden an keiner Stelle angewandt. Das bedeutet, daß zur Realisierung des Kommandos An - non-applied (declarations/imports) und analoger Kommandos die während des **Edierens eingetragenen Kanten** zur Darstellung kontextsensitiver Beziehungen **ausreichen**, um durch einen lesenden Zugriff, d.h. Untergraphentest, die nötigen Informationen zu erhalten.

5.2 Unterscheidung zwischen lokalen und globalen Datenobjekten

Im Abschnitt 2.2.2 haben wir erläutert, daß bei mehreren Kommandos zur statischen Analyse die Information benötigt wird, welche zur aktuell untersuchten Prozedur **global deklarierten Datenobjekte** bei Aufruf dieser Prozedur gesetzt oder benutzt werden können. Insbesondere sei hier das Kommando Ag - global (data objects) erwähnt, dessen Aufgabe es ist, eben diese Datenobjekte zu ermitteln. Gemäß den im Kapitel 4 erläuterten Kriterien ist nun zu diskutieren, ob diese Information aus einem gegebenen Modulgraphen algorithmisch ermittelt werden soll oder durch zusätzliche Knoten bzw. Kanten im Modulgraphen direkt ausgedrückt werden soll. Diese bei jeder einzelnen werkzeugspezifischen Information durchzuführende Diskussion erläutern wir im folgenden sehr ausführlich. An dieser beispielhaft geführten Diskussion wollen wir deutlich machen, welche Überlegungen dazu führen, eine bestimmte Information direkt im Modulgraphen auszudrücken bzw. algorithmisch ermitteln zu lassen.

Bei den bei Aufruf einer Prozedur P angewandten, zu P global deklarierten Datenobjekten können **drei Arten** unterschieden werden:

a) die angewandten Auftreten befinden sich im Rumpf von P,
b) die angewandten Auftreten befinden sich im Rumpf einer Prozedur P', die lokal zu P deklariert ist und im Rumpf von P aufgerufen wird,
c) die angewandten Auftreten befinden sich im Rumpf einer Prozedur Q, die in einem die Prozedurdeklaration von P statisch umgebenden Deklarationsteil deklariert ist.

Die folgende Abbildung stellt diese drei Situationen schematisch dar:

```
MODULE Example1;

VAR A : ....;

PROCEDURE Q;
BEGIN
   A := ..., (c)
END Q;

PROCEDURE P;
   PROCEDURE P';
   BEGIN
      A := ...; (b)
   END P';
BEGIN
   A := ...; (a)
   P';
   Q;
END P;

BEGIN
   ...
END Example1.
```

Abb. 5.2.1: Angewandtes Auftreten global deklarierter Datenobjekte

Bei dem bisher vorgestellten Vorgehen wird zwischen jedem angewandten Auftreten eines Datenobjekts und der dazugehörenden Deklaration eine mit "ESetDec" bzw. "EUseDec" markierte Kante gezogen. Um nun nur diejenigen angewandten Auftreten eines Datenobjekts zu ermitteln, die bei einem Aufruf der Prozedur P erreicht werden können, ist sowohl der Rumpf von P als auch transitiv aller im Rumpf von P aufgerufener Prozeduren zu durchsuchen, ob ein solches angewandtes Auftreten eines Datenobjekts in ihm enthalten ist. Da hierbei u.U. große Teile des Modulgraphen durchsucht werden müssen, werden wir einen Teil dieser gesuchten Information in der Datenstruktur, also dem Modulgraphen ablegen. Hierzu werden wir die durch EUseDec- bzw. ESetDec-Kanten dargestellte **Information** weiter verfeinern. Dies ermöglicht dann, die zu Punkt a) und b) gehörenden angewandten, global deklarierten Datenobjekte durch einen lesenden Zugriff auf den Modulgraphen zu ermitteln.

Die EUseDec- bzw. ESetDec-Kanten werden während des Edierens beim Eintragen eines Bezeichners im Anweisungsteil in den Modulgraphen aufgenommen. Bei der dabei durchzuführenden Suche nach der zugehörigen Deklaration werden im zugrundeliegenden abstrakten Syntaxgraphen von der aktuellen Stelle an aufwärts alle statisch umgebenden Deklarationsteile durchsucht, bis die zugehörige Deklarationsstelle gefunden ist. Bei dieser Aufwärtssuche wird also bereits ermittelt, ob das aktuelle Datenobjekt **lokal** oder **global** deklariert ist. Da es jedoch (bisher) keine unterschiedlichen Kantenmarkierungen gibt, wird dieses Wissen nicht im Modulgraphen abgelegt. Andererseits ist diese Information wichtig für die Ermittlung global deklarierter Datenobjekte. Aus diesem Grunde ersetzen wir die eingeführten EUseDec- bzw. ESetDec-Kanten zwischen angewandten Auftreten von Datenobjektbezeichnern und ihren zugehörigen Deklarationsstellen durch die folgenden Knoten und Kanten:

- Alle Prozedurdeklarationen erhalten als weiteren Bestandteil eine Liste von Bezeichnern, in der alle Datenobjekte aufgeführt werden, die im Rumpf dieser Prozedur oder in statisch tiefer liegenden Prozeduren global angewandt werden. Der Listenkopfknoten wird mit **"VarCopyList"** markiert.
- Von der angewandten Stelle eines global deklarierten Datenobjekts im Rumpf einer Prozedur P wird eine mit **"ELocSetDec"** bzw. **"ELocUseDec"** markierte Kante zum entsprechenden Knoten in der an dieser Prozedurdeklaration hängenden Liste von Bezeichnern gezogen.
- Von einem Datenobjektbezeichner in einer derartigen Liste wird eine mit **"EGlobUse"** bzw. mit **"EGlobSet"** markierte Kante zum entsprechenden Knoten der Liste der statisch umgebenden Prozedur bzw. zur Deklarationsstelle gezogen.

Wir verdeutlichen diese zusätzlichen Knoten und Kanten an dem folgenden Beispiel:

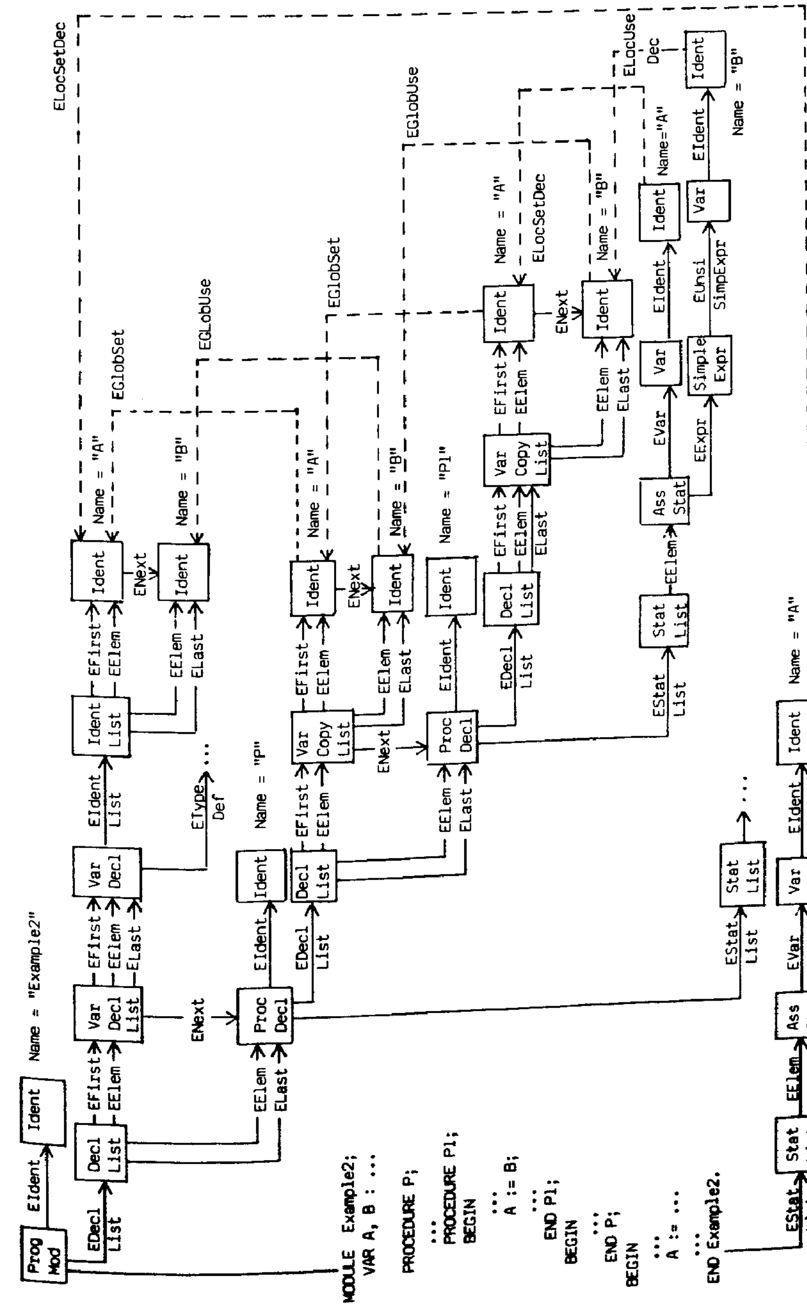


Abb. 5.2.2: Modulgraphausschnitt mit VarCopy-Listen

Die Grundidee obiger Vorgehensweise besteht darin, ein **Bündel** an einem Knoten einlaufender Kanten dadurch zu strukturieren, daß **Teilmengen** dieses Bündels bereits vorher an **Kopien** dieses Knotens zusammengeführt werden und nur noch **eine** Kante von dieser Kopie zum Zielknoten gezogen wird:

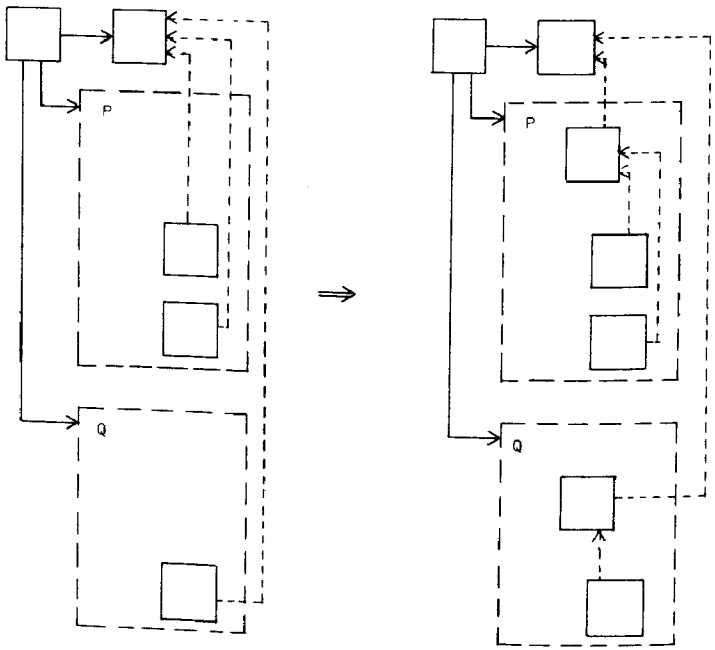


Abb. 5.2.3: Strukturierung eines Kantenbündels durch zusätzliche Knoten und Kanten

Dadurch wird zum einen der **Informationsgehalt erhöht,** da nun zum Beispiel im Teilgraphen P nur noch die an der Kopie einlaufenden Kanten relevant sind, und zum anderen werden die Verbindungen zwischen verschiedenen Teilen des Graphen dünner. Andererseits ist zu beachten, ob diese zusätzlichen Knoten und Kanten auch einen **erhöhten Verwaltungsaufwand** während des Edierens bedeuten, da diese kontext-sensitiven Beziehungen ja stets bei einer Änderung des Modulgraphen sofort zu aktualisieren sind.

Hierzu untersuchen wir die folgenden Modifikationen des Modulgraphen während des Edierens:

- **Eintragen** eines neuen **angewandten** Auftretens eines Datenobjekts:
  Falls im Deklarationsteil der aktuellen Prozedur die zugehörige Deklaration steht, wird eine mit "ELocSetDec" bzw. mit "ELocUseDec" markierte Kante gezogen.

Falls dies nicht der Fall ist, wird überprüft, ob bereits ein entsprechender Knoten in der Liste der Kopien von Datenobjektbezeichnern (VarCopyList) existiert. Im positiven Fall muß dann ebenfalls nur eine mit "ELocSetDec" bzw. "ELocUseDec" markierte Kante gezogen werden. Zur Typüberprüfung kann der zugehörige Typ über eine Folge von mit "EGlobUse" bzw. "EGlobSet" markierten Kanten gefunden werden. In diesem Fall erleichtert also die Existenz der Kopien auch die Suche nach einer zugehörigen Deklaration. Denn nur in dem Fall, daß dieses Datenobjekt zum ersten Mal global angewandt wird, müssen in allen Prozedurdeklarationen zwischen der aktuellen Anwendung und der Deklaration dieses Datenobjekts eine Kopie angelegt und entsprechende Kanten gezogen werden.

- **Löschen** eines **angewandten** Auftretens eines Datenobjekts:
  Zunächst muß die auslaufende mit "ELocSetDec" bzw. "ELocUseDec" markierte Kante gelöscht werden. Außerdem kann auch die Kopie des Bezeichners in der aktuellen und allen statisch höher liegenden Prozedurdeklarationen gelöscht werden, falls keine weitere Anwendung dieses Datenobjekts in dem entsprechenden oder einem statisch tiefer liegenden Prozedurrumpf existiert.

- **Löschen** einer **Deklaration** eines Datenobjekts:
  Falls eine Deklaration gelöscht werden soll, zu der noch einlaufende mit "ELocSet/UseDec" bzw. "EGlobSet/Use" markierte Kanten existieren, muß überprüft werden, ob in einem statisch höher liegenden Deklarationsteil eine passende Typdeklaration mit demselben Bezeichner existiert. Im positiven Fall kann in der aktuellen Prozedurdeklaration eine Kopie dieses Bezeichners angelegt werden, der dann auch die einlaufenden Kanten übernimmt. Alle statisch tiefer liegenden Prozedurdeklarationen können unverändert bleiben.

- **Eintragen** einer **Deklaration:**
  Beim Eintragen einer Datenobjektdeklaration kann aufgrund der Existenz einer Kopie mit demselben Bezeichner sofort ermittelt werden, ob sich der Gültigkeitsbereich von Bezeichnern in der aktuellen oder statisch tiefer liegenden Prozeduren ändert. Existiert eine derartige Kopie, übernimmt die neue Deklaration alle dort einlaufenden Kanten und die Kopie kann gelöscht werden.

Insgesamt ergibt sich, daß der zusätzliche Verwaltungsaufwand sehr gering ist und außerdem die während des Edierens durchzuführenden Modulgraphmodifikationen teilweise erheblich effizienter realisiert werden können. Dies rechtfertigt die Entscheidung, diese Information im Modulgraph, also in der Datenstruktur abzulegen und nicht algorithmisch zu ermitteln.

Die an einer Prozedurdeklaration hängende Liste von Kopien von Datenobjektbezeichnern enthält alle Datenobjekte, die in diesem Prozedurrumpf oder einem statisch tiefer liegenden Prozedurrumpf angewandt werden (vgl. Punkt a) und

b) in Abb. 5.2.1). Da wir bisher bei einem Eintrag einer Kopie eines global deklarierten Datenobjekts an einer Prozedurdeklaration nicht überprüfen, ob diese Prozedur überhaupt an irgendeiner Stelle aktiviert wird, kann diese Liste von Kopien von Bezeichnern u.U. Datenobjekte enthalten, die nie angewandt werden. Da andererseits solche Prozedurdeklarationen sofort mit dem Kommando An - non-applied (declarations/imports) gefunden werden können, gehen wir hier von der vereinfachten Annahme aus, daß **alle deklarierten Prozeduren** auch an mindestens einer Stelle **aufgerufen werden.**

Zusätzlich zu den oben unter Punkt a) und b) angesprochenen Situationen sind nun noch die Datenobjekte zu ermitteln, deren Deklaration global zur aktuellen Prozedurdeklaration P steht und die im Rumpf einer Prozedur Q angewandt werden, deren Deklaration in einem die Deklaration von P umfassenden Deklarationsteil steht und die selbst im Rumpf von P aufgerufen wird (vgl. Punkt c) in Abb. 5.2.1). Da letzteres auch über mehrere Stufen gehen kann, ist der unter dem Namen "calling-Graph" (vgl. /AU 79/) bekannte Graph näher zu untersuchen.
Ein "**calling-Graph**" ist ein gerichteter, knotenmarkierter Graph, bei dem die Menge der Knotenmarkierungen alle in einem Programm auftretenden Prozedurbezeichner umfaßt. Zu jeder Prozedurdeklaration existiert ein entsprechend markierter Knoten und zwischen zwei mit P bzw. Q markierten Knoten existiert eine Kante genau dann, wenn die Prozedur Q im Rumpf von P aufgerufen wird. Man betrachte hierzu das folgende Beispiel:

```
MODULE Example3;
  ...
  PROCEDURE P;
    ...
    PROCEDURE P1;
    BEGIN
      ...
      Q;
      ...
    END P1;
  BEGIN
    ...
    P1;
    ...
  END P;
  PROCEDURE Q;
    ...
  END Q;
BEGIN
  ...
  P;
  Q;
  ...
END Example3.
```

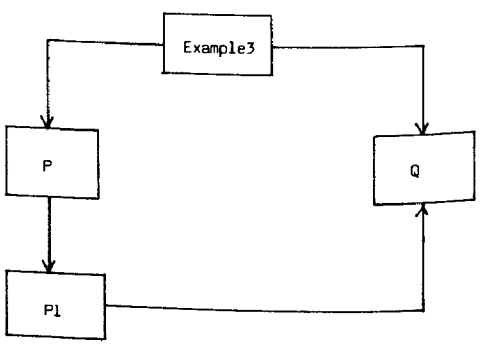


Abb. 5.2.4: Ausschnitt aus einem Modula-2-Modul mit zugehörigem calling-Graph

Sollen nun alle bei Aufruf von P möglichen angewandten globalen Datenobjekte

bestimmt werden, ist in dem zugehörigen calling-Graph der **transitive Abschluß** über alle von P aus erreichbaren Prozeduren zu bilden und für jede Prozedur die Menge der in ihrem Rumpf angewandten, zu P global deklarierten Datenobjekte zu bestimmen.

Wie wir oben erläutert haben, ist die Menge der im Rumpf einer Prozedur und in allen statisch tiefer liegenden Prozeduren angewandten globalen **Datenobjekte** im Modulgraphen dargestellt durch eine Liste von Knoten mit Kopien dieser Datenobjektbezeichner. Diese sind also durch einen rein lesenden Zugriff ermittelbar. Um jedoch zu ermitteln, welche **Prozeduren** im Rumpf einer Prozedur aufgerufen werden, muß im Gegensatz zu der im calling-Graph direkt vorliegenden Information im Modulgraphen der gesamte Rumpf der aktuell untersuchten Prozedurdeklaration durchsucht werden. Denn die bisher im Modulgraphen vorhandenen, mit "EUseDec" markierten Kanten zwischen der Aufrufstelle einer Prozedur und ihrer Deklaration geben keine Auskunft darüber, in welchem Prozedurrumpf diese Prozedur aktiviert wird. Diese Information kann jedoch analog zu der obigen Vorgehensweise bei global deklarierten Datenobjekten ohne großen Mehraufwand im Modulgraphen abgelegt werden, indem Kopien der Prozedurbezeichner angelegt werden und die mit "EUseDec" markierten Kanten ersetzt werden durch eine mit "ELocUseDec" markierte Kante und mit "EGlobUse" markierte Kantenfolge. Die an einer Prozedurdeklaration P anhängende Liste von Kopien von Prozedurbezeichnern ("**ProcCopyList**") umfaßt dann alle Prozeduren, die im aktuellen und allen statisch tiefer liegenden Rümpfen aufgerufen werden und in einem P umfassenden Deklarationsteil deklariert sind.

Vergleicht man die durch die Kopien von Datenobjektbezeichnern (VarCopyList) im Modulgraphen abgelegte Information mit der durch einen calling-Graph gegebenen Information, so ist in diesem Fall der transitive Abschluß über alle **statisch tiefer** liegenden Prozeduren bereits durchgeführt worden. Um **alle** bei Aufruf einer Prozedur P u. U. angewandten globalen Datenobjekte zu bestimmen, ist nur noch der **transitive Abschluß** über alle zu P global deklarierten und bei Aufruf von P u. U. aktivierten Prozeduren durchzuführen. Alle diese Prozedurbezeichner befinden sich in der an der Prozedurdeklaration von P befindlichen ProcCopy-Liste.
Dieser restliche transitive Abschluß wird durch den folgenden Algorithmus realisiert, der in einer pseudo-programmiersprachlichen Notation angegeben wird.

```
Algorithmus zur Bestimmung aller bei Aufruf einer Prozedur
möglicherweise angewandten, global deklarierten Datenobjekte

Globale Variablen:  GlobalObjects      : SET OF Identifier
                    VisitedProcedures  : SET OF Node

PROCEDURE DetermineGlobalDataObjects ( ActProc : Node );
BEGIN
  FOR ALL VarId IN VarCopyList OF ActProc DO
        GlobalObjects := GlobalObjects + { VarId }
  OD;

  VisitedProcedures := VisitedProcedures + { ActProc } ;

  FOR ALL ProcId IN ProcCopyList OF ActProc DO
        NextProc := DeclarationNode OF ProcId;

        IF NextProc NOT IN VisitedProcedures THEN
           DetermineGlobalDataObjects( NextProc )
        FI
  OD;
END DetermineGlobalDataObjects;
```

Abb. 5.2.5: Ermittlung global deklarierter Datenobjekte

In der Variablen GlobalObjects werden die ermittelten, global deklarierten Datenobjekte abgelegt. In der Menge VisitedProcedures werden die Deklarationsknoten der untersuchten Prozeduren abgelegt, so daß jede Prozedur nur ein einziges Mal durchsucht wird und z.B. bei rekursiven Prozeduren ein Abbruchkriterium für den obigen Algorithmus vorliegt.

Dieser Algorithmus kann auf verschiedene Arten verfeinert werden:

- Ist man an den zugehörigen **Deklarationen** der ermittelten Datenobjektbezeichner interessiert, kann über eine Folge von mit "EGlobUse" bzw. mit "EGlobSet" markierten Kanten der zugehörige Deklarationsknoten ermittelt werden und die Ergebnismenge entsprechend ergänzt werden.
- Ist man z.B. an global deklarierten Datenobjekten interessiert, für die mindestens ein **setzendes** Auftreten vorliegt, ist zu überprüfen, ob an der Kopie eine einlaufende, mit "ELocSetDec" bzw. "EGlobSet" markierte Kante existiert, bevor man den zugehörigen Bezeichner in die Ergebnismenge aufnimmt.

Da in **Standardprozeduren** keine globalen Datenobjekte verändert werden, müssen Aufrufe von Standardprozeduren beim obigen Algorithmus nicht berücksichtigt werden. Aus diesem Grunde werden derartige Prozeduren auch nicht in den ProcCopy-Listen mitverwaltet. Es wird nur eine mit "EUseDec" markierte Kante zwischen dem Aufruf einer Standardprozedur und der Deklaration gezogen (vgl. Abb. 3.2.6.2).

Der obige Algorithmus bildet die Grundlage für die Realisierung des Kommandos Ag - global (data objects). Vergleicht man diese Realisierung mit den im Kapitel 4 vorgestellten Kriterien, liegt hier eine **Mischform** zwischen einer algorithmus- und einer datenstruktur-orientierten Realisierung vor. Während die zu einer Prozedurdeklaration global deklarierten Datenobjekte und Prozeduren in der Datenstruktur in Listen von Kopien abgelegt und bei einer Veränderung der Datenstruktur stets sofort aktualisiert werden, wird der transitive Abschluß über aufgerufene, global deklarierte Prozeduren erst bei Aktivierung des Kommandos durch einen entsprechenden Algorithmus durchgeführt.
Bei einer rein datenstruktur-orientierten Vorgehensweise müßte an jeder Prozedurdeklaration das Ergebnis eines solchen transitiven Abschlusses abgelegt werden. Dies bedeutete jedoch einen enormen Aktualisierungsaufwand bei einer Änderung des Modulgraphen. So müßte z.B. beim Löschen des angewandten Auftretens eines Datenobjekts überprüft werden, bei Ausführung welcher Prozeduren dadurch ein mögliches angewandtes Auftreten dieses Datenobjekts entfällt. Diese Überlegungen führten zu dem oben erläuterten Kompromiß, nur einen Teil der Information im Modulgraphen abzulegen und den restlichen Teil auf Anforderung algorithmisch zu ermitteln.

5.3 Ermittlung angewandter Auftreten eines Datenobjekts

Die im letzten Paragraphen erläuterte verfeinerte Darstellung der kontextsensitiven Beziehung zwischen der Deklaration und einem angewandten Auftreten eines Datenobjekts wird auch bei der Realisierung des Kommandos Ao - (setting/using) occurrences ausgenutzt. Bei Aktivierung dieses Kommandos sollen zu einem deklarierten Datenobjekt **alle setzenden und benutzenden Auftreten** dieses Datenobjekts angezeigt werden. Damit der Benutzer verfolgen kann, wo sich diese angewandten Auftreten im Quelltext befinden, sollen die entsprechenden Stellen dem Benutzer in **textueller Reihenfolge** angezeigt werden. Dies bedeutet, daß ausgehend von der Programmeinheit, in deren Deklarationsteil das zu untersuchende Datenobjekt deklariert ist, der Anweisungsteil dieser Programmeinheit und aller lokalen Prozedurdeklarationen in textueller Reihenfolge nach angewandten Auftreten zu durchsuchen ist. Durch die Existenz der oben eingeführten Kanten und Knoten zur Darstellung der Beziehung zwischen einer Deklaration und zugehörigen angewandten Auftreten kann dieser im allgemeinen Fall sehr aufwendige Suchalgorithmus erheblich **effizienter** realisiert werden. Zunächst einmal müssen nur die Anweisungsteile untersucht werden, in denen mindestens eine mit "ELocSetDec" bzw. "ELocUseDec" markierte Kante zum Deklarationsknoten bzw. zu einer entsprechenden Kopie in einer VarCopy-Liste existiert. Außerdem muß der Suchalgorithmus nur auf solche lokalen Prozedurdeklarationen ausgedehnt werden, bei denen eine

Kopie in der VarCopy-Liste und eine mit "EGlobUse" bzw. "EGlobSet" markierte Kante existiert.

Der entsprechende Algorithmus wird in der folgenden Abbildung in pseudo-programmiersprachlicher Notation angegeben:

```
Algorithmus zur Ermittlung aller angewandten Auftreten zu
einem Datenobjekt

PROCEDURE ShowAllOccurrencesOfIdent ( ProcDeclNode : Node;
                                      ActIdentNode : Node );
BEGIN
   (* Teil 1 *)
   NodeList := LIST OF SourceNodes OF
                  (ELocUseDec- OR ELocSetDec-)Edge AT ActIdentNode;

   FOR ALL Statement IN StatementList OF ProcDeclNode
       AND (NodeList not empty) DO

      IF TestOccurrenceOfActIdentInStatement
            (Statement, NodeList, ActIdentOccurrence) THEN
         DisplayMessageAndOccurrence( Statement, ActIdentOccurrence)
      FI;
   OD;

   (* Teil 2 *)
   NodeList := LIST OF SourceNodes OF
                  (EGlobSet- OR EGlobUse-)Edge AT ActIdentNode;

   FOR ALL ProcDecl IN DeclList OF ProcDeclNode
       AND (NodeList not empty) DO

      IF TestOccurrenceOfActIdentInVarCopyList
            (ProcDecl, NodeList, ActIdentOccurrence) THEN
         ShowAllOccurrencesOfIdent( ProcDecl, ActIdentOccurrence)
      FI;
   OD;

END ShowAllOccurrencesOfIdent;
```

Abb. 5.3.1: Algorithmus zur Ermittlung angewandter Auftreten

Im **ersten Teil** wird der Anweisungsteil der aktuellen Prozedurdeklaration in textueller Reihenfolge durchsucht. Bei jeder Anweisung wird durch die Funktion "TestOccurrenceOfActIdentInStatement" überprüft, ob ein angewandtes Auftreten des aktuellen untersuchten Datenobjekts vorliegt. Dies liegt z.B. vor, wenn das untersuchte Datenobjekt in der linken oder rechten Seite einer Zuweisungsanweisung oder in dem booleschen Ausdruck einer bedingten Anweisung oder einer Schleifen-anweisung angewandt wird. Jedes gefundene angewandte Auftreten wird sofort aus NodeList entfernt. Dies hat zur Folge, daß die Suche vorzeitig abgebrochen werden kann, sobald NodeList keine Elemente mehr enthält und somit alle angewandten Auftreten in diesem Anweisungsteil gefunden wurden.

Im zweiten Teil werden alle lokalen Prozedurdeklarationen untersucht. Hierbei wird

die Analyse nur dann auf eine Prozedurdeklaration ausgedehnt, wenn eine Kopie des aktuell untersuchten Datenobjekts in der VarCopy-Liste dieser Prozedurdeklaration enthalten ist. Dies überprüft die Funktion "TestOccurrenceOfActIdentInVarCopy-List". Existiert eine derartige Kopie, wird die Suche durch einen rekursiven Aufruf auf die Prozedurdeklaration ausgedehnt. Auch hier kann die Suche vorzeitig abgebrochen werden, wenn alle globalen angewandten Auftreten ermittelt wurden.

Der obige Algorithmus zeigt, wie die im letzten Abschnitt erläuterte verfeinerte Darstellung der kontextsensitiven Beziehung zwischen einer Deklaration und einem angewandten Auftreten eines Datenobjekts eine effiziente Realisierung eines weiteren Analysekommandos ermöglicht. Auch hier ist die Ursache für die effiziente Realisierung der gefundene **Kompromiß** zwischen inkrementell verwalteter, im Modulgraphen abgelegter und algorithmisch ermittelter Information.

5.4 Ermittlung nicht erreichbarer Anweisungen

Ein ähnlicher Kompromiß zwischen einer datenstruktur- und algorithmus-orientierten Vorgehensweise befindet sich auch bei der Realisierung des Kommandos Au - unreachable (statements) zum Auffinden **nicht erreichbarer Anweisungen.** Wir hatten bereits im Abschnitt 3.3.2 erläutert, welche zusätzlichen Kanten zur Darstellung des Kontrollflusses im Modulgraphen abgelegt werden. Diese Information kann nun dazu benutzt werden, nicht erreichbare Anweisungen zu entdecken.

Hierzu werden in der textuellen Reihenfolge nacheinander alle Anweisungen überprüft, ob der entsprechende Knoten im Modulgraphen eine einlaufende Kontrollflußkante besitzt. Eine derartige Kante existiert nicht, wenn die untersuchte Anweisung unmittelbar hinter einer return- oder exit-Anweisung steht. Falls die untersuchte Anweisung unmittelbar hinter einer loop-Anweisung steht, ist zu überprüfen, ob innerhalb der loop-Anweisung mindestens eine exit-Anweisung steht. Im Modulgraphen bedeutet dies, daß im Endeknoten der loop-Anweisung mindestens eine einlaufende "EControlFlow"-Kante existiert. Falls die untersuchte Anweisung unmittelbar hinter einer if-, case- oder with-Anweisung steht, muß überprüft werden, ob der Endeknoten dieser Anweisung erreicht werden kann. Steht z.B. in einer if-Anweisung im then-Teil, in allen elsif-Teilen und dem else-Teil als letzte Anweisung stets eine return- oder exit- Anweisung, kann die hinter der if-Anweisung stehende Anweisung nicht erreicht werden. Aus diesem Grunde muß überprüft werden, ob in mindestens einem Teil als letzte Anweisung nicht eine return- oder exit-Anweisung steht. Da innerhalb einer if-, case- oder with-Anweisung eine weitere derartige Anweisung stehen kann und dies über mehrere Stufen gehen kann, muß auch im Modulgraphen dementsprechend u.U. über mehrere Stufen gesucht werden. Diesen Test übernimmt der in Abb. 5.4.1 in pseudo-programmiersprachlicher Notation

geschriebene rekursive Algorithmus:

```
Algorithmus zum Test, ob der aktuell untersuchte Endeknoten
von einer davorstehenden Anweisung über Kontrollflußkanten
erreicht werden kann

PROCEDURE TestControlFlowAtEndNode ( ActNode : Node ) : BOOLEAN;

BEGIN

  FOR ALL SourceNode OF ControlFlowEdge AT ActNode DO

    IF SourceNode = EndNode
              OF { IfStat, CaseStat, WithStat, LoopStat } THEN
      IF TestControlFlowAtEndNode ( SourceNode ) THEN
        RETURN TRUE
      FI
    ELSE
      RETURN TRUE
    FI

  OD;
  RETURN FALSE

END TestControlFlowAtEndNode;
```

Abb. 5.4.1: Algorithmus zum Test der Erreichbarkeit eines Ende-Knotens

Damit können wir nun den Algorithmus formulieren, durch den in einem gegebenen Inkrement, z.B. einer Prozedurdeklaration, nacheinander alle Anweisungen überprüft werden, ob sie erreichbar sind.

```
Algorithmus zum Entdecken nicht erreichbarer Anweisungen

PROCEDURE CetermineUnreachableStatements ( ActInc : Node );

BEGIN
  FOR ALL Statement IN ActInc DO

      UnreachableStatementFound := FALSE;
      PredNode := SourceNode OF ControlFlowEdge AT Statement;

      IF PredNode not exists THEN
        UnreachableStatementFound := TRUE

      ELSIF PredNode =
              EndNode OF {IfStat, CaseStat, WithStat, LoopStat } THEN
        UnreachableStatementFound := TestControlFlowAtEndNode( PredNode )
      FI;

      IF UnreachableStatementFound THEN
        DisplayMessageAndStatement ( Statement )
      FI
  OD

END DetermineUnreachableStatements;
```

Abb. 5.4.2: Algorithmus zum Entdecken nicht erreichbarer Anweisungen

Durch obigen Algorithmus wird stets nur die **erste** nicht erreichbare Anweisung ermittelt, die textuell hinter einer exit- oder return-Anweisung bzw. einer loop-Anweisung ohne exit-Anweisung steht. Der Algorithmus ist geeignet zu erweitern, wenn auch alle textuell folgenden, ebenso nicht erreichbaren Anweisungen dem Benutzer angezeigt werden sollen.

Durch diesen Algorithmus kann das Kommando Au - unreachable (statements) realisiert werden. Durch die Benutzung der im Modulgraphen vorhandenen Kontrollflußkanten und einen zusätzlichen Algorithmus haben wir auch hier eine **Mischform** einer datenstruktur- und algorithmus- orientierten Vorgehensweise.

5.5 Ermittlung von Datenflußinformationen

Das letzte der im Paragraphen 2.2 vorgestellten Kommandos der statischen Analyse dient dazu, ausgehend von einem setzenden Auftreten eines Datenobjekts alle im Kontrollfluß folgenden benutzenden Auftreten zu ermitteln, bei denen auf diesen gesetzten Wert zugegriffen wird. Dies ist ein Beispiel für eine Vielzahl von Analyseproblemen, bei denen der in einem Modul verkapselte **Datenfluß** untersucht wird. Derartige Analysen sind gut bekannt im Bereich der klassischen Vorgehensweise, Programme zu entwickeln. Nahezu jeder **Compiler** enthält heutzutage eine **Optimierungsphase**, in der auf einer Zwischencodedarstellung des Programms Optimierungen durchgeführt werden, um laufzeit- und speicherplatzeffizienten Code zu erhalten (vgl. /AU 79/). Solch einer Optimierung geht stets eine statische Analyse voraus, in der der Kontroll- bzw. Datenfluß untersucht wird. Bei der Untersuchung der Frage, inwieweit dort etablierte Techniken auf die Realisierung des IPSEN-Werkzeugs der statischen Analyse übertragen werden können, sind folgende **Unterschiede** zu beachten:

- Ein Compiler, und damit der in ihm enthaltene Optimierer, wird erst aktiviert, wenn ein Programm vollständig erstellt worden ist. Demgegenüber können im IPSEN-System Kommandos des Werkzeugs der statischen Analyse bereits während der Programmerstellung auf vielen Inkrementebenen aktiviert werden.
- Die Optimierungsphase ist in der Regel eine selbständige Phase, die von der zuvor durchgeführten kontextfreien und kontextsensitiven Analyse völlig unabhängig abläuft. Das bedeutet, daß in früheren Phasen ermittelte Informationen nicht mehr zur Verfügung stehen und mühsam wieder ermittelt und abgelegt werden müssen in einer für die statische Analyse und den Optimierer speziellen Datenstruktur. In IPSEN dagegen existiert nur eine einzige Datenstruktur zur Darstellung des aktuell bearbeiteten Programms, auf der alle Werkzeuge arbeiten. Dies bedeutet, daß die einmal ermittelten, im Graphen abgelegten Informationen (z.B. kontextsensitive Zusammenhänge, Kontrollflußkanten) von allen Werkzeugen und damit auch von der statischen Analyse benutzt werden können.

Wir wollen in diesem Paragraphen an einem Beispiel erläutern, wie aus der Datenflußanalyse bekannte Probleme in einer Entwicklungsumgebung wie IPSEN gelöst werden können, in der alle Werkzeuge eine gemeinsame Datenstruktur, den Modulgraphen benutzen. Hierbei ist es **nicht** unser Ziel, **neue Analysealgorithmen** im Bereich der Datenflußanalyse zu entwickeln bzw. für dort noch offene, ungelöste Probleme eine Lösung anzubieten.
Die Idee, dem IPSEN-Benutzer die Ergebnisse einer derartigen Datenflußanalyse zugänglich zu machen, ist **neu** im Bereich von Entwicklungsumgebungen, die auf einer anweisungsorientierten Programmiersprache basieren. Es ist nicht bekannt, daß in einem anderen Forschungsprojekt ein vergleichbares Werkzeug angeboten wird.

In der klassischen Compilerbautheorie unterscheidet man im Bereich der statischen Analyse zwischen **intraprozeduraler** und **interprozeduraler** Analyse (vgl. /He 77/). Während im ersten Fall nur innerhalb einer Prozedur analysiert wird, zählen zur zweiten Art auch Verfahren, die den Datenfluß über Prozedurgrenzen hinweg betrachten. Bei der Beschreibung des Kommandos As - set/use chains im Abschnitt 2.2.2 haben wir bereits erläutert, daß wir in IPSEN eine **benutzergesteuerte** Form der **interprozeduralen Analyse** zur Verfügung stellen. Dies heißt, daß der Benutzer bei jedem während der Analyse gefundenen Prozeduraufruf explizit entscheiden muß, ob die Analyse auf den zugehörigen Prozedurrumpf ausgedehnt werden soll. Dies hat den Vorteil, daß der IPSEN-Benutzer entscheiden kann, welche Prozedurdeklarationen bereits in einem analysierbaren Zustand sind. Außerdem kann der Benutzer dafür sorgen, daß einzelne Prozedurdeklarationen nur ein einziges Mal analysiert werden und bei rekursiven Prozeduren die Analyse auch abbricht.

Wir haben erläutert, daß die kontextsensitiven Beziehungen zwischen dem angewandten Auftreten eines Datenobjekts und der zugehörigen Deklaration durch eine Folge von Kanten im Modulgraphen dargestellt wird. Auch die durch das Kommando As - set/use chains zu ermittelnden Beziehungen sind kontextsensitive Zusammenhänge zwischen zwei angewandten Auftreten desselben Datenobjekts. Prinzipiell könnten derartige Beziehungen auch im Modulgraphen durch zusätzliche Kanten ausgedrückt werden, so daß die Realisierung des Kommandos As aus reinen Leseoperationen bestehen könnte. Wie man sich jedoch leicht überlegt, ist eine **inkrementelle Verwaltung** derartiger Kanten während des Edierens sehr aufwendig. Aus diesem Grunde besteht die Realisierung des Kommandos As - set/use chains aus einem Algorithmus, bei dem allerdings auf die inkrementell verwalteten Kontrollflußkanten und Kanten zwischen angewandtem und deklarierendem Auftreten eines Datenobjekts zurückgegriffen wird.

Wesentliche Aufgabe des zu entwickelnden Algorithmus ist somit, ausgehend von einer bestimmten Position alle erreichbaren Kontrollflußwege innerhalb des aktuellen Prozedurrumpfs zu durchlaufen. Dabei sind folgende Restriktionen zu beachten:

Jede Kontrollflußkante sollte **nur einmal** durchlaufen werden, damit dasselbe benutzende Auftreten dem Benutzer nicht mehrfach angezeigt wird.
Die einzelnen benutzenden Auftreten sollten dem Benutzer in **textueller Reihenfolge** angezeigt werden, damit dieser nicht die Übersicht über den auf dem Bildschirm angezeigten Quelltext verliert.

Die Grundidee des Algorithmus besteht nun darin, ausgehend von der aktuellen Position **alle Kontrollflußwege sukzessive zu durchlaufen** und alle gefundenen benutzenden Auftreten des Datenobjekts dem IPSEN-Benutzer anzuzeigen. Dabei wird die Suche auf einem Kontrollflußweg abgebrochen, sobald ein neues setzendes Auftreten entdeckt wird. Damit dem Benutzer die gefundenen benutzenden Auftreten in textueller Reihenfolge angezeigt werden, wird beim Erreichen des Endeknotens einer bedingten Anweisung, z.B. if-Anweisung, nach der Analyse des then-Teils zunächst mit der Analyse der elsif-part-Liste fortgesetzt, bevor die Analyse hinter der bedingten Anweisung fortgesetzt wird. Zur Verwaltung der Informationen, welche Teile des Graphen bereits durchsucht wurden, verwenden wir einen zusätzlichen **Stack** als Hilfsdatenstruktur, der die üblichen Zugriffsoperationen Push, Pop und TopOfStack besitzt.

Wir geben den Algorithmus in einer pseudo-programmiersprachlichen Notation in Abb. 5.5.1 an. Hierbei beschränken wir uns bei der Darstellung auf die Diskussion einer if- bzw. while-Anweisung im Bereich der Kontrollstrukturen. Andere Kontrollstrukturen werden ähnlich behandelt.

```
Algorithmus zum Auffinden benutzender Auftreten hinter einem
setzenden Auftreten eines Datenobjekts

PROCEDURE ShowUsingOccurrencesOfActIdent ( ActPos   : Node;
                                           ActIdent : Node );

BEGIN

    Push( Bottom );
    ActPos := TargetNode OF (EControlFlow- OR EBackControlFlow-)Edge
                         AT ActPos;

    REPEAT
      CASE ActPos OF

        AssStat : Display all using occurrences of ActIdent in AssStat;
                  IF setting occurrence of ActIdent THEN
                     ActualizeStackAndPosition ( ActPos )
                  ELSE
                     ActPos := TargetNode OF (EControlFlow- OR
                                 EBackControlFlow-)Edge AT ActPos
                  FI |
```

```
WhileStat :
          IF TopOfStack = ActPos THEN
             Pop;
             ActPos := TargetNode OF
                       EFalseControlFlow-Edge AT ActPos

          ELSIF ActPos IN VisitedLoops THEN
             ActualizeStackAndPosition( ActPos )

          ELSE
             VisitedLoops := VisitedLoops + {ActPos};
             Push( ActPos );
             Display all using occurrences of ActIdent in
                                  loop condition;
             ActPos := TargetNode OF
                       ETrueControlFlow-Edge AT ActPos
          FI |

IfStat :  Push( ActPos, TruePart );
          Display all using occurrences of ActIdent in condition;
          ActPos := TargetNode OF
                    ETrueControlFlow-Edge AT ActPos |

EndIfStat :
          IF TruePart AT TopOfStack THEN
             Pop;
             Push( ActPos, FalsePart, MarkTruePart );
             ActPos := TargetNode OF
                       EFalseControlFlow-Edge AT TopOfStack
          ELSE
             ActPos := TargetNode OF EControlFlow-Edge AT ActPos
          FI |

ProcCall :
          Display all using occurrences of ActIdent in
                                           parameterlist;
          IF ( ActIdent used as parameter ) OR
             ( global occurrence of ActIdent ) THEN
             Push( ActPos );
             RETURN   (* for user interaction *)
          ELSE
             ActPos := TargetNode OF
                       EControlFlow-Edge AT ActPos
          FI |
...

UNTIL ( ActPos IN {EndProgMod, EndProcDecl} ) OR ( Stack empty )

END ShowUsingOccurrencesOfActIdent;
```

Abb. 5.5.1: Algorithmus zur Ermittlung von set/use-Ketten

Der Algorithmus läuft solange von Anweisung zu Anweisung bis das Ende des zu untersuchenden Anweisungsteils erreicht wird oder auf jedem möglichen Kontrollflußweg ein neues setzendes Auftreten entdeckt worden ist. Letzteres erkennt man daran, daß das zu Beginn auf dem Stack abgelegte Element 'Bottom' gelöscht worden ist und der Stack leer ist. Dies wird verständlich bei der folgenden Erläuterung des Algorithmus:

- Zu Beginn wird die im Kontrollfluß unmittelbar folgende Anweisung untersucht.

Diese ist über eine mit "EControlFlow" oder "EBackControlFlow" markierte Kante zu erreichen.

- Falls diese Anweisung eine **Zuweisung** ist, werden dem Benutzer zunächst alle eventuell vorhandenen benutzenden Auftreten des aktuell untersuchten Datenobjekts ActIdent angezeigt. Wird das Datenobjekt durch diese Zuweisungsanweisung neu gesetzt, wird dieser Kontrollflußweg nicht weiter untersucht. Anhand der auf dem Stack abgelegten Informationen wird überprüft, ob ein noch zu untersuchender Kontrollflußweg existiert und ActPos auf die nächste zu untersuchende Anweisung gesetzt. Ansonsten wird das Bottom-Element auf dem Stack gelöscht und dadurch die Analyse beendet.
- Ist die aktuell zu untersuchende Anweisung eine **while-Anweisung**, wird überprüft, ob dieser Knoten schon einmal erreicht worden ist. Hierbei sind zwei Fälle zu unterscheiden:
  - Wurde dieser Knoten über eine mit "EBackControlFlow" markierte Kante nach durchgeführter Überprüfung des Schleifenrumpfes erreicht, entspricht dieser Knoten dem obersten Stackelement. In diesem Fall wird das oberste Stackelement gelöscht und hinter der while-Anweisung die Suche fortgesetzt.
  - Andererseits kann dieser Punkt im Kontrollfluß auch ein zweites Mal erreicht werden, wenn diese Schleife innerhalb einer weiteren Schleifenanweisung liegt. Abb. 5.5.2 gibt ein Beispiel für eine derartige Situation, bei der das setzende Auftreten eines Datenobjekts X innerhalb von zwei while-Schleifen liegt.

```
...
WHILE ... DO
  ...
  WHILE ... DO
    ...
    X := ...
  END
END;
...
```

Abb. 5.5.2

Bei der Ausführung des obigen Algorithmus wird ausgehend von dem setzenden Auftreten zunächst die innere Schleife durchlaufen und anschließend die äußere Schleife untersucht. Beim wiederholten Erreichen der inneren Schleife muß dann erkannt werden, daß dieser Teil schon einmal untersucht wurde, damit der Algorithmus terminiert. Dies heißt etwas formaler, daß, ausgehend von einem setzenden Auftreten eines Datenobjekts nur der **Baumanteil** aller dort startenden Kontrollflußwege untersucht werden soll. Alle einmal untersuchten Schleifenanweisungen werden deshalb in der Menge VisitedLoops abgelegt. Dies geschieht beim erstmaligen Erreichen einer derartigen Schleifenanweisung. Wird eine Schleifenanweisung ein zweites Mal erreicht, wird die Untersuchung

auf diesem Kontrollflußweg abgebrochen und ActPos anhand der auf dem Stack abgelegten Informationen auf die nächste zu untersuchende Anweisung gesetzt. Beim erstmaligen Erreichen der while-Schleife wird dieser Knoten zusätzlich auch auf dem Stack abgelegt, damit der Schleifenrumpf beim nächsten Erreichen des Schleifenkopfs nicht noch einmal untersucht wird. Danach werden alle benutzenden Auftreten von ActIdent in der Schleifenbedingung dem Benutzer angezeigt und die Überprüfung mit dem Schleifenrumpf fortgesetzt.

- Bei Erreichen einer **if-Anweisung** wird der entsprechende Knoten auf dem Stack abgelegt. Zusätzlich wird vermerkt, daß man zunächst nur den then-Teil (d.h. TruePart) überprüft hat. Nach Ausgabe der benutzenden Auftreten in der Schleifenbedingung wird die Überprüfung mit dem then-Teil fortgesetzt.
- Erreicht man den zu einer **if-Anweisung gehörenden Endeknoten** nach der Überprüfung des then-Teils, merkt man sich dies auf dem Stack (MarkTruePart). Diese Information wird benötigt, um bei einer vorzeitigen Beendigung der Überprüfung des elsif- bzw. else-Teils zu wissen, daß die Suche nach weiteren benutzenden Auftreten hinter der if-Anweisung fortgesetzt werden muß. Wird der Endeknoten der if-Anweisung nach Überprüfung des elsif- bzw. else-Teils erreicht, kann die Suche mit der folgenden Anweisung fortgesetzt werden.
- Bei einem **Prozeduraufruf** werden zunächst alle Auftreten von ActIdent in der aktuellen Parameterliste dem Benutzer angezeigt. Bei einer Benutzung von ActIdent als aktuellen Parameter bzw. bei einem globalen angewandten Auftreten hat der Benutzer die Möglichkeit, die Analyse auf die zugehörige Prozedurdeklaration auszudehnen. Hierzu wird die Aufrufstelle zunächst auf den Stack gelegt und der Algorithmus unterbrochen, um den Dialog mit dem Benutzer durchzuführen. Da der Stack während einer solchen Unterbrechung nicht zerstört wird, kann die Analyse anschließend geeignet fortgesetzt werden. Bei der Ermittlung der globalen angewandten Auftreten muß der bei der Realisierung des Kommandos Ag - global data objects beschriebene transitive Abschluß über alle aufgerufenen Prozeduren vollzogen werden. Dies bedeutet, daß bei der Realisierung dieses Kommandos As - set/use chains ein Großteil der inkrementell verwalteten Informationen benutzt wird, darüberhinaus aber der algorithmische Anteil auch noch relativ hoch ist.
An dieser Stelle kann der Algorithmus effizienter gestaltet werden, wenn man den **Benutzer entscheiden** läßt, ob bei Aufruf einer Prozedur das aktuell untersuchte Datenobjekt global angewandt wird. Dies erspart den u. U. aufwendigen Algorithmus zum Bilden des oben angesprochenen transitiven Abschlusses.

Anstelle der Benutzung eines **zusätzlichen Stacks** ist auch denkbar, die für die Verwaltung des Durchlaufs benötigten Informationen unmittelbar an den entsprechenden Knoten im **Modulgraphen** abzulegen. Dies bedeutet dann andererseits, daß

bei einem vorzeitigen Abbruch der Analyse der Modulgraph durchsucht werden muß, um alle diese Hilfsinformationen wieder zu entfernen. Bei Benutzung eines vom Modulgraphen unabhängigen Stacks muß in diesem Fall nur der Stack gelöscht werden.

Da eine vollständige und umfassende Datenflußanalyse in der Regel sehr aufwendig ist, wird häufig darauf verzichtet, sämtliche Spezialfälle zu untersuchen, und dadurch der Analysevorgang an bestimmten Stellen abgekürzt (vgl. /AU 79/). Um in diesen Situationen dann jedoch keine falschen Analyseergebnisse zu erzielen, müssen vom Algorithmus sogenannte "konservative Annahmen" gemacht werden. Diese Annahmen haben allerdings zur Folge, daß das Ergebnis einer solchen Analyse sehr ungenau ist und wenig Information enthält. Man betrachte hierzu das folgende Beispiel aus /AU 79/, S. 505: Gegeben ist eine Folge von drei Zuweisungsanweisungen, und es ist zu untersuchen, ob der Wert der Variablen X in der dritten Anweisung identisch mit dem Wert dieser Variablen in der ersten Anweisung ist.

```
A := B + X;     (* 1. Anweisung *)
Y := C;         (* 2. Anweisung *)
D := B + X;     (* 3. Anweisung *)
```

Abb. 5.5.3

Dies kann, muß aber nicht zutreffen. Denn wenn z.B. X und Y formale call-by-reference-Parameter in einer Prozedur P sind und ein Prozeduraufruf P(Z,Z) existiert, bezeichnen X und Y denselben Speicherplatz, so daß die in der zweiten Anweisung stehende Wertzuweisung an die Variable Y auch den Wert der Variablen X verändert. Diese Situation ist ein Beispiel für das in der Literatur allgemein als "aliasing-Problem" bezeichnete Problem. Eine analoge Situation kann auch bei der Benutzung dynamischer Datenstrukturen auftreten, wenn zwei Objekte eines Zeigertyps auf denselben Speicherplatz zeigen.

Im obigen Algorithmus 5.5.1 haben wir dieses Problem bisher nicht berücksichtigt. Es existieren unterschiedliche Möglichkeiten, dieses und ähnliche Probleme zu lösen:

- Macht man eine konservative Annahme, so würde man hier davon ausgehen, daß alle formalen call-by-reference-Parameter denselben Speicherplatz bezeichnen können, so daß die in der zweiten Anweisung stehende Wertzuweisung den Wert der Variablen X verändern kann.
- Erweitert man den Analysealgorithmus, so ist zu überprüfen, ob z.B. ein Prozeduraufruf P(Z,Z) dieser Prozedur existiert.
- In einer interaktiven Umgebung wie IPSEN hat man schließlich die weitere Möglichkeit, den IPSEN-Benutzer zu fragen, ob bei der Untersuchung der Variablen X davon ausgegangen werden muß, daß eine weitere Variable denselben Speicherplatz bezeichnet.

Wählt man eine der ersten beiden, algorithmischen Lösungen, so sind auch hier die

bereits im Modulgraphen verkapselten Informationen hilfreich. Z.B. kann man sich bei einer konservativen Annahme nur auf die call-by-reference-Parameter beschränken, die von demselben Typ sind wie die untersuchte Variable.

Wir haben in diesem Kapitel erläutert, wie die im Paragraphen 2.2 vorgestellten Kommandos des Werkzeugs der statischen Analyse unter Benutzung der zentralen Datenstruktur Modulgraph realisiert werden können. Hierbei haben wir im wesentlichen **drei unterschiedliche Vorgehensweisen** vorgestellt:

- Für die Realisierung einiger Kommandos reichte die bereits im Modulgraphen abgelegte Information aus.
- Um andere Kommandos effizient realisieren zu können, wurde die im Modulgraphen abzulegende Information weiter verfeinert, indem zusätzliche Knoten und Kanten eingeführt wurden.
- Im letzten Fall geschah die Realisierung durch einen aufwendigen Graph-Algorithmus.

Wir haben bereits an verschiedenen Stellen darauf hingewiesen, welche **Erweiterungen** des Werkzeugs der statischen Analyse im IPSEN-System denkbar sind. Die Realisierung dieser und aller sonstigen Erweiterungen kann **analog** zu den oben exemplarisch erläuterten Überlegungen und Vorgehensweisen durchgeführt werden.

6 Ein-/Ausgabe eines Modulgraphen

Der Modulgraph stellt für die Spezifikation der Software-Entwicklungsumgebung IPSEN auf konzeptioneller Ebene die Datenstruktur zur Darstellung eines Modula-2-Moduls dar. Es ist offensichtlich, daß eine derartige, graphartige Darstellung eines Moduls nicht dazu geeignet ist, dem IPSEN-Benutzer die aktuelle Gestalt seines Modula-2-Moduls anzuzeigen. Hinzu kommt, daß der Modulgraph als gemeinsame Datenstruktur aller Werkzeuge eine Vielzahl von werkzeugspezifischen Informationen enthält, die für den Benutzer in dieser Form irrelevant sind. Aus diesem Grunde muß der Modulgraph bzw. zumindest der aktuell auf dem Bildschirm darstellbare Teilgraph zunächst in eine benutzerfreundliche Darstellung transformiert werden. Diese Transformation wird in der Literatur häufig als **"Pretty-printing"** (z.B. /Go 73/, /Op 80/) oder auch als **"Unparsing"** (z.B. /Me 82/) bezeichnet.

Die Vorgehensweise der Transformation der internen Repräsentation in eine für den Benutzer gewohnte, textuelle Repräsentation wird von den meisten Software-Entwicklungsumgebungsprojekten angewandt (z.B. Gandalf /Me 82/, Mentor /DH 84/, Cornell Program Synthesizer /TR 81/). Die im PECAN-Projekt (/Re 84/) verfolgte Vorgehensweise der parallelen Verwaltung verschiedener Darstellungen eines Programms (interne Darstellung, textuelle Darstellung, Flußdiagramm, Nassi-Shneiderman-Diagramm etc.) impliziert eine Reihe unnötiger Konsistenzprobleme zwischen den verschiedenen Darstellungen. Durch den Anschluß unterschiedlicher Transformationsprogramme, auch **"Unparser"** genannt, können aus der internen abstrakten Darstellung eines Moduls mehrere verschiedene konkrete textuelle oder auch graphische Darstellungen erzeugt werden.
Wir haben uns bisher im Projekt IPSEN auf eine **textuelle Darstellung** eines Modula-2-Moduls beschränkt. Die hierbei auftretenden Probleme wurden intensiv in der Diplomarbeit /Ha 85/ untersucht; einige der dort erzielten Ergebnisse fließen in die Darstellung dieses Kapitels ein.

Die interne Darstellung eines Programms wird in erster Linie durch Aktionen des syntaxgestützten Editors modifiziert. Bei Eingabe eines Editorkommandos durch den IPSEN-Benutzer werden im Modulgraphen entsprechende Graphersetzungen durchgeführt. Neben dieser kommandogesteuerten Vorgehensweise hat der IPSEN-Benutzer aus Komfortgründen auch die Möglichkeit (vgl. Paragraph 2.1), Teile der textuellen Repräsentation in der sogenannten "freien Eingabe" zu modifizieren. In dieser Situation steht dem IPSEN-Benutzer ein üblicher zeilenorientierter Texteditor zur Verfügung, mit dem er die textuelle Repräsentation des aktuellen Inkrements beliebig verändern kann. Nach Abschluß einer derartigen freien Eingabe wird dieses veränderte Quelltextstück vom System analysiert und durch eine implizit aktivierte Folge üblicher Editorkommandos in den Modulgraphen eingetragen. Dies bewirkt, daß anschließend die für die Ausführung weiterer IPSEN-Kommandos nötige Modulgraph-

darstellung des Moduls zur Verfügung steht. Diese Aufgabe übernimmt ein **multiple-entry parser**, der im Rahmen der Diplomarbeit /Sl 86/ ausführlich untersucht und implementiert wird. Nähere Informationen zum Parser befinden sich sowohl in der zitierten Diplomarbeit, als auch in zusammengefaßter Form in /Sc 86/.

Die beiden Transformationen zwischen **interner, abstrakter** und **textueller, konkreter** Repräsentation eines Modula-2-Moduls können schematisch wie folgt dargestellt werden:

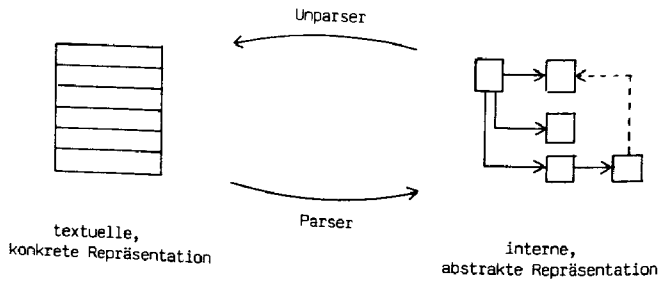


Abb. 6.1: Transformation zwischen verschiedenen Darstellungen eines Modula-2-Moduls

6.1 Aufgaben des Unparsers

Bevor wir die Anforderungen an den in IPSEN benötigten Unparser näher erläutern, wollen wir kurz zusammenfassen, welche Informationen in der **konkreten Repräsentation** eines in IPSEN bearbeiteten Modula-2-Moduls enthalten sein können.

Während in der Modulgraphdarstellung nur die abstrakte Syntax eines Moduls zusammen mit den vom Benutzer gewählten Bezeichnern dargestellt wird, enthält die konkrete Repräsentation zusätzlich den zur Programmiersprache gehörenden **"syntactic sugar"**, d.h. reservierte Worte und Begrenzer. Hinzu kommen **Darstellungsattribute**, die z.B. festlegen, welcher Schrifttyp für die Ausgabe zu wählen ist und welche physikalischen Darstellungsattribute (z.B. unterstrichen, fett, invers, blinkend, ...) benutzt werden sollen. Außerdem liegt der konkreten Repräsentation stets eine Zeilen-/Spaltenstruktur zugrunde, die eine geeignete **Formatierung** der konkreten Repräsentation ermöglicht. Aufgrund der inkrementorientierten Arbeitsweise aller Werkzeuge ist es in IPSEN besonders wichtig, daß die dem aktuellen Programm zugrundeliegende Inkrementstruktur in geeigneter Weise dem Benutzer in der konkreten Repräsentation deutlich gemacht wird. Dies geschieht bei Programmtexten üblicherweise durch das Verteilen des Textes auf unterschiedliche Zeilen und

durch geeignetes Einrücken ("indentation") der textuellen Darstellung innerer Inkremente. Abb. 6.1.1 gibt ein kurzes Beispiel eines Ausschnitts aus einem geeignet formatierten Deklarationsteil:

```
VAR A : RECORD
          B : REAL;
          C : CARDINAL
        END;
    D, E : CARDINAL;
```

Abb. 6.1.1: Ausschnitt aus einem formatierten Deklarationsteil

Neben den zur Programmiersprache Modula-2 gehörenden Sprachkonstrukten kann ein in IPSEN bearbeiteter Modul noch die folgenden Informationen enthalten:

- Da der aktuell bearbeitete Modul noch unvollständig sein kann, werden dem Benutzer die noch offenen Stellen geeignet angezeigt. Bei den an diesen Stellen noch nicht expandierten Inkrementen wird unterschieden zwischen **obligaten** und **optionalen** Inkrementen. Obligate Inkremente müssen vom Benutzer auf jeden Fall noch expandiert werden, damit ein korrekter Modula-2-Modul entsteht. Optionale Inkremente hingegen müssen nicht unbedingt expandiert werden. Die Darstellung einer noch offenen Stelle geschieht durch Ausgabe einer entsprechenden Kennzeichnung des noch nicht expandierten Inkrements (z.B. entsprechende Knotenmarkierung im Modulgraphen), die durch (< ... >) geklammert dargestellt wird (vgl. Paragraph 2.1 und /Sc 86/).
- Die vom Benutzer durch Kommandos des Werkzeugs Testvorbereitung (vgl. Paragraphen 2.3 und 2.4) eingefügten Ergänzungen des Programms werden durch (#...#) geklammerte Zeilen zusätzlich in der konkreten Repräsentation dargestellt.
- Da die Bildschirmfläche und das auf ihm befindliche Ausgabefenster in der Regel nicht ausreicht, um die gesamte konkrete Repräsentation eines Modulgraphen ausgeben zu können, wird in mehreren Projekten die Idee verfolgt, für den Benutzer derzeit irrelevante Teil der konkreten Repräsentation auszublenden und durch ein entsprechendes **Platzhaltersymbol** zu ersetzen. Dieses Vorgehen wird in der Literatur sehr unterschiedlich bezeichnet, z.B. als "holophrasting" in /Ha 71/, /DH 84/ und /Al 83/, als "ellipsis" in /Mi 81/ und /TR 81/, bzw. als "folding" in /Ma 84/. Bei der Angabe, welche Teile der konkreten Repräsentation ausgeblendet werden sollen, arbeiten die meisten Systeme interaktiv, d.h. der Benutzer muß explizit angeben, welche Inkremente aus- bzw. wieder eingeblendet werden sollen. Im Gegensatz dazu wird in /Mi 81/ vorgeschlagen, dem Benutzer verschiedene Modi anzubieten (Edier-, Lese-, "multiple-focus"-Modus), bei dem dann vom Unparser selbst entschieden wird, welche Teile auszublenden sind. Auch in IPSEN ist in der Prototypversion bisher nur vorgesehen, daß der Benutzer die konkrete Repräsentation einzelner Inkremente explizit aus- bzw. wieder einblendet (vgl. /Sc 86/). In einer späteren Version von IPSEN sollte jedoch auch der in /Mi 81/

vorgeschlagene multiple-focus-Modus angeboten werden, um textuell weit entfernt liegende Inkremente durch Ausblenden der dazwischenliegenden Textstücke in einer begrenzten Bildschirmfläche anzeigen zu können.

Damit sind die **Aufgaben des Unparsers** in IPSEN zum größten Teil schon erläutert: Zum aktuellen Inkrement, festgelegt durch die aktuelle Cursorposition im Modulgraphen, muß die zugehörige konkrete Repräsentation erzeugt werden. Bei der Formatierung muß neben der Betonung der logischen Struktur durch entsprechendes Aufteilen des Quelltextes auf verschiedene Zeilen und Einrücken von Quelltext die durch das Ausgabemedium vorgegebene Größe (z.B. Höhe und Breite eines Fensters) beachtet werden. Um zur Darstellung des Kontexts des aktuellen Inkrements bzw. zum Füllen eines Fensters genügend Quelltext zur Verfügung zu haben, muß in einem begrenzten Umfang auch zu Bruder- und Vaterinkrementen konkrete Repräsentation erzeugt werden.

Der **Umfang** der erzeugten konkreten Repräsentation wird dabei durch verschiedene Faktoren bestimmt:

- Zu allen expandierten und obligaten Inkrementen wird stets Quelltext erzeugt. Der Benutzer kann festlegen, welche offenen Stellen optionaler Inkremente ebenfalls im Quelltext dargestellt werden sollen.
- Weiterhin kann der Benutzer festlegen, ob die für Testzwecke im Modulgraphen zusätzlich eingetragenen Informationen im Quelltext dargestellt werden sollen oder nicht.
- Der Benutzer kann die textuelle Darstellung beliebiger Inkremente ausblenden. Diese Stelle wird dann durch ein entsprechendes Platzhaltersymbol gekennzeichnet.

Durch die Einführung einer zweiten Darstellung für einen Teil des aktuell bearbeiteten Moduls stellt sich die Frage, welche **Beziehungen zwischen diesen beiden Repräsentationen** für spätere Aufgaben aufgehoben werden müssen. Wie im Paragraphen 2.1 bereits kurz erwähnt, muß bei Aktivierung der freien Eingabe bekannt sein, in welchem Bereich der konkreten Repräsentation sich die Darstellung des aktuellen Inkrements befindet, damit der Benutzer nur diesen Teil modifizieren kann. Diese Beziehung zwischen einer Position in der konkreten Repräsentation und dem zugehörigen Inkrement, d.h. Knoten, im Modulgraphen, muß darüberhinaus für jede beliebige Position aufgehoben werden. Durch die Benutzung einer Maus hat der Benutzer nämlich die Möglichkeit, in der konkreten Repräsentation am Bildschirm ein neues Inkrement zu selektieren (vgl. /Sc 86/). Das bedeutet, daß zu jeder "angeklickten" Position im Quelltext ein zugehöriger Modulgraphknoten ermittelbar sein muß.

6.2 Tabellengesteuerte Realisierung des Unparsers

Da in den meisten Programmierumgebungen das aktuell bearbeitete Programm intern durch eine abstrakte Repräsentation dargestellt wird, ist die Existenz eines Unparsers nichts Neues bei der Darstellung einer Programmierumgebung. Ziel dieses Paragraphen ist deshalb in erster Linie zu erläutern, inwiefern bei der Realisierung eines solchen Unparsers auf **Adaptabilitätsüberlegungen** geachtet werden kann. Es wird eine ebenfalls graphartige Datenstruktur zur Abspeicherung der konkreten, textuellen Repräsentation vorgestellt, die es u.a. ermöglicht, die textuelle Repräsentation eines Modulgraphausschnitts schrittweise zu erzeugen bzw. zu erweitern. Da die hier vorgestellte Realisierung des Unparsers bisher noch nicht in die Prototypimplementierung des IPSEN-Systems integriert worden ist, erläutern wir im nächsten Kapitel kurz die Realisierung des derzeit im IPSEN-System eingesetzten Unparsers.

Unparser bzw. Pretty-printer wurden in der Literatur zunächst vor allem im Zusammenhang mit **LISP-Umgebungen** vorgestellt (z.B. /Go 73/). Die dort intern manipulierten Bäume bzw. Listen mußten durch ein Transformationsprogramm in eine textuelle Darstellung übersetzt werden. Auch in IPSEN ist in erster Linie der dem Modulgraphen zugrundeliegende **abstrakte Syntaxbaum** bzw. **-graph** ausschlaggebend für eine Erzeugung der zugehörigen konkreten Repräsentation. Um zu einem derartigen Baum die zugehörige textuelle Darstellung zu erzeugen, wird durch einen geeigneten **Traversierungsalgorithmus** jeder Knoten des Baums besucht und entsprechender Quelltext zu diesem Knoten erzeugt.

Da wir in IPSEN nicht mit geordneten Bäumen arbeiten, sondern bei Struktur-Graphinkrementen die Söhne eines Knotens nur durch unterschiedlich markierte Kanten mit dem Vater verbinden bzw. bei Listengraphinkrementen eine explizite Ordnungskante zwischen zwei Söhnen eintragen, ist bei einer Traversierung des dem Modulgraphen zugrundeliegenden abstrakten Syntaxgraphen festzulegen, in welcher Reihenfolge die Söhne besucht werden sollen. Weiterhin muß festgelegt werden, welche reservierten Wörter, Begrenzer, Attributwerte an einem Knoten in der konkreten Repräsentation zu erzeugen sind, welche Darstellungsangaben zu beachten sind und wie der Quelltext zu formatieren ist.

Für alle diese Informationen ist der im Kapitel 4 unter Frage 1) erläuterte Trade-off zwischen Algorithmus und Datenstruktur abzuwägen. Verkapselt man alle diese Informationen "hart" in einem Algorithmus bedeutet z.B. bereits der Wunsch nach einer geänderten Formatierung eine Modifikation und Recompilation des Algorithmus. Leichte Anpaßbarkeit an geänderte Anforderungen wird erreicht, wenn man große Teile der benötigten Informationen in separaten Datenstrukturen, d.h. Tabellen, ablegt. Derartige Tabellen können dann in externen Dateien abgespeichert sein, so daß bereits durch Austausch einer Datei die Anpassung an geänderte

Anforderungen realisiert werden kann. Diese Vorgehensweise ist z.B. im Gandalf-Projekt (/Me 82/) oder DICE-Projekt (/Fr 83/) verfolgt worden. In beiden Projekten wird eine einfache formale Sprache eingeführt, um zu jedem Inkrementtyp des abstrakten Syntaxbaumes ein sogenanntes **Unparsing-Schema** anzugeben. Durch diese Unparsing-Schemata werden dann alle oben erwähnten Informationen zu einem Knoten festgelegt.

Da ein wesentliches Charakteristikum von IPSEN die Adaptabilität an geänderte Anforderungen ist, wird der Unparser im IPSEN-System ebenfalls als **tabellengesteuertes Transformationsprogramm** realisiert. Die dazu benötigte, im IPSEN-System eingesetzte formale Sprache zur Festlegung der Unparsing-Schemata erläutern wir im übernächsten Abschnitt 6.2.2. Zunächst wollen wir die Datenstruktur zur Darstellung der erzeugten konkreten Repräsentation näher erläutern.

6.2.1 Der Textgraph

Es hat sich herausgestellt, daß auch für die Darstellung der konkreten Repräsentation eine graphartige Datenstruktur am geeignetesten ist, um die während der Erzeugung der konkreten Repräsentation durchzuführenden Modifikationen effizient realisieren zu können. Wir werden in diesem Abschnitt diese Datenstruktur sukzessive entwickeln und an Hand von Beispielen für derartige Modifikationen den Einsatz dieser Datenstruktur weiter motivieren.

Betrachtet man einmal die textuelle, formatierte Repräsentation eines Modulgraphausschnitts etwas genauer, erkennt man, daß der textuellen Darstellung sowohl eine physische als auch eine logische Struktur zugrundeliegt.

Die **physische Struktur** ist dabei durch die zu einer textuellen Darstellung gehörende Zeilenstruktur gegeben. Weiterhin sind bei einem Programmiersprachentext auch die innerhalb einer Zeile auftretenden Zeichenfolgen beschränkt auf die zur Programmiersprache gehörenden reservierten Wörter, Begrenzer, vom Benutzer definierte Bezeichner oder Folgen von Leerzeichen (blanks). Bei einem in IPSEN bearbeiteten Programm kommen außerdem Platzhaltersymbole für noch fehlende bzw. ausgeblendete Programmteile hinzu. (Auf die Darstellung von Zusatzinformationen und Kommentaren gehen wir an dieser Stelle nicht ein. Eine entsprechende Erweiterung der Datenstruktur ist jedoch problemlos.) Eine denkbare Datenstruktur zur Darstellung der physischen Struktur eines Quelltextausschnitts ist somit eine Folge von Zeilen, wobei jede Zeile aus einer Folge von Textstücken besteht. Analog zur Vorgehensweise bei einem Modulgraphen kann man diese verschiedenen Arten von Textstücken auch durch Knoten unterschiedlicher Markierung darstellen und die konkreten Textstücke in Attributen ablegen. Die in Abb. 6.1.1 dargestellten Variablendeklarationen haben in einer solchen baumartigen Darstellung dann das

folgende Aussehen:

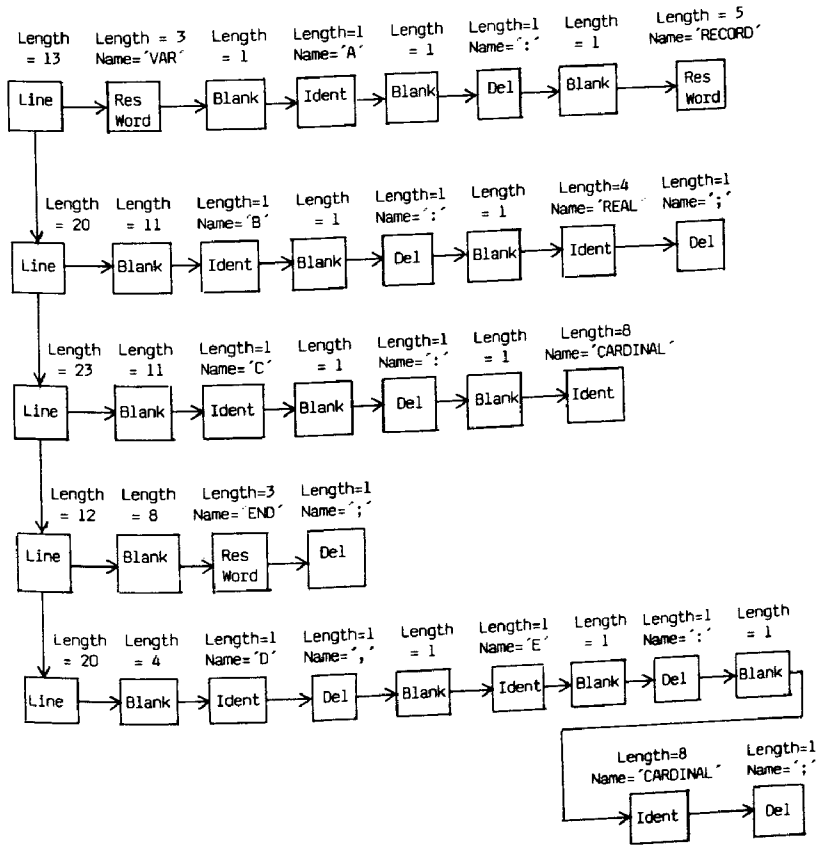


Abb. 6.2.1.1: Darstellung der physischen Struktur formatierter Variablendeklarationen

Die hierbei auftretenden Knotenmarkierungen haben folgende Bedeutung:

- Line kennzeichnet den Beginn einer neuen Zeile,
- ResWord kennzeichnet ein reserviertes Wort,
- Del kennzeichnet einen Begrenzer (Delimiter),
- Ident kennzeichnet einen Bezeichner,
- Blank kennzeichnet eine Folge von Leerstellen (blanks).

Jeder Knoten trägt das Attribut "Length", das die Länge der im Attribut "Name" abgelegten Zeichenfolge angibt (bei ResWord, Del, Ident), die Anzahl von Leerstellen angibt (bei Blank) bzw. die aktuelle Zeilenlänge angibt (bei Line).

Durch die Verwendung **unterschiedlicher** Knotenmarkierungen erspart man sich das Ablegen zusätzlicher Darstellungsangaben für einzelne Arten von Textstücken. Z.B. kann eine besondere Darstellung von reservierten Wörtern (z.B. fett) nun in einer globalen Tabelle vermerkt werden und muß nicht an jedem entsprechenden Knoten abgelegt werden.

Durch die Aufteilung des Quelltextes auf verschiedene Zeilen sowie durch Einrücken bestimmter Quelltextstücke versucht man, die einer textuellen Repräsentation zugrundeliegende **logische Struktur** sichtbar zu machen. Dieses Hervorheben der logischen Struktur ist in einer inkrementorientierten Programmierumgebung wie IPSEN besonders wichtig und muß deshalb vom Unparser zufriedenstellend realisiert werden. /Mi 81/ weist darauf hin, daß die logische Struktur besonders deutlich wird, wenn man um jede Darstellung einer logischen Einheit einen **rechteckigen Rahmen** zeichnet. Man erkennt man, daß die logische Struktur in der textuellen Darstellung durch eine horizontale bzw. vertikale Reihung und Ineinanderschachtelung derartiger rechteckiger Rahmen ausgedrückt wird. Als Beispiel betrachte man noch einmal die Variablendeklarationen aus Abb. 6.1.1:

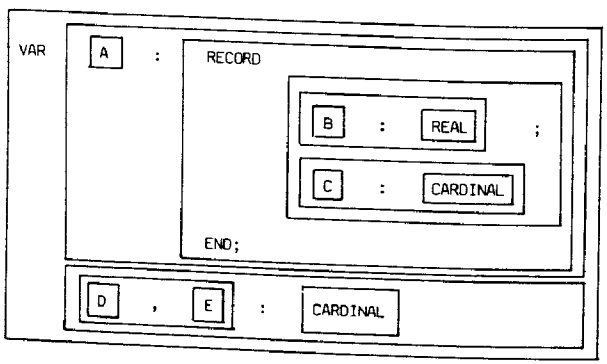


Abb. 6.2.1.2: Hervorhebung der logischen Struktur durch rechteckige Rahmen

Es ist naheliegend, zur Darstellung der **Ineinanderschachtelung** bzw. **Reihung von Rahmen** einen **Baum** als Datenstruktur zu wählen. Dieser hat zum obigen Beispiel dann die in Abb. 6.2.1.3 angegebene Gestalt.
Jeder Knoten in dieser Darstellung steht für einen rechteckigen Rahmen. Die Kantenmarkierung "in" kennzeichnet, daß der zum Zielknoten dieser Kante gehörende Rahmen innerhalb des zum Quellknoten gehörenden Rahmens liegt. Entsprechend kennzeichnet die Kantenmarkierung "next", daß die beiden zugehörigen Rahmen horizontal oder vertikal parallel liegen.

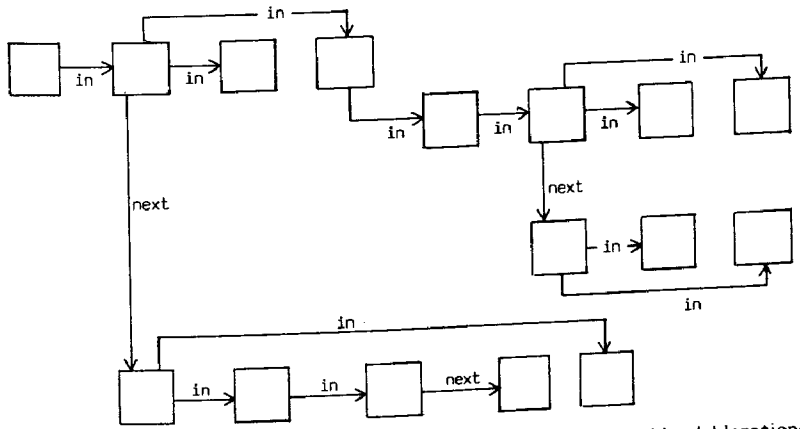


Abb. 6.2.1.3: Darstellung der logischen Struktur formatierter Variablendeklarationen

Die durch die Rahmen ausgedrückte logische Struktur steht natürlich in direktem **Zusammenhang** zu dem im Modulgraphen enthaltenen abstrakten Syntaxgraphen. Das bedeutet, daß bei jedem Knoten in einem Attribut "Node" die Knotennummer des zugehörigen Knotens im Modulgraphen abgelegt werden kann. Dies ist wichtig für die Ermittlung des zugehörigen Inkrements nach einem Mausklick durch den Benutzer an einer beliebigen Position in der konkreten Repräsentation.

Bei der Festlegung einer Datenstruktur für die konkrete Repräsentation wäre es nun denkbar, einen der beiden oben vorgestellten Bäume als Grundlage zu nehmen und in zusätzlichen Attributen die in dem anderen Baum verkapselte Information abzulegen. Die Bevorzugung einer Struktur müßte jedoch rein willkürlich geschehen, so daß wir uns dazu entschlossen haben, durch **Vereinigung** der beiden obigen **Bäume** eine graphartige Datenstruktur für die konkrete Repräsentation zu wählen. Diese Datenstruktur wollen wir im folgenden **Textgraph** nennen (vgl. hierzu auch /Ha 85/). Wir verzichten an dieser Stelle auf eine formale Definition (z.B. durch Angabe eines Graph-Ersetzungssystems) und erläutern die Gestalt der Datenstruktur an dem folgenden, im Vergleich zu Abb. 6.1.1 etwas abgemagerten Beispiel von Variablendeklarationen, bei denen eine Typdefinition vom Benutzer noch nicht eingegeben wurde bzw. zur Zeit ausgeblendet worden ist:

```
VAR A : (<Type Definition>);
    D, E : CARDINAL;
```

Abb. 6.2.1.4: Beispiel formatierter Variablendeklarationen

Der zugehörige Textgraph hat das in Abbildung 6.2.1.5 dargestellte Aussehen.

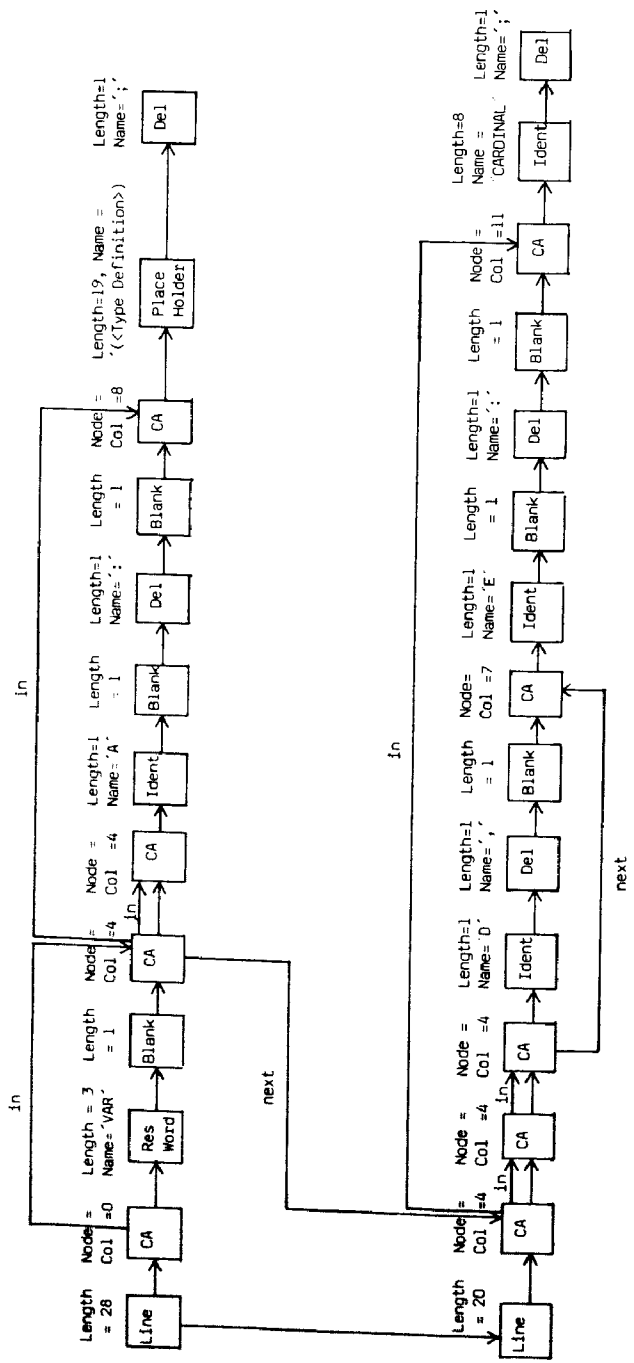


Abb. 6.2.1.5: Beispiel für einen Textgraphen

Die Knoten zur Darstellung der logischen Struktur sind mit "CA" (Abk. für Column Alignment) markiert. Das Attribut "Col" (Abk. für Column) gibt an, in welcher Spalte der aktuellen Zeile der zugehörige Rahmen beginnt. Der Knoten zur Darstellung der noch fehlenden bzw. ausgeblendeten Typdefinition ist mit "PlaceHolder" markiert.

Der **Vorteil** dieser auf den ersten Blick recht aufwendig erscheinenden Datenstruktur sei an den folgenden typischen Anwendungen erläutert:

- Da im voraus nicht feststellbar ist, wie umfangreich die konkrete Repräsentation zu einem Modulgraphausschnitt ist, wird in den meisten Projekten zur gesamten abstrakten Repräsentation Quelltext erzeugt, um ausreichend Quelltext ausgeben zu können (vgl. z.B. /Mi 81/). Durch die Textgraphdarstellung ist es leicht möglich, **Schritt für Schritt** mehr Quelltext zu erzeugen, wenn die erwünschte Größe des zu erzeugenden Quelltexts noch nicht erreicht ist. Hierzu wird sukzessive im Modulgraphen zum Vater des in (Zeile 1, Spalte 1)-stehenden Knotens aufgestiegen, zu diesem Knoten Quelltext erzeugt und dabei der bereits erzeugte Quelltext als Teilgraph in den neuen Textgraphen eingehängt.
- Bei diesem Einhängen eines Textgraphen in einen übergeordneten Textgraphen, aber auch beim Erzeugen von Quelltext kann es passieren, daß die aktuelle Zeilenlänge die erlaubte **Zeilenlänge überschreitet.** In diesem Fall wird hinter der aktuellen Zeile eine weitere Zeile eingefügt und der Rest der Zeile bis zum letzten CA-Knoten gemäß dem über die einlaufende in-Kante erreichbaren CA-Knoten in dieser neuen Zeile spaltenausgerichtet abgelegt. Dieser Vorgang ist u.U. zu iterieren. Im obigen Beispiel würde dies z.B. bei einer maximalen Zeilenlänge von 25 Zeichen bedeuten, daß zwischen der ersten und zweiten Zeile eine weitere Zeile eingefügt wird und der Quelltext wie folgt formatiert wird:

```
VAR A :
    (<TypeDefinition>);
    D, E : CARDINAL;
```

Abb. 6.2.1.6: Beispiel formatierter Variablendeklarationen bei begrenzter Zeilenlänge

(In /Ha 85/) wird zusätzlich die Möglichkeit diskutiert, bei solchen Zeilenumbrüchen unterschiedliche Prioritäten zu beachten.)

- Das Vorhandensein einer Zeilenstruktur ermöglicht eine **schnelle Ausgabe** einzelner (Bildschirm-)Zeilen.
- Wird vom Benutzer mit Hilfe der **Maus** eine beliebige Position angeklickt, kann mittels einer sequentiellen Suche der zugehörige Knoten im Textgraphen ermittelt werden. Der am nächsten links davon stehende CA-Knoten enthält als Attribut die zugehörige Knotennummer im Modulgraphen. Nach einer derartigen Auswahl eines neuen aktuellen Inkrements muß auf dem Bildschirm die Darstellung dieses Inkrements entsprechend gekennzeichnet werden (z.B. durch einen Flächencursor).

Alle von diesem CA-Knoten über in-Kanten erreichbaren Zeilen und Spalten gehören zum aktuellen Inkrement und können auf dem Bildschirm entsprechend dargestellt werden. Das bedeutet, daß bei **Cursorbewegungen** im bereits erzeugten Quelltext der Unparser nicht erneut aktiviert werden muß.

- Dasselbe gilt für das **Aus- bzw. Einblenden** von Quelltextstücken. Beim Ausblenden eines Inkrements wird der zugehörige Teilgraph im Textgraph durch einen einzelnen Platzhalterknoten ersetzt. Beim Einblenden eines Inkrements wird der Textgraph an dem Platzhalterknoten wieder aufgebläht.

In der Diplomarbeit /Ha 85/ wird konkret erläutert, welche Probleme bei derartigen Textgraphmanipulationen auftauchen. Dort wird auch eine Implementierung dieser Datenstruktur vorgestellt.

Die oben erläuterten Anwendungen zeigen, daß ein erzeugter Textgraph für spätere Unparsing-Aufrufe wiederbenutzt werden kann, wenn zwischenzeitlich der zugehörige Modulgraphanteil nicht verändert worden ist. Dies impliziert einen effizienteren Unparsingprozeß, als wenn bei jedem Unparsingaufruf die textuelle Darstellung vollkommen neu erzeugt werden müßte. Gemäß der im Kapitel 4 diskutierten Klassifikationsmerkmale werden hier also Teile der für spätere Unparsingaufrufe benötigten Informationen in einer Datenstruktur, hier Textgraph, abgelegt. In diesem Sinne stellt dieser Unparser einen **inkrementellen Unparser** dar.

Durch die bisher vorgestellte, baumartige Darstellung der logischen Struktur ist es nicht möglich, **kontextabhängige Beziehungen** zwischen den einzelnen Rahmen zu beschreiben. Dies heißt konkret, daß es in dem in Abb. 6.2.1.4 angegebenen Beispiel zwar möglich ist, dafür zu sorgen, daß die beiden Variablendeklarationen spaltenausgerichtet dargestellt werden. Es ist jedoch bisher nicht beschreibbar, daß auch die beiden Typdefinitionen spaltenausgerichtet dargestellt werden, wie es in der folgenden Darstellung der Fall ist:

```
VAR A :    (<TypeDefinition>);
    D, E : CARDINAL;
```

Abb. 6.2.1.7: Beispiel einer kontextabhängigen Spaltenausrichtung

Durch die Verwendung einer **graphartigen** Darstellung ist es möglich, auch derartige kontextabhängige Spaltenausrichtungen im Textgraphen vorzusehen. Hierzu wird eine zusätzliche Kante zwischen den beiden zugehörigen CA-Knoten gezogen. Es ist jedoch zu bedenken, daß dieser zusätzliche Komfort weitere Konfliktfälle bei der Formatierung von Quelltext impliziert. Um den Unparsing-Vorgang deshalb nicht zu sehr zu verlangsamen, sollten derartige Spaltenausrichtungen als erstes nicht beachtet werden, wenn der erzeugte Quelltext nicht in die gewünschte Zeilenbreite paßt.

6.2.2 Unparsing-Schemata

Die zum Aufbau des Textgraphen und Erzeugen der konkreten Repräsentation zusätzlich zum abstrakten Syntaxgraphen benötigte Information wird in einer Tabelle mit Unparsing-Schemata abgelegt. Diese Unparsing-Schemata legen für jedes Graphinkrement fest, welche konkrete Syntax auszugeben ist, welche Attributwerte bei atomaren Graphinkrementen auszugeben sind, an welcher Stelle die konkrete Repräsentation von Söhnen einzufügen ist und wie die gesamte konkrete Repräsentation zu formatieren ist.

Ein **Unparsing-Schema** ist eine Folge von **Unparsing-Primitiven,** die bei der Erzeugung der konkreten Repräsentation von einem im Unparser enthaltenen Unparsing-Schema-Interpreter von links nach rechts interpretiert wird. Jede Interpretation eines Unparsing-Primitivs bewirkt eine entsprechende Modifikation des Textgraphen. Unparsing-Schemata werden u.a. im Gandalf-Projekt (/Me 82/) und im DICE-Projekt (/Fr 83/) eingesetzt. Die dort eingeführten Sprachen für Unparsing-Schemata dienten als Grundlage für die in /Ha 85/ erläuterte Unparsing-Schema-Sprache. Wir stellen hier exemplarisch einige Unparsing-Primitive vor:

"ResWord" oder $string$
: das zwischen den beiden "-Symbolen stehende reservierte Wort bzw. der zwischen den beiden $-Symbolen stehende string wird an der aktuellen Position im Textgraphen ausgegeben

'symbol'
: das zwischen den beiden '-Symbolen stehende symbol ist ein Begrenzer und wird an der aktuellen Position im Textgraphen ausgegeben

(<string>)
: diese Zeichenfolge kennzeichnet ein Platzhaltersymbol und wird in derselben Form an der aktuellen Position im Textgraphen ausgegeben

?(<string>)?
: die am aktuellen Modulgraphknoten im Attribut "Name" abgelegte Zeichenfolge wird ausgegeben; falls das Attribut leer ist, wird stattdessen das zwischen den ?-Symbolen stehende Platzhaltersymbol ausgegeben

_
: an der aktuellen Position im Textgraphen wird ein Blank ausgegeben.

Im Gegensatz zu abstrakten Syntaxbäumen (vgl. /Me 82/, /Fr 83/) ist im IPSEN-System die Reihenfolge, in der zu Söhnen Quelltext zu erzeugen ist, nicht durch die vorliegende Ordnung festgelegt. Dies bedeutet, daß in einem Unparsing-Schema festgelegt werden kann, in welcher Reihenfolge der Quelltext zu den einzelnen Söhnen zu erzeugen ist. Die Position, an der der Quelltext zu einem Sohn zu erzeugen ist, geschieht im Unparsing-Schema durch Angabe der Kantenmarkierung zu diesem Sohn:

<EdgeLabel>

an der aktuellen Position im Textgraph ist der Quelltext zu dem über die Kante mit der Markierung EdgeLabel im Modulgraphen erreichbaren Knoten einzutragen.

Da bei optionalen Teilen im Modulgraphen eine Kante mit dieser Markierung nicht unbedingt existiert, dient das folgende Unparsing-Primitiv dazu zu beschreiben, welche konkrete Repräsentation nur bei Existenz dieser Kante zu erzeugen ist:

[<EdgeLabel>/pre-list/post-list]

nur bei Existenz der mit EdgeLabel markierten Kante im Modulgraphen wird zunächst die in pre-list stehende Folge von Unparsing-Primitiven ausgewertet, dann die konkrete Repräsentation zu dem über die mit EdgeLabel markierte Kante erreichbaren Knoten ausgegeben und anschließend die in post-list stehende Folge von Unparsing-Primitiven ausgewertet.

Bei Listengraphinkrementen ist im abstrakten Syntaxgraph die Anzahl der Listenelemente nicht beschränkt. Das folgende Unparsing-Primitiv ist deshalb entsprechend der Anzahl der Listenelemente wiederholt auszuwerten:

{ UP-List/ (<string>)}

da im Modulgraphen die Kanten zum Durchlaufen einer Liste stets gleich markiert sind (EFirst, ENext), wird in diesem Unparsing-Primitiv keine Kantenmarkierung benötigt. UP-List gibt eine Folge von Unparsing-Primitiven an, die zwischen zwei Listenelementen zu interpretieren sind. Falls kein Listenelement vorhanden ist, wird das Platzhaltersymbol (<string>) an der aktuellen Position im Textgraphen ausgegeben.

Mit Hilfe dieser Unparsing-Primitive können als Beispiel bereits die Unparsing-Schemata für die drei Graphinkremente Variablendeklaration, Identifierliste und Identifier angegeben werden:

```
Variablendeklaration:   <EIdentList> _ ':' _ <ETypeDef>
Identifierliste:        { ',' _ / (<IdentList>) }
Identifier:             ? (<Ident>) ?
```

Abb. 6.2.2.1: Beispiele für Unparsing-Schemata

Bei der Interpretation der Unparsing-Primitive wird bei jedem Übergang zu einem neuen Unparsing-Schema, d.h. bei jedem Übergang zu einem neuen Knoten im Modulgraphen, ein neuer Rahmen im Textgraphen eröffnet. Soll ein solcher Rahmen über mehrere Zeilen gehen, wird hierzu das folgende Unparsing-Primitiv benötigt:

NL

(Abk. für New Line) an der aktuellen Position im Textgraphen wird ein Zeilenumbruch durchgeführt. Die neue Startspalte in der nächsten Zeile

entspricht der Position des linken Randes des zu diesem Unparsing-Schema gehörenden Rahmens.

Damit kann z.B. ein Unparsing-Schema für eine Record-Typdefinition angegeben werden:

```
"RECORD" NL _ _ _ <EFieldCompList> NL "END"
```

Abb. 6.2.2.2: Unparsing-Schema für eine Record-Typdefinition

Durch dieses Unparsing-Schema wird beschrieben, daß die Record-Komponentenliste in der Zeile unterhalb des reservierten Wortes RECORD beginnt und um 3 Stellen eingerückt ist. Das reservierte Wort END steht ebenfalls in einer eigenen Zeile in derselben Spalte wie das zugehörige RECORD.

In der Diplomarbeit /Ha 85/ werden weitere Unparsing-Primitive zum Formatieren von Quelltext erläutert, auf die wir hier nicht näher eingehen wollen.

Die Einführung von Unparsing-Schemata ermöglicht es, relativ problemlos die Art und Weise der textuellen Repräsentation des Modulgraphen zu modifizieren. Prinzipiell ist dies sowohl für den IPSEN-Benutzer als auch für den IPSEN-Implementierer denkbar. Für die **Modifikation der Unparsing-Schemata** könnte hierzu ebenfalls ein syntaxgestützter Editor vorgesehen werden, der entweder nur vom IPSEN-Implementierer in der Implementierungs- bzw. Testphase für die Eingabe der Unparsing-Schemata benutzt wird, oder als weiteres Werkzeug dem IPSEN-Benutzer in der Programmierumgebung zur Verfügung steht. Weitere Überlegungen zu diesem Thema findet man u.a. in /Ga 85/.

Da in der Regel die textuelle Darstellung des Modulgraphen sehr eng mit der normierten EBNF als Ausgangsbasis für die Festlegung der Graphinkremente zusammenhängt, können die rechten Seiten der entsprechenden EBNF-Produktionen auch bei der Festlegung der Unparsing-Schemata als Ausgangsbasis benutzt werden. Im wesentlichen sind hierbei nur die in der EBNF-Produktion auftretenden nichtterminalen Symbole durch die entsprechende Kantenmarkierung im Modulgraphen zu ersetzen. Wie bereits im Kapitel 4 bei der Diskussion der Frage 1) erwähnt, kann hier durch Einsatz eines **Generators** das Grundgerüst der Unparsing-Schemata **automatisch erzeugt** werden. Der Benutzer bzw. Implementierer hätte dann nur noch die Aufgabe, die Unparsing-Primitive für die Formatierung des Quelltextes hinzuzufügen. Auf diese Weise erhält man zu jeder Knotenmarkierung des Modulgraphen ein Unparsing-Schema (vgl. /Ha 85/). Neben dieser systematischen Ableitung der Unparsing-Schemata aus der EBNF ist es aber auch möglich, jedes beliebige sinnvolle Unparsing-Schema zu einem Knoten festzulegen. Beispiele hierfür werden in Kapitel 7 bei der Abspeicherung von Zusatzinformationen im Modulgraphen angegeben.

Da zu derselben Knotenmarkierung dann unterschiedliche Unparsing-Schemata gehören können, kann die Zuordnung der Unparsing-Schemata nicht markierungsabhängig sein. Aus diesem Grunde existiert an jedem Knoten im Modulgraphen das Attribut "Scheme", in dem das Unparsing-Schema selbst bzw. ein Verweis auf das zugehörige Unparsing-Schema in einer Tabelle von Unparsing-Schemata abgelegt ist.

Die bisher dargestellten Möglichkeiten zur Erzeugung der konkreten Repräsentation sind stark orientiert an der Graphinkrementstruktur des Modulgraphen. Das wird vor allem daran deutlich, daß bei einer Erweiterung des Textgraphen im Modulgraphen stets zum Vater bez. der Graphinkrementstruktur aufgestiegen wird. Eine denkbare Erweiterung der oben vorgestellten Unparsing-Primitive könnte darin bestehen, daß anstelle der durch die Graphinkrementstruktur implizit gegebenen Vater-/Sohnbeziehungen eine **eigene Rahmenstruktur** für die konkrete Repräsentation festgelegt wird. Eine denkbare Anwendung im Bereich des Programmierens-im-Kleinen könnte sein, in unmittelbarer textueller Nähe zu einer Variablendeklaration alle zugehörigen Typdeklarationen darstellen zu wollen. Diese können über auslaufende "EObject"- und "EType"-Kanten gefunden werden. Die Festlegung einer derartigen Darstellung könnte dadurch möglich werden, daß anstelle einer einzelnen Kantenmarkierung eine Kantenfolge in den Unparsing-Schemata erlaubt ist. Dies hätte dann zur Folge, daß die konkrete Repräsentation weit entfernt liegender Graphinkremente textuell sehr dicht zusammen stehen kann. Weiterhin muß dann jedoch zu jedem Knoten im Modulgraphen festgelegt werden, welcher Knoten der Vaterknoten bez. der Rahmenstruktur ist. Nur dadurch ist es dann weiterhin möglich, den Textgraphen zumindest teilweise von innen nach außen aufzubauen, wenn z.B. festgestellt wird, daß die bereits erzeugte konkrete Repräsentation noch nicht ausreichend groß ist.
Derartige Erweiterungen der Unparsing-Schemata sind bisher noch nicht abschließend untersucht worden. Sie sollten Inhalt weiterer Forschung im Bereich komfortabler, inkrementeller Unparser sein.

6.3 Realisierung des Unparsers durch rekursiven Abstieg

Die Realisierung des in der Diplomarbeit /Ha 85/ dargestellten Unparsers wurde zu einem Zeitpunkt gestartet, als noch viele der für eine Implementierung benötigten Basiskomponenten nicht vorhanden waren. Dies hat zur Folge, daß die derzeit vorliegende Implementierung in Pascal nach Modula-2 portiert werden muß und an die mittlerweile vorhandenen, mit festen Schnittstellen versehenen Basiskomponenten angepaßt werden muß. Da bei einer derartigen Integration eine Reihe von Schwierigkeiten zu erwarten sind, wird im Moment im IPSEN-System eine zweite, weniger Komfort bietende Realisierung des Unparsers eingesetzt. Wir wollen in

diesem Paragraphen kurz das Konzept dieser Realisierung erläutern.

Anstelle eines tabellengesteuerten Verfahrens haben wir uns hierbei für einen "hart verdrahteten" Algorithmus entschieden. Hierbei wird der dem Modulgraph zugrundeliegende abstrakte Syntaxgraph nach dem Prinzip des "rekursiven Abstiegs" durchlaufen. Das heißt, daß für jedes Graphinkrement eine Prozedur existiert, in der festgelegt ist, welche konkrete Syntax zu erzeugen ist, welche Formatierungsangaben zu beachten sind und in welcher Reihenfolge entsprechende Prozeduren für die Erzeugung des Quelltextes der Söhne aufzurufen sind.

Auch die Datenstruktur zur Darstellung des erzeugten Quelltext weicht von dem oben vorgestellten Textgraphen ab. Das heißt, daß wir bei der jetzigen Datenstruktur **Textdokument** die Darstellung der physischen und logischen Struktur getrennt verwalten. Die physische Struktur, d.h. der eigentliche Text, wird in einer Liste von Textzeilen abgelegt. Parallel dazu wird ein Baum zur Darstellung der logischen Struktur verwaltet. Jeder Knoten dieses Baums stellt einen der oben vorgestellten Rahmen dar. Um die Beziehung zwischen der logischen und physischen Struktur verwalten zu können, wird an jedem Knoten der logischen Struktur in einem Attribut Anfangs- und Endposition des zugehörigen Rahmens in der physischen Struktur abgelegt. Diese Angaben bestehen aus Zeilen- und Spaltennummer der linken oberen Ecke bzw. rechten unteren Ecke des rechteckigen Rahmens.

Schließlich muß nach einem Mausklick an einer bestimmten Position im Quelltext auch das zugehörige Inkrement im Modulgraph ermittelbar sein. Diese Beziehung zwischen Textdokument und Modulgraph wird durch ein zusätzliches Attribut an jedem Knoten der logischen Struktur aufgehoben, in dem die zugehörige Nummer des Knotens im Modulgraphen abgelegt wird. Nach einem Mausklick kann dann an Hand der gegebenen Position im Quelltext durch einen **Suchalgorithmus** schnell der kleinste umfassende Rahmen und damit das zugehörige Inkrement ermittelt werden.

Da bei Aufruf des Unparsers in der Regel nicht der gesamte Quelltext, sondern nur eine bestimmte Anzahl von Zeilen erzeugt werden muß, kann der Unparser mit der Erzeugung des Quelltextes zu jedem beliebigen Inkrement gestartet werden. In diesem Sinne stellt der Unparser einen **"multiple-entry Unparser"** dar. Falls der Quelltext zu diesem Inkrement kürzer als eine geforderte Anzahl von Zeilen ist, wird der Unparser sukzessive bei dem Vaterinkrement neu gestartet, bis genügend Quelltext erzeugt worden ist. Analog zu der oben beim Textgraphen vorgestellten Vorgehensweise wird auch hier der bereits erzeugte Quelltext an der entsprechenden Stelle im Textdokument eingehängt. Dies kann effizient realisiert werden, weil sowohl die logische als auch die physische Datenstruktur als verzeigerte Datenstrukturen realisiert sind.

Diese Realisierung des Unparsers hat, wie bereits gesagt, gegenüber der

tabellengesteuerten Vorgehensweise den Nachteil, daß jede Veränderung der gewünschten Darstellung eine Recompilation des Unparser-Moduls bedeutet. Weiterhin existieren in dieser Realisierung auch einige, auf die Dauer nicht tragbare Einschränkungen. So wird z.B. in der jetzigen Realisierung kein impliziter Zeilenumbruch bei Überschreitung der Zeilengrenze durchgeführt. Es wird davon ausgegangen, daß der erzeugte Quelltext nicht breiter als 132 Zeichen ist. Diese Einschränkungen werden behoben sein, wenn der oben beschriebene, tabellengesteuerte Unparser im IPSEN-System integriert ist.

7 Realisierung der Ausführung

Wir haben bisher erläutert, daß der Modulgraph eine geeignete Datenstruktur ist, um die während der Erstellung bzw. Analyse eines Modula-2-Moduls benötigten Informationen effizient ermitteln zu können. Darüberhinaus dient der Modulgraph aber auch als grundlegende Datenstruktur, in der die für die Ausführung und Testunterstützung benötigten Informationen abgelegt sind. Es ist Ziel der nächsten beiden Kapitel, diese weiteren Aufgaben des Modulgraphen ausführlich zu erläutern. In diesem Kapitel erläutern wir zunächst die **Realisierung der Ausführung.** Dabei greifen wir im wesentlichen auf die Ergebnisse der Diplomarbeit /Sa 86/ zurück, die derzeit in unserer Arbeitsgruppe erstellt wird. Inhalt dieser Arbeit ist Entwurf und Implementierung eines Ausführungswerkzeugs für Modulgraphen, das die Integration testunterstützender Werkzeuge ermöglicht.

Für die Realisierung der Ausführung eines Modulgraphen ist der Trade-off zwischen einer interpretativen und compilativen Vorgehensweise zu diskutieren. In den in den letzten Jahren entwickelten Programmierumgebungen sind beide Möglichkeiten realisiert worden. Eine rein **interpretative Vorgehensweise** findet man z.B. im Cornell Program Synthesizer (/TR 81a/, /TR 81b/). In diesem Projekt wird das bearbeitete Programm intern in einer baumartigen, interpretierbaren Datenstruktur gehalten. Für die Verwaltung des Gültigkeits- und Sichtbarkeitsbereichs von Bezeichnern, Typen und Adressen von Variablen wird außerdem permanent eine Symboltabelle verwaltet. Eine derartige interpretative Vorgehensweise ermöglicht insbesondere, noch unvollständige Programme ausführen zu lassen. Schwierigkeiten entstehen allerdings beim Ändern von Deklarationen: Um sicherzustellen, daß das aktuell bearbeitete Programm auch kontextsensitiv korrekt ist, wird in diesem Fall die gesamte Symboltabelle gelöscht, sämtliche kontextsensitiven Überprüfungen erneut durchgeführt und eine neue Symboltabelle aufgebaut (/TR 81a/, S.572).

Eine **compilative Vorgehensweise** findet man z.B. in den Projekten Gandalf (/KE 82/) und DICE (/Fr 84/). In beiden Projekten wird das bearbeitete Programm vor Beginn der Ausführung vollständig in eine maschinennähere Form übersetzt. Da diese Vorgehensweise bei Benutzung eines üblichen Compilers selbst bei kleinen Quelltextmodifikationen eine vollständige Recompilation bedeuten würde, verwendet man in diesen beiden Projekten einen sogenannten **inkrementellen Compiler.** Grundidee hierbei ist, bei einer Veränderung des Quelltexts nur die wirklich betroffenen Inkremente neu zu übersetzen und den restlichen Objektcode unverändert zu lassen. Hierbei beschränkt man sich im Gandalf-Projekt auf Prozeduren als kleinste Recompilationseinheit, während man im Projekt DICE bis auf Anweisungsebene heruntergeht (/Fr 85/).

Im Gegensatz zu einer rein interpretativen Vorgehensweise ist durch diese in eine maschinennähere Form übersetzte Darstellung eines Programms die **Ausführung**

erheblich **effizienter.** Andererseits hat eine rein interpretative Vorgehensweise gegenüber einer rein compilativen Vorgehensweise u.a. die folgenden Vorteile:

- die Modifikation eines Inkrements erfordert keine erneute Compilation des gesamten Moduls,
- eine Änderung der Testumgebung, z.B. Ausführungsart, bedeutet nur das Umsetzen eines Schalters im Interpreter und ebenfalls keine erneute Compilation,
- es ist möglich, nur teilweise fertiggestellte Moduln bereits auszuführen,
- zur Realisierung einer integrierten Arbeitsweise verschiedener Werkzeuge muß nicht jeder, bei einer Unterbrechung der Ausführung mögliche Wechsel zu anderen Werkzeugen bereits im übersetzten Code vorgesehen werden.

Da alle diese Eigenschaften einer interpretativen Vorgehensweise mit den im IPSEN-System gegebenen Anforderungen an ein Ausführungswerkzeug korrespondieren (vgl. Paragraphen 2.3 und 2.4), wurde auch für die Realisierung des Ausführungswerkzeugs im IPSEN-System eine interpretative Vorgehensweise gewählt. Andererseits wollten wir auch die Vorteile einer effizienteren Ausführung bei Vorhandensein eines maschinennäheren Codes ausnutzen, so daß wir uns dazu entschieden haben, bei der Ausführung die Vorteile der beiden unterschiedlichen Vorgehensweisen zu vereinen (vgl. /Na 79/). Dies führte somit zu einer Realisierung, die wir **Hybrid-Interpreter** nennen. Wir erläutern diese Vorgehensweise im nächsten Paragraphen.

7.1 Hybrid-Interpreter

Die Grundidee eines derartigen Hybrid-Interpreters besteht darin, mit Hilfe eines Interpreters den Modulgraphen zu durchlaufen und interpretativ auszuführen. Um diese Ausführung zu beschleunigen, werden zuvor bestimmte Inkrementarten in eine maschinennähere Form übersetzt. Wird während der Interpretation des Modulgraphen ein derartiges Inkrement erreicht, wird anstelle einer direkten Interpretation das zugehörige, in maschinennähere Form übersetzte Codestück ausgeführt. Dies geschieht durch Aktivierung eines zweiten Interpreters, der derartige Codestücke ausführen kann. Aufgrund der Existenz dieser zwei Interpreter, die abwechselnd während der Ausführung aktiv sind und unterschiedliche Darstellungen interpretieren können, nennen wir dieses Werkzeug einen **"Hybrid-Interpreter"**.

Da z.B. bei der Interpretation von Schleifen, rekursiven Prozeduren etc. dieselben Inkremente wiederholt ausgeführt werden, ist es naheliegend, einmal erzeugte, maschinennähere Codestücke für eine zu wiederholende Ausführung aufzuheben. Dies geschieht, wie üblich, durch Ablegen des zugehörigen Codestücks in einem Attribut mit Namen "Code" im Modulgraphen. Aufgrund dieser inkrementellen Verwaltung eines Teils der für die Interpretation benötigten Informationen nennen wir diese

Vorgehensweise auch **inkrementelle Interpretation.**

Die Festlegung der Teile eines Modulgraphen, die in eine maschinennähere Form übersetzt werden sollen, orientiert sich sinnvollerweise an der gegebenen Inkrementstruktur. Derzeit haben wir uns dazu entschieden, alle **Zuweisungsanweisungen, Prozeduraufrufe** und **Ausdrücke** in eine maschinennähere Form zu übersetzen. Das bedeutet dementsprechend, daß alle Kontrollstrukturen im Modulgraphen direkt interpretiert werden. Hierbei sind die oben eingeführten Kontrollflußkanten hilfreich, die es dem Interpreter ermöglichen, direkt das nächste auszuführende Inkrement zu bestimmen.

Wie bereits oben erwähnt, werden während der Ausführung anfallende Daten nicht im Modulgraphen, sondern in einer weiteren Datenstruktur, den **Laufzeitdaten,** abgelegt. Wie bei prozedurorientierten Programmiersprachen üblich, besteht diese Datenstruktur aus dem **Laufzeitheap** für die Verwaltung dynamischer Datenobjekte und aus dem **Laufzeitkeller,** auf dem für jeden Prozeduraufruf ein sogenannter **"Stack-Frame"** abgelegt wird (vgl. /Pe 82/). Weiterhin muß Speicherplatz angelegt werden für alle modullokalen Datenobjekte. Um an dieser Stelle bereits die Integration des Programmieren-im-Kleinen-Anteils mit dem Programmieren-im-Großen vorzusehen, muß beachtet werden, daß während der Ausführung modullokale Daten für mehrere Moduln zu verwalten sind. Aus diesem Grunde wird für **jeden Modul** ein **eigener Datenbereich** angelegt, in dem die Werte solcher modullokalen Datenobjekte abgelegt werden können. Die Anfangsadressen der entsprechenden Bereiche merkt man sich in einer Tabelle. Der für die Ausführung benutzte Datenbereich ist somit wie folgt organisiert:

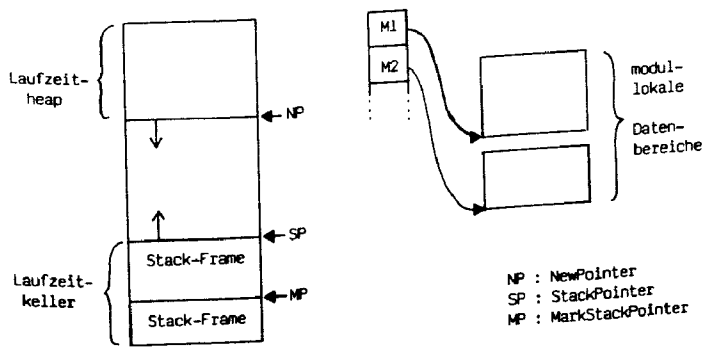


Abb. 7.1.1: Organisation des Laufzeitdatenbereichs

Für die Verwaltung des Laufzeitdatenbereichs werden 3 Zeiger benötigt. Der Zeiger SP (Stack Pointer) gibt die maximale Ausdehnung des Laufzeitkellers an, der Zeiger MP (Mark Stack Pointer) kennzeichnet den Beginn des aktuellen Frames und NP (New

Pointer) gibt die nächste freie Stelle im Laufzeitheap an.

Die obige Erweiterung der üblichen Architektur einer abstrakten Maschine für die Ausführung prozedurorientierter Programmiersprachen entspricht der Architektur der an der ETH Zürich entwickelten Lilith-Maschine (vgl. /Wi 81/, /Ge 83/). Ein Unterschied besteht jedoch darin, daß im IPSEN-System während der Ausführung der auszuführende Code nicht im Speicher dieser abstrakten Ausführungsmaschine steht, sondern im Modulgraphen abgelegt ist.

Bei der Auswahl eines geeigneten maschinennahen Codes für eine derartige Maschine haben wir die bei der Übersetzung von Pascal-/Modula-2-Programmen häufig benutzten Alternativen **M-Code** (vgl. /Wi 81/) und **P-Code** (vgl. /Pe 82/, /Gl 82/) untersucht. Da der speziell für die Lilith-Maschine entwickelte M-Code keine typabhängigen arithmetischen Operationen kennt, müssen sämtliche arithmetischen Operationen auf Wort- bzw. Doppelwortebene durchgeführt werden. Andererseits wird durch diese Abstraktion bis auf Wortebene eine effiziente Speicherausnutzung der Lilith-Maschine ermöglicht. Da es nicht geplant ist, das IPSEN-System derart von einer bestimmten Hardware abhängig zu machen und derartige Effizienzüberlegungen auch den Rahmen der bisher möglichen Untersuchungen gesprengt hätten, haben wir uns für die Benutzung des P-Codes entschieden. Da dieser ursprünglich für die Übersetzung von Pascal-Programmen entworfen worden ist, mußte er leicht modifiziert werden. So wurden zwei zusätzliche Adressierungsarten eingeführt, die für die gegenüber Pascal in Modula-2 vorhandenen Spracherweiterungen benötigt werden. Dies sind **external** für die Adressierung importierter Objekte und **modulelocal** für die Adressierung modullokaler Datenobjekte. Diese Erweiterungen entsprechen den im M-Code zur Verfügung stehenden Adressierungsarten (vgl. /Wi 81/). Weiterhin wurden weitere typabhängige arithmetische Operationen eingeführt, z.B. für arithmetische Operationen mit Objekten vom Typ CARDINAL, da dieser Standarddatentyp in Pascal nicht vorhanden ist. Da alle Kontrollstrukturen direkt interpretiert werden, werden keine Sprunganweisungen des P-Codes benutzt. Wir verzichten an dieser Stelle auf eine detaillierte Auflistung der im IPSEN-System verwendeten P-Code-Anweisungen. Der interessierte Leser sei auf /Pe 82/ bzw. /Sa 86/ verwiesen. Im folgenden werden wir auch die Erweiterungen des üblichen P-Codes als **P-Code-Anweisungen** bezeichnen.

Bei der Erzeugung von P-Code für Ausdrücke, Zuweisungsanweisungen und Prozeduraufrufe werden spezielle Informationen über die auftretenden Datenobjekte benötigt. Hierzu zählt z.B. der benötigte Speicherplatz für ein Objekt eines bestimmten Typs, die relative Distanzadresse des Datenobjekts zum Anfang eines entsprechenden Stack-Frames bzw. die statische Verschachtelungstiefe bei globaler Benutzung von Datenobjekten. Während für die Bestimmung dieser Größen in üblichen Compilern eine entsprechende Datenstruktur, die sogenannte **Symboltabelle**,

aufgebaut wird, haben wir alle benötigten Informationen bereits im Modulgraphen abgelegt. Das heißt im einzelnen:

- Da die Feinstruktur jeder Typdefinition durch mit "EType" markierte Kanten im Modulgraphen dargestellt wird (vgl. Abschnitt 3.2.6), kann durch einen entsprechenden Graphalgorithmus der **benötigte Speicherplatz** für Objekte dieses Typs schnell ermittelt werden. Diese Größe wird im Modulgraphen bei jeder Typdefinition im Attribut "Size" abgelegt.

- Mit Hilfe dieser Angabe über den benötigten Speicherplatz für ein Datenobjekt kann durch einen Graphalgorithmus der Deklarationsteil einer Prozedur durchlaufen werden und jedem Datenobjekt eine **relative Distanzadresse** zugeordnet werden. Diese Adresse wird an allen Bezeichnerknoten in einer Variablen- bzw. Parameterdeklaration in dem Attribut "**Address**" abgelegt.

- Zahlenliterale sind im Modulgraphen atomare Inkremente. Das heißt, daß an diesem Knoten ein Attribut existiert, in dem die zugehörige Quelltextdarstellung abgelegt ist. Neben dieser Quelltextdarstellung haben Zahlenliterale auch einen **konstanten Wert**, der bei der Erzeugung des P-Codes benötigt wird. Aus diesem Grund wird an jedem derartigen atomaren Inkrement ein weiteres Attribut, z.B. "CardinalValue", angelegt, in dem der Wert dieses Zahlenliterals steht. Dieser Wert wird bereits beim Eintragen des Knotens in den Modulgraphen während des Edierens berechnet (vgl. /Sc 86/). Dies bedeutet, daß bei dieser Vorgehensweise die auftretenden Konstanten nicht, wie sonst üblich, im Laufzeitdatenbereich abgelegt werden, sondern in Attributen im Modulgraphen stehen. Bei der Erzeugung entsprechender P-Code-Anweisungen werden diese Konstanten direkt in den P-Code eingetragen.

- Neben diesen konstanten Werten bei Zahlenliteralen sind in Modula-2 **konstante Ausdrücke** vorgesehen. Auch bei diesen Ausdrücken ist der Wert bereits zu dem Zeitpunkt, wenn der P-Code erzeugt wird, bekannt bzw. ermittelbar. Da für die Berechnung des Werts eines konstanten Ausdrucks u.U. arithmetische Operationen auszuführen sind, wird im Gegensatz zu obigen Zahlenliteralen diese Berechnung im IPSEN-System nicht bereits bei der Eingabe in den Modulgraphen durch den Editor durchgeführt. Erst bei der Erzeugung des P-Codes, in dem der Wert dieses konstanten Ausdrucks benötigt wird, wird für den konstanten Ausdruck ebenso P-Code erzeugt. Dieser P-Code wird dann ausgeführt und der Wert des konstanten Ausdrucks ermittelt. Der so berechnete Wert kann dann bei der weiteren Erzeugung von P-Code benutzt werden

- Bei der Erzeugung des P-Codes für Ausdrücke, Zuweisungsanweisungen bzw. Prozeduraufrufe muß zu jedem auftretenden Datenobjekt die zugehörige Speicheradresse ermittelt werden. Diese setzt sich zusammen aus der **relativen statischen Verschachtelungstiefe** und der **relativen Distanzadresse**. Diese Angaben

können schnell ermittelt werden, da im Modulgraphen zwischen einem angewandten Auftreten eines Datenobjekts und der zugehörigen Deklaration eine Folge kontextsensitiver Kanten existiert, die mit "ELocUse/SetDec" bzw. "EGlobUse/Set" markiert sind (vgl. Paragraph 5.2). Die Anzahl der hierbei zwischen einem angewandten Auftreten und der zugehörigen Deklaration existierenden Kanten gibt dabei die relative statische Verschachtelungstiefe an. Die relative Distanzadresse befindet sich in dem zuvor berechneten Wert des Attributs Address. Durch diese einfache Bestimmbarkeit der relativen statischen Verschachtelungstiefe liegt somit eine weitere Motivation für die im Paragraphen 5.2 eingeführten VarCopy-Listen und die entsprechenden "ELocUse/SetDec" bzw. "EGlobUse/Set" Kanten vor.

Zusammenfassend heißt dies, daß für die Realisierung des Hybrid-Interpreters eine Reihe von Informationen unmittelbar im Modulgraphen zu finden sind. Bei den noch fehlenden Informationen ist insbesondere bei den Attributen Size, Address und Code zu diskutieren, wann die entsprechenden Attributwerte im Modulgraphen abzulegen bzw. zu aktualisieren sind. Gemäß der im Kapitel 4 bei der Diskussion der Frage 3) erläuterten Möglichkeiten kann dies, wie bei den anderen Attributen üblich, bereits während des Edierens geschehen. Dieser Aufwand kann jedoch sehr hoch sein, da z.B. bei einer Änderung einer Deklaration eines Datenobjekts die dadurch betroffene Menge von zu aktualisierenden Address- und Code-Attributen nur schwer bestimmbar ist (vgl. /Sa 86/). Aus diesem Grunde haben wir uns dazu entschieden, bei einem **Neubeginn der Ausführung** diese drei Attribute Size, Address und Code stets vollständig neu zu berechnen. Hierbei beschränken wir uns allerdings darauf, diese Attribute nur dann zu bestimmen, wenn sie auch benötigt werden. Das heißt, daß erst bei Aufruf einer Prozedur alle Size- und Address-Attribute im Deklarationsteil neu berechnet werden. Ebenso werden die Code-Attribute nur derjenigen Inkremente berechnet, die während der Ausführung erreicht werden. Damit während der aktuellen Ausführung dieselben Attribute nicht mehrfach berechnet werden, erhält jeder Prozedurdeklarationsknoten bzw. jeder Knoten, an dem ein Code-Attribut abgelegt ist, als weiteres Attribut eine aktuelle **Versionsnummer**. Aufgrund dieser Versionsnummer ist es nicht nötig, vor dem Start der Ausführung alle im Modulgraph vorhandenen Size-, Address- und Code-Attribute zu löschen.

Bei einer rein compilativen Vorgehensweise wird bereits zur Übersetzungszeit die maximale Ausdehnung des im Stack-Frame enthaltenen **Zwischenergebniskellers** berechnet. Dadurch wird bereits bei Aufruf einer Prozedur und Ablegen eines neuen Stack-Frame auf dem Laufzeitkeller sichergestellt, daß der benötigte Speicherplatz für den Laufzeitkeller nicht in den Bereich für den Laufzeitheap hineinragt. Da bei unserer Vorgehensweise die Erzeugung des P-Code erst bei Erreichen der jeweiligen Inkremente während der Interpretation erfolgt, ist die maximale Ausdehnung des

Laufzeitkellers erst während der Ausführung eines Prozeduraufrufs bekannt. Dies hat den Vorteil, daß der Speicherplatz für den Zwischenergebniskeller nur dann reserviert wird, wenn er auch tatsächlich benötigt wird. Andererseits bedeutet dies, daß bei jedem Ablegen eines Zwischenergebnisses oben auf dem Laufzeitkeller überprüft werden muß, ob dieser Speicherplatz noch nicht durch den Laufzeitheap belegt ist.

Der beim Aufruf einer Prozedur auf dem Laufzeitkeller abgelegte **Stack-Frame** enthält im wesentlichen die üblichen Angaben wie Funktionswert, statischer, dynamischer Vorgänger und den u.U. benötigten Speicherplatz für Parameter und lokale Variable. Der Zeiger MP (MarkStackPointer) kennzeichnet den Anfang des aktuellen Frames und SP (StackPointer) kennzeichnet die maximale Ausdehnung. Eventuelle Zwischenergebnisse werden an der durch SP angegebenen Position abgelegt. Da der auszuführende Code bei unserer Vorgehensweise im Modulgraphen liegt, besteht die Rücksprungadresse aus der Angabe eines Modulgraphen und eines in ihm enthaltenen Knotens. Da wir für später noch zu erläuternde Anwendungen auch einen Rückverweis von einem Stack-Frame auf die zugehörige Prozedurdeklaration benötigen, legen wir als weitere Information die Angabe eines Modulgraphen und des zugehörigen Prozedurdeklarationsknoten im Stack-Frame ab. Damit hat ein solcher Stack-Frame schematisch den folgenden Aufbau:

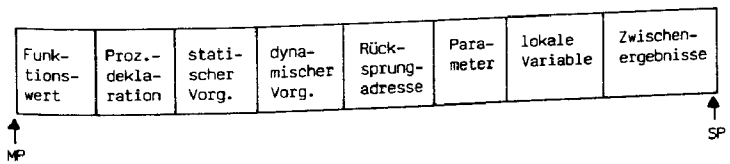


Abb. 7.1.2: Aufbau eines Stack-Frame

Da die einzelnen P-Code-Anweisungen typabhängig sind, werden auch in den Speicherplätzen für Parameter, lokale Variable bzw. Zwischenergebnisse Werte für Datenobjekte eines entsprechenden, in Modula-2 vordefinierten Typs abgelegt. Dies hat den Vorteil, daß bei einer Realisierung der Ausführung einzelner P-Code-Anweisungen (z.B. Addition zweier Cardinal-Zahlen) unmittelbar auf die in der **Realisierungssprache** vorhandenen arithmetischen Operationen zurückgegriffen werden kann. Dasselbe gilt z.B. für die Ausführung vordefinierter Funktionen (z.B. Odd, Mod). Wir haben bereits im Abschnitt 3.2.6 erläutert, daß eine zugehörige Deklaration derartiger vordefinierter Funktionen in jedem Modulgraphen enthalten ist (vgl. Abb. 3.2.6.2). Bei Aufruf einer derartigen Funktion wird an dem leeren, mit "Standard" markierten Prozedurrumpf erkannt, daß es sich um eine vordefinierte Funktion handelt. In diesem Fall wird die Interpretation des Modulgraphen unterbrochen und diese Funktion durch Aufruf einer in der Realisierungssprache vorhandenen Funktion realisiert. (Dies ist im IPSEN-System besonders einfach, da

sowohl die zu unterstützende Sprache, deren Programme interpretiert werden sollen, als auch die Sprache, in der das IPSEN-System realisiert ist, **Modula-2** ist).

Wir haben in diesem Paragraphen nur einen groben Überblick über die Vorgehensweise des im IPSEN-Systems eingesetzten Hybrid-Interpreters gegeben. In der Diplomarbeit /Sa 86/ findet man eine entsprechend detailliertere Erläuterung. In den nächsten Paragraphen werden wir erläutern, welche Vorteile dieser von uns gewählte inkrementelle Interpreteransatz bietet, um dem IPSEN-Benutzer umfangreiche Testunterstützungen anbieten zu können.

7.2 Realisierung von Unterbrechungspunkten

Im Paragraphen 2.3 haben wir verschiedene Möglichkeiten vorgestellt, wie der IPSEN-Benutzer den Verlauf der Ausführung beeinflussen kann. Hierzu zählten z.B. verschiedene Ausführungsarten, die bewirkten, daß die Ausführung an bestimmten Punkten automatisch unterbrochen wird bzw. vom IPSEN-Benutzer manuell unterbrochen werden kann. Weiterhin wurden eine Reihe von **impliziten Unterbrechungspunkten** erörtert (z.B. Auftreten von Laufzeitfehlern, Quelltextlücken, Aufruf importierter Prozeduren), an denen die Ausführung ebenfalls unterbrochen werden sollte. Alle diese Unterbrechungen der Ausführung können aufgrund unseres interpretativen Ansatzes direkt im Interpreter realisiert werden. Weiterhin ist bei einer Unterbrechung der Ausführung aufgrund einer einzigen, von allen Werkzeugen gemeinsam benutzten Datenstruktur die Stelle im Modulgraphen, an der die Ausführung unterbrochen wird, unmittelbar bekannt. Dies bedeutet, daß dem IPSEN-Benutzer die Unterbrechungsposition im Quelltext (u.U. mit einer entsprechenden Meldung) angezeigt werden kann (vgl. /Sa 86/).

Neben diesen impliziten Unterbrechungspunkten hat der IPSEN-Benutzer die Möglichkeit, **explizite Unterbrechungspunkte** zu setzen. Dies sind unbedingte bzw. bedingte Unterbrechungsanweisungen (vgl. Abschnitt 2.3.2). Da diese Unterbrechungsanweisungen sowohl im Quelltext dargestellt werden, als auch bei der Interpretation bei eingeschalteter Testumgebung beachtet werden sollen, müssen sie im Modulgraphen abgelegt werden. Hierbei bieten sich verschiedene Möglichkeiten an:

a) Ablage in einem zusätzlichen Attribut,
b) Einführung einer zusätzlichen, speziellen Graphinkrementart,
c) Simulation durch bereits vorhandene Graphinkrementarten.

Da z.B. in bedingten Unterbrechungsanweisungen boolesche Ausdrücke enthalten sind, die bei der Ausführung ausgewertet werden müssen, ist es naheliegend, derartige Ausdrücke wie beliebige Modula-2-Ausdrücke im Modulgraphen abzulegen. Dies bewirkt, daß die entsprechenden kontextsensitiven Kanten gezogen werden und

so bei der Ausführung wie üblich P-Code für diese Ausdrücke erzeugt werden kann, der dann ausgeführt wird. Dies zeigt, daß die unter a) angesprochene Alternative der Einführung zusätzlicher Attribute nicht sinnvoll ist. Verfolgt man die unter b) angesprochene Alternative, muß sowohl der Editor als auch der Interpreter an diese neuen Inkrementarten angepaßt werden. Die geringste Änderung ergibt sich, wenn man die bereits **existierenden Graphinkrementarten benutzt,** um diese zusätzlichen Unterbrechungsanweisungen zu **simulieren.** Zur Kennzeichnung, daß die herkömmlichen Graphinkremente für einen anderen Zweck eingesetzt werden, erhalten sie ein zusätzliches Attribut mit Namen "Task", in dem vermerkt ist, ob es sich um ein herkömmliches Modula-2-Graphinkrement handelt. Wir erläutern dies im folgenden an einigen konkreten Beispielen.

Im vorigen Abschnitt haben wir erläutert, daß bei Aufruf einer vordefinierten Prozedur die Interpretation unterbrochen wird und diese Prozedur durch Aufruf einer vordefinierten Prozedur der Programmiersprache realisiert wird, in der der Interpreteralgorithmus realisiert ist. Ausschlaggebend für die Unterbrechung war hierbei, daß im Modulgraphen in der entsprechenden Prozedurdeklaration anstelle eines Rumpfes nur ein Knoten mit der Markierung "Standard" existierte. Dieses Verhalten nutzen wir nun aus, um eine **unbedingte Unterbrechungsanweisung** zu realisieren. Im Modulgraphen wird hierzu eine weitere, parameterlose vordefinierte Prozedur mit dem Namen "**Break**" eingetragen (vgl. hierzu Abb. 3.2.6.2). Fügt der IPSEN-Benutzer dann an einer beliebigen Stelle im Anweisungsteil eine unbedingte Unterbrechungsanweisung ein, wird im Modulgraphen an dieser Stelle ein Aufruf dieser Prozedur Break eingetragen. Wird dann während der Interpretation dieser Prozeduraufruf erreicht, wird die Ausführung unterbrochen, da eine vordefinierte Prozedur auszuführen ist. An diesem Unterbrechungspunkt kann dann die Eingabe beliebiger anderer Benutzerkommandos verwaltet werden, bevor die Interpretation fortgesetzt wird. Da dieser Prozeduraufruf wie ein beliebiger Prozeduraufruf in den Modulgraphen eingetragen wird, ist sichergestellt, daß die Kontrollflußkanten korrekt gezogen werden. Da während der Ausführung derartige Unterbrechungsanweisungen nur beachtet werden sollen, wenn die Testumgebung eingeschaltet ist, muß der Interpreter erkennen können, daß es sich bei diesem Prozeduraufruf um die Simulation einer Unterbrechungsanweisung handelt. Aus diesem Grunde erhält an einem solchen Prozeduraufruf das Attribut Task den Wert "BreakPoint". Diese Information benötigt auch der Unparser, da derartige Zusatzinformationen im Quelltext ebenfalls nur bei angeschalteter Testumgebung dargestellt werden sollen. Außerdem muß der Unparser erweitert werden, so daß er einen derartigen Prozeduraufruf als unbedingte Unterbrechungsanweisung darstellt. Bei einem tabellengesteuerten Unparser bedeutet dies die Angabe eines entsprechenden Unparsing-Schemas. Abbildung 7.2.1 illustriert die Darstellung einer unbedingten Unterbrechungsanweisung im Modulgraphen.

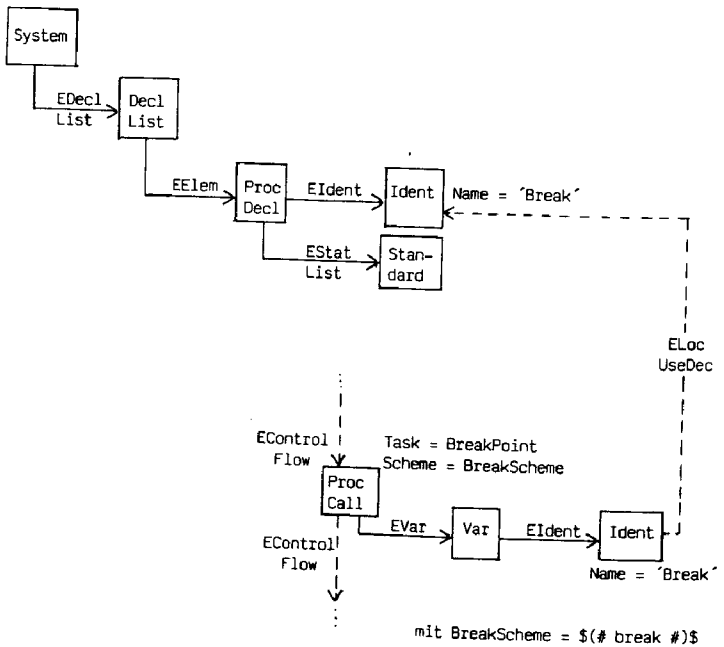


Abb. 7.2.1: Darstellung einer unbedingten Unterbrechungsanweisung

Bedingte Unterbrechungsanweisungen werden im Modulgraphen durch einseitig bedingte Anweisungen dargestellt. Da die Ausführung unterbrochen werden soll, wenn die Bedingung nicht erfüllt ist, wird obige Prozedur Break im else-Teil aufgerufen. Wir illustrieren dies an dem folgenden Beispiel:

Quelltextdarstellung:

```
...
(# assert: Last >= 0  #)
   (# break #)
(# end assert #)
...
```

-->

Interne Realisierung:

```
...
IF Last >= 0 THEN
ELSE
   Break
END;
...
```

Abb. 7.2.2: Interne Realisierung einer bedingten Unterbrechungsanweisung

Das Eintragen der Bedingung in einer bedingten Unterbrechungsanweisung als Bedingung in einer if-Anweisung bewirkt, daß direkt alle kontextsensitiven Kanten im Modulgraphen gezogen werden. Die Auswertung der Bedingung und damit die Überprüfung, ob die Ausführung an dieser Stelle zu unterbrechen ist, geschieht somit in der gleichen Art und Weise wie die übrige Interpretation. Analog zu unbedingten Unterbrechungsanweisungen muß natürlich auch in diesem Fall ein spezielles

Unparsing-Schema in den Modulgraphen eingetragen werden. Außerdem erhält das Attribut Task an der if-Anweisung den Wert "**Assertion**".

Für die Eingabe der booleschen Bedingung durch den IPSEN-Benutzer kann auf den **Parser** zurückgegriffen werden. Das heißt, die vom Benutzer in freier Eingabe eingegebene Bedingung wird vom Parser analysiert und in den Modulgraphen eingetragen.

Diese Erläuterungen zeigen, daß bei der Realisierung von Unterbrechungsanweisungen auf die vorhandenen Komponenten Editor, Parser, Unparser und Interpreter zurückgegriffen werden kann. Wir werden im folgenden sehen, daß sich die Idee, bei der Realisierung auf diese **vorhandenen Komponenten zurückzugreifen,** auch bei der Realisierung der Testunterstützungskommandos anwenden läßt.

8 Realisierung der Testunterstützung

Im Paragraphen 2.4 haben wir eine Reihe von Kommandos vorgestellt, die dem IPSEN-Benutzer zum Testen des aktuell erstellten Moduls zur Verfügung stehen. Wir werden in diesem Kapitel erläutern, wie diese Kommandos der Werkzeuge Testvorbereitung bzw. Ausführung und Test unter Benutzung der gemeinsamen, zentralen Datenstruktur Modulgraph realisiert werden können.

8.1 Realisierung der Ausgabe von Variablenwerten

Nach Aufruf des Kommandos Xv - variable inspection (vgl. Abschnitt 3.4.1) hat der Benutzer die Möglichkeit, einen Variablenbezeichner einzutippen. Nach Beendigung der Eingabe ist es Aufgabe des IPSEN-Systems, den entsprechenden Typ dieses Bezeichners zu ermitteln. Kennzeichnet der eingetippte Bezeichner ein Datenobjekt eines elementaren Datentyps (z.B. CARDINAL, INTEGER etc.), wird der Wert dieses Datenobjekts dem Benutzer angezeigt. Kennzeichnet der Bezeichner einen komplexen Datentyp (z.B. ARRAY, RECORD etc.), werden keine Werte ausgegeben. Dem Benutzer werden in diesem Fall die relevanten Typdeklarationen angezeigt. Diese Information erleichtert es ihm, den eingetippten Bezeichner zu einem Variablenbezeichner eines Objekts eines elementaren Datentyps zu verlängern. Die für die Realisierung dieses Kommandos benötigten Informationen Typ und Adresse im Laufzeitdatenbereich können am einfachsten ermittelt werden, wenn der eingetippte **Bezeichner** wie bei einer Editoraktion **in den Modulgraphen eingetragen wird.** In diesem Fall wird, wie bei allen Editoraktionen, sofort überprüft, ob der eingetragene Bezeichner syntaktisch korrekt und an der aktuellen Position gültig und sichtbar ist. Derartige kontextsensitive Beziehungen werden durch entsprechende Kanten ("ELocUse/SetDec" bzw. "ELocUse/Set") im Graphen dargestellt. Ebenso wird eine "EVarType"- bzw. "EExprType"-Kante zur entsprechenden Typdefinition gezogen. An Hand dieser Kanten kann dann durch einen einfachen Graphalgorithmus **Adresse** und **Typ** des eingegebenen Variablenbezeichners ermittelt werden (vgl. Paragraph 7.1). Falls der eingegebene Variablenbezeichner kein Objekt eines elementaren Datentyps kennzeichnet, müssen ausgehend von dem Knoten im Modulgraphen, der den aktuellen Typ des eingegebenen Bezeichners angibt, alle relevanten Typdefinitionen ermittelt und dem Benutzer angezeigt werden. Dieser Vorgang stellt somit eine weitere Form eines **Unparsing-Prozesses** dar. Bei der Ermittlung der relevanten Typdefinitionen sind die im Modulgraphen bereits existierenden "EType"-Kanten hilfreich, die die Abhängigkeiten zwischen verschiedenen Typdefinitionen darstellen.

Um nun eine Möglichkeit zu haben, den vom Benutzer eingegebenen Bezeichner in den Graphen eintragen zu lassen, führen wir analog zu obiger Prozedur "Break"

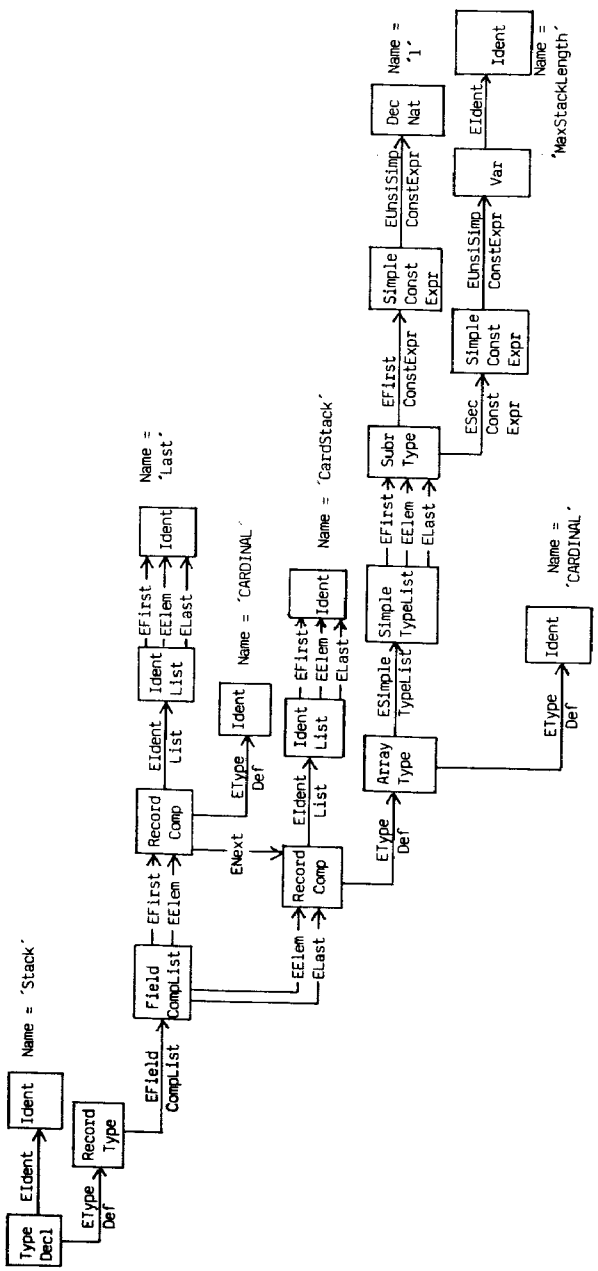
eine weitere vordefinierte Prozedur mit Namen "Inspec" ein, deren Rumpf ebenfalls nur aus einem mit "Standard" markierten Knoten besteht (vgl. Abb. 3.2.6.2). Weiterhin besitzt diese Prozedur einen formalen Parameter, so daß der eingegebene Variablenbezeichner als aktueller Parameter eines Aufrufs dieser Prozedur "Inspec" in den Graphen eingetragen werden kann.

Für die Analyse des eingegebenen Variablenbezeichners und das Eintragen in den Modulgraphen wird bei der Realisierung auf den vorhandenen Parser zurückgegriffen. Da der eingegebene Bezeichner ein Objekt eines beliebigen vordefinierten bzw. vom Benutzer selbstdefinierten Datentyps kennzeichnen kann, erhält der formale Parameter der Prozedur Inspec den mit jedem anderen Typ **kompatiblen Typ** "ARRAY OF WORD". Dies verhindert, daß beim Eintragen des Bezeichners in den Graphen aufgrund etwaiger Typunverträglichkeiten zwischen aktuellem und formalem Parametertyp Fehler auftreten. (Diese Vorgehensweise findet man bei Programmiersprachen üblicherweise bei der Realisierung von Ein-/Ausgabeprozeduren (Read, Write). Auch dort existiert in der Regel nur eine Prozedur für die Ein- bzw. Ausgabe von Objekten verschiedener Typen (vgl. z.B. /DL 84/)). Der Typ WORD wird als zusätzlicher vordefinierter Typ in die Standardumgebung eines jeden Modulgraphen aufgenommen.

8.2 Realisierung der Ausgabe eines Speicherauszugs

Während bei der Realisierung des oben erläuterten Kommandos Xv - variable inspection der Benutzer einen Variablenbezeichner eingibt, müssen bei der Realisierung des Kommandos Xd - dump alle Variablenbezeichner automatisch vom IPSEN-System **erzeugt** werden. Als Beispiel wiederholen wir in Abbildung 8.2.1 noch einmal die bereits in Abb. 2.4.1.1 angegebene Typdeklaration eines Stacks und geben einen Ausschnitt der zugehörigen Modulgraphdarstellung an.

Ein Objekt dieses Typs besteht aus sechs Elementen, der Komponente Last und einem fünfelementigen Feld. Für jedes dieser Elemente muß nun der entsprechende Bezeichner erzeugt werden, um bei der Ausgabe eines Speicherauszugs alle gültigen Variablenbezeichner dem Benutzer anzeigen zu können. Hierzu muß eine Transformation durchgeführt werden, die aus der Modulgraphdarstellung einer Variablendeklaration und zugehöriger Typdeklaration die textuelle Darstellung aller möglichen Variablenbezeichner liefert (vgl. Abb. 2.4.2.1). In diesem Sinne stellt diese Transformation ebenfalls einen speziellen **Unparsing-Prozeß** dar. Analog zu der im Paragraphen 6.3 erläuterten Vorgehensweise, wird auch diese Tranformation im Prinzip durch einen **rekursiven Algorithmus** realisiert. Es bestehen jedoch einige entscheidende Unterschiede:




```
CONST MaxStackLength = 5;

TYPE Stack = RECORD
               Last : CARDINAL;
               CardStack : ARRAY [ 1 .. MaxStackLength ] OF CARDINAL
             END
```

Abb. 8.2.1: Modulgraphdarstellung einer Typdeklaration

- Bei der im Paragraphen 6.3 beschriebenen Quelltextdarstellung ist die Struktur des rekursiven Algorithmus identisch mit der Struktur des zugrundeliegenden abstrakten Syntaxgraphen bzw. -baums. Das bedeutet insbesondere, daß jeder Teil des Graphen genau einmal durchlaufen wird. Bei der Erzeugung aller möglichen Variablenbezeichner gilt dies nicht. So müssen z.B. bei auftretenden Unterbereichstypen bestimmte Teile des Graphen entsprechend der Anzahl von Komponenten dieses Unterbereichstyps durchlaufen werden (siehe obiges Beispiel). Das bedeutet, daß der rekursive Transformationsalgorithmus in diesem Fall auch **iterative Schritte** enthält.

- Der Algorithmus zur Erzeugung von Quelltext wird bei der Ausführung nur durch die im abstrakten Syntaxgraphen verkapselte kontextfreie Struktur gesteuert. Bei der Erzeugung von Variablenbezeichnern ist die **Ausführung** darüberhinaus z.B. auch **abhängig** von dem Wert konstanter Ausdrücke, von dem Typ eines aktuellen Parameters bei offenen Feldern als formalen Parametertyp bzw. von dem Wert eines Diskriminators in einem varianten Verbund. Im letzteren Fall bestimmt z.B. der aktuelle Wert eines Diskriminators, welcher Teil des Modulgraphen weiter durchlaufen werden muß, um die Bezeichner für die entsprechenden Verbundkomponenten zu erzeugen.

- Falls die Typbeschreibung des derzeit zu erzeugenden Variablenbezeichners über mehrere Typdeklarationen verteilt ist, müssen bei der Erzeugung des Variablenbezeichners die Graphdarstellungen dieser Typdeklarationen nacheinander durchlaufen werden. Die Abhängigkeiten zwischen den einzelnen Typdefinitionen werden im Modulgraphen durch EType-Kanten dargestellt. Dies bedeutet, daß in diesem Fall die Ausführung nicht an dem zugrundeliegenden abstrakten Syntaxgraph bzw. -baum orientiert ist, sondern an dem zusätzlich durch EType-Kanten beschriebenen **Graph**, der die **Typstruktur und -abhängigkeiten** darstellt.

Dieser **Unparser für Variablenbezeichner** stellt somit einen Transformationsalgorithmus von einer Graph- in eine Quelltextdarstellung dar, dessen Funktionalität weit über die Funktionalität des im Kapitel 6 beschriebenen Unparsers hinausgeht. Aus diesem Grunde wurde diese Transformation bisher auch nur als hart verdrahteter rekursiver Algorithmus realisiert. Es bleibt zu untersuchen, inwieweit die im Abschnitt 6.2.2 vorgestellte Sprache für Unparsing-Schemata zu erweitern ist, damit auch dieser Unparser für Variablenbezeichner als tabellengesteuerter Algorithmus realisiert werden könnte.

Für jeden erzeugten Variablenbezeichner muß anschließend der **aktuelle Wert** ermittelt werden. Dies geschieht analog zu der im letzten Paragraphen erläuterten Vorgehensweise, indem nacheinander jeder einzelne Variablenbezeichner als aktueller Parameter eines Aufrufs der Prozedur "Inspec" in den Modulgraphen eingetragen

wird. Dies wird auch wieder durch Benutzung des Parsers realisiert. Mit Hilfe der dabei in den Graphen eingetragenen kontextsensitiven Kanten und der vom Interpreter eingetragenen Attribute Address und Size kann die relative Distanzadresse im zugehörigen Stack-Frame im Laufzeitdatenbereich ermittelt werden.

Damit kann das Kommando nun wie folgt realisiert werden:
Da der Inhalt des Laufzeitkellers abhängt von der aktuellen dynamischen Verschachtelung, muß bei Aufruf des Kommandos Xd - dump ermittelbar sein, zu welcher Prozedurdeklaration der jeweilige Stack-Frame gehört. Wie bereits im Paragraphen 7.1 erläutert, war hierzu in jeden Stack-Frame die Angabe des Knotens der zugehörigen Prozedurdeklaration im Graphen mit aufgenommen worden. Mit Hilfe dieser Angabe und der Verkettung der dynamischen Vorgänger kann ermittelt werden, zu welcher Prozedurdeklaration der i-te Stack-Frame gehört. Zu einer Prozedurdeklaration können dann mit obigem Unparser alle Bezeichner von Parametern und lokalen Variablen erzeugt werden. Für jeden Bezeichner kann die relative Distanzadresse ermittelt werden, so daß der zugehörige Wert dem Benutzer ausgegeben werden kann.

8.3 Realisierung der Simulation von Prozeduraufrufen

Bei Aufruf des Kommandos Xp - proc. call simulation kann der IPSEN-Benutzer allen call-by-reference-Parametern bzw. den zu der aufgerufenen Prozedur globalen Variablen einen neuen Wert zuweisen. Hierzu werden ihm zunächst alle diese Bezeichner mit ihren entsprechenden Werten in einem Fenster angezeigt. Die Erzeugung der hierfür benötigten Bezeichner geschieht analog zu der oben beschriebenen Vorgehensweise. Anschließend hat der Benutzer die Möglichkeit, einzelne Variablenbezeichner zu selektieren und diesen Variablen einen neuen Wert zuzuweisen. Da der IPSEN-Benutzer diesen Wert auch in Form eines beliebigen Ausdrucks formulieren kann, kann diese Wertzuweisung am einfachsten realisiert werden, wenn sie im Modulgraphen wie eine übliche Zuweisungsanweisung dargestellt wird. Hierzu wird vom Parser der Variablenbezeichner als linke Seite und der vom Benutzer eingegebene Ausdruck als rechte Seite dieser Zuweisungsanweisung in den Modulgraphen eingetragen. Durch Aktivierung des Interpreters wird diese Zuweisungsanweisung ausgeführt und der zugehörige Wert im Laufzeitdatenbereich entsprechend verändert. Der neue Wert kann anschließend dem Benutzer angezeigt werden.

Nachdem alle gewünschten Variablen vom IPSEN-Benutzer derartig gesetzt worden sind, wird der eigentliche Prozeduraufruf im Modulgraphen übersprungen und die Ausführung mit der im Kontrollfluß folgenden Anweisung fortgesetzt.

8.4 Realisierung der Laufzeit-/Speicherplatzstatistik

Laufzeitzähler dienen dazu, die Häufigkeit der Ausführung bestimmter Anweisungen bzw. des Zugriffs auf bestimmte Variable zu ermitteln (vgl. Abschnitt 2.4.4). Diese zu diesem Zweck während der Ausführung zu verwaltenden Zähler können je nach Aufgabenstellung unterschiedlich realisiert werden. Soll z.B. die Anzahl aller ausgeführten Anweisungen ermittelt werden, ist es sicherlich am effizientesten, solch einen **Zähler im Interpreteralgorithmus** zu realisieren, der dann während des Interpretierens beim Durchlauf durch den Modulgraphen jeweils beim Erreichen der nächsten Anweisung um 1 erhöht wird. Eine derartige Erweiterung des Interpreteralgorithmus ist jedoch nicht möglich, wenn z.B. die Anzahl der Durchläufe einer bestimmten Schleife gezählt werden soll (vgl. Abschnitt 2.4.4). In diesem Fall muß die entsprechende Schleife im Modulgraphen geeignet gekennzeichnet werden. Außerdem muß der Schleifenzähler mit einem vom Benutzer frei wählbaren Namen bezeichnet werden können, über den dann auch Anfragen an den aktuellen Wert dieses Schleifenzählers gemacht werden können. Um derartige Anfragen analog zu Anfragen an sonstige Programmvariable realisieren zu können und auch die Verwaltung derartiger Zähler nicht ein zweites Mal realisieren zu müssen, werden diese **Schleifenzähler** wie sonstige Datenobjekte innerhalb der aktuellen Prozedur **deklariert.** Die Initialisierung dieses Schleifenzählers und die Erhöhung des Schleifenzählers um 1 bei jedem Schleifendurchlauf werden durch **zwei zusätzliche Zuweisungsanweisungen** vor der Schleife bzw. als erste Anweisung im Schleifenrumpf realisiert. Insgesamt wird die interne Darstellung eines Programms bei Einführung eines Schleifenzählers wie folgt modifiziert:

```
Quelltextdarstellung:                          interne Darstellung:

PROCEDURE ...                                  PROCEDURE ...
  ...                                          ...
                                               VAR Schleifel : CARDINAL;
BEGIN                                          BEGIN
  ...                                            ...
  (# loop counter: Schleifel #)        -->       Schleifel := 0;
  WHILE  X >= 0 DO                               WHILE  X >= 0 DO
                                                   Schleifel := Schleifel + 1;
    ...                                            ...
  END;                                           END;
  ...                                            ...
END;                                           END;
```

Abb. 8.4.1: Interne Darstellung eines Schleifenzählers

Damit kann die Realisierung eines Schleifenzählers auf die Modulgraphmodifikationen zum Eintragen einer Variablendeklaration bzw. einer Zuweisungsanweisung zurückgeführt werden. Zur Kennzeichnung, daß die Deklaration des Schleifenzählers und die beiden Zuweisungsanweisungen zur Realisierung eines

Testunterstützungskommandos dienen, erhalten sie ein zusätzliches Task-Attribut. Dies bedeutet, daß diese Einträge im Modulgraphen nur beachtet werden, wenn die Testumgebung eingeschaltet ist. Ebenso muß auch die Unparsing-Information modifiziert werden, da für diese zusätzliche Deklaration sowie die beiden Zuweisungsanweisungen nicht die übliche Quelltextdarstellung erzeugt werden darf. Vor der Darstellung der while-Schleife muß vielmehr eine Zeile mit der Darstellung des Schleifenzählers eingefügt werden. Hierzu wird das Unparsing-Schema der Zuweisungsanweisung zur Initialisierung des Schleifenzählers durch das folgende Schema

```
$(# loop counter: $ [ <EVar>/ / ] $ #)$
```

ersetzt. Um zu gewährleisten, daß die zusätzliche Variablendeklaration sowie die im Schleifenrumpf enthaltene Zuweisungsanweisung im Quelltext nicht dargestellt werden, wird ihnen das leere Unparsing-Schema Scheme = " " zugeordnet.

Beim Löschen des Schleifenzählers muß neben diesen direkt an der while-Schleife stehenden Zuweisungsanweisungen auch die zugehörige Variablendeklaration gelöscht werden. Diese kann aufgrund der kontextsensitiven, mit "ELocSetDec" markierten Kante vom setzenden Auftreten des Schleifenzählers zur Deklarationsstelle direkt ermittelt werden.

Die Behandlung eines Schleifenzählers analog zu üblichen Datenobjekten hat den weiteren Vorteil, daß die Ausgabe des Wertes von Schleifenzählern genauso realisiert werden kann wie bei Variablen. Insgesamt kann also auch die **Realisierung** dieser für betimmte Anweisungen geltenden Laufzeitzähler auf die bisher vorgestellten **Aktionen zur Modulgraphveränderung** bzw. **Ausführung** zurückgeführt werden.

Ausgabe von Speicherplatzstatistik bedeutet im IPSEN-System im wesentlichen, dem Benutzer die aktuellen Werte der für die Verwaltung des Laufzeitdatenbereichs eingesetzten Zeiger StackPointer, MarkStackPointer bzw. NewPointer mitzuteilen. Soll die im Laufzeitdatenbereich benötigte Größe für ein bestimmtes Datenobjekt ermittelt werden, wird analog zur Realisierung der Ausgabe des Wertes eines bestimmten Datenobjekts dieses Datenobjekt zunächst in den Modulgraphen eingetragen (z.B. als aktueller Parameter der Prozedur "Inspec"). An Hand der im Modulgraphen eingetragenen Kanten zur Darstellung kontextsensitiver Beziehungen und der aktuellen Werte des Attributs "Size" kann dann die benötigte Speicherplatzgröße ermittelt werden. Die Darstellung des Wertes solcher Speicherplatzgrößen ist somit ebenfalls eine **Transformation** von der Modulgraphdarstellung und dem Laufzeitdatenbereich in eine textuelle Darstellung (vgl. Abb. 2.4.4.2). Eine andere, z.B. **graphische Visualisierung** dieser Größen auf dem Bildschirm bedeutet deshalb nur, daß dieser Transformations-, d.h. **Unparsing-Prozeß** durch einen anderen ersetzt

werden muß.

8.5 Realisierung der Testumgebung

Wir haben erläutert, wie die Realisierung der Testunterstützungskommandos z.T. auf das Eintragen von Zuweisungsanweisungen und Prozeduraufrufen in den Modulgraphen zurückgespielt werden kann. Ein analoges Vorgehen haben wir auch bereits im Paragraphen 7.2 bei der Realisierung der Unterbrechungsanweisungen erörtert. Alle diese zusätzlichen Einträge können nun **im Modulgraphen stehen bleiben,** wenn der IPSEN-Benutzer wünscht, daß bei einer wiederholten Ausführung des entsprechenden Modulgraphteils dieselben Aktionen zur Testunterstützung durchgeführt werden sollen. Entsprechend können die Kommandos des Werkzeugs Testvorbereitung so realisiert werden, daß entsprechende Ergänzungen in den Modulgraphen eingetragen werden, die dann bei einer Ausführung bei eingeschalteter Testumgebung zu beachten sind. Diese zusätzlichen Einträge sind im wesentlichen Aufrufe zusätzlich vordefinierter Prozeduren mit leerem, im Graphen mit "Standard" markierten Rumpf (vgl. "Break", "Inspec"). Dies ermöglicht dem IPSEN-Benutzer, im voraus eine bestimmte Testumgebung aufzubauen bzw. während der Ausführung aufgerufene Testunterstützungskommandos im Modulgraphen zu konservieren.

Damit allen Werkzeugen bekannt ist, daß in diesem Fall übliche Modula-2-Anweisungen für Testzwecke benutzt werden, erhalten alle diese Einträge im Modulgraphen ein **zusätzliches Attribut "Task"** mit entsprechender aktueller Attributierung. Ebenso muß durch entsprechende Unparsing-Schemata bzw. Ergänzung des hart verdrahteten Unparsers dafür gesorgt werden, daß diese Modulgrapheinträge entsprechend im Quelltext darzustellen sind.

Das **Ein- bzw. Ausschalten der Testumgebung** wird im IPSEN-System durch eine für alle Werkzeuge bekannte globale Variable vermerkt. Bei eingeschalteter Testumgebung werden dann alle diese zusätzlichen Modulgrapheintragungen im Quelltext geeignet dargestellt. Ebenso werden sie bei der Ausführung vom Interpreter beachtet. Während Zuweisungsanweisungen und bedingte Anweisungen wie üblich ausgeführt werden, bedeutet der Aufruf einer der zusätzlichen vordefinierten Prozeduren eine Unterbrechung der Ausführung. In diesem Fall werden dann die oben erläuterten Aktionen durchgeführt und anschließend die Ausführung fortgesetzt.

8.6 Realisierung des Taschenrechners

Da die Benutzung eines Taschenrechners (vgl. Abschnitt 2.4.6) nichts anderes als die Ausführung einer Zuweisungsanweisung ist, werden die oben erläuterten Techniken auch zur Realisierung des Taschenrechners eingesetzt.

Das heißt im einzelnen:

- In dem jeden Modulgraphen umgebenden Graphen zur Darstellung der Standard-datentypen und vordefinierten Prozeduren werden drei Variablen DeskCard, DeskInt bzw. DeskReal vom Typ CARDINAL, INTEGER bzw. REAL deklariert.
- Bei Aufruf des Taschenrechners und Eingabe eines arithmetischen Ausdrucks wird zunächst der Parser aktiviert, um diesen vom Benutzer eingegebenen Ausdruck als rechte Seite einer am aktuellen Unterbrechungspunkt neu eingefügten Zuweisungs-anweisung in den Modulgraphen einzutragen. (Wird der Taschenrechner nicht während einer Unterbrechung der Ausführung aufgerufen, wird diese Zuweisungs-anweisung als einzige Anweisung in einen neu zu erzeugenden Modulgraphen eingetragen.) Da beim Eintrag dieses Ausdrucks auch alle kontextsensitiven Kanten gezogen werden, wird u.a. eine mit "ExprType" markierte Kante gezogen, die den Typ dieses Ausdrucks festlegt.
- In Abhängigkeit von diesem Typ ist es nun möglich, auf der linken Seite der Zuweisungsanweisung eine der Variablen DeskCard, DeskInt bzw. DeskReal einzutragen.
- Diese Zuweisungsanweisung kann wie üblich interpretiert werden. Der Wert des arithmetischen Ausdrucks steht dann im Laufzeitdatenbereich an der für die Variable auf der linken Seite reservierten Stelle, so daß der Wert dieser Variablen ohne Schwierigkeiten ermittelt werden und dem Benutzer angezeigt werden kann.
- Anschließend kann diese zusätzliche Zuweisungsanweisung im Modulgraphen wieder gelöscht werden.

9 Entwurf

Wir haben im bisherigen Teil dieser Arbeit auf konzeptioneller Ebene erläutert, wie die im Kapitel 2 beschriebenen Anforderungen an die Werkzeuge des Programmierens-im-Kleinen im IPSEN-System realisiert werden können. Hierzu haben wir zunächst erläutert, wie mit Hilfe des Kalküls der Graph-Ersetzungs-systeme auf systematische Weise eine Spezifikation der zugrundeliegenden Datenstruktur Modulgraph mit ihren dazugehörigen Zugriffsoperationen erstellt werden kann. Im weiteren Verlauf der Arbeit haben wir dann, weiterhin auf konzeptioneller Ebene, aber weniger formal beschrieben, welche weiteren Daten-strukturen z.B. für die Ausgabe bzw. Ausführung eines Modulgraphen benötigt werden. Unter Benutzung dieser Datenstrukturen haben wir erläutert, wie die Kommandos der einzelnen Werkzeuge realisiert werden können. Diese Erläuterungen stellen somit einerseits auf konzeptioneller Ebene eine **Spezifikation** des Vehaltens der einzelnen Werkzeuge im Bereich des Programmierens-im-Kleinen dar. Anderer-seits sind viele der obigen Argumente und Begründungen auch unmittelbar anwendbar bei dem in diesem Kapitel vorzustellenden **Entwurf des IPSEN-Systems.** In diesem Sinne sind die obigen Erläuterungen größtenteils auch Bestandteil einer Begründung des Entwurfs ("design rationale") des IPSEN-Systems.
Bei der Vorstellung des Entwurfs, auch **Software-Architektur** genannt, werden wir uns auf die Teilkomponenten beschränken, die für das Verständnis dieser Arbeit benötigt werden. Ein Überblick über den gesamten Entwurf des Programmieren-im-Kleinen-Anteils des IPSEN-Systems befindet sich in /ES 85b/ und in detaillierterer Form in /Sc 86/.

Bei der Darstellung des Entwurfs stützen wir uns im wesentlichen auf das ebenfalls im IPSEN-Projekt entwickelte **Modulkonzept** ab, das an anderer Stelle ausführlich beschrieben worden ist (vgl. /Ga 83/, /Le 84/, /LN 85/). Bei diesem Modulkonzept unterscheiden wir drei Arten von Moduln: Datentyp-, Datenobjekt- und Funktionsmoduln. Dabei verkapseln **Datentyp-** bzw. **Datenobjektmoduln** Daten-strukturen mit zugehörigen Zugriffsoperationen. **Funktionsmoduln** hingegen ver-kapseln komfortablere Zugriffsoperationen auf derartige Datenstrukturen bzw. Transformationen zwischen verschiedenen Datenstrukturen.
Zwischen zwei Moduln kann eine **Benutzbarkeitsbeziehung** bestehen, die besagt, daß zur Realisierung der Ressourcen eines Moduls die Ressourcen eines anderen Moduls benutzt werden dürfen. Werden die Ressourcen eines Moduls nur in einem lokalen Kontext zur Verfügung gestellt, spricht man von **lokaler Benutzbarkeit,** ansonsten von **genereller Benutzbarkeit.**

Software-Architekturen können sehr gut durch eine graphartige Darstellung veranschaulicht werden. Hierzu werden die einzelnen Moduln durch Knoten und die Benutzbarkeitsbeziehungen durch Kanten dargestellt. Auch wir werden im folgenden

diese Darstellungsweise wählen. In der Legende zu den folgenden Abbildungen erläutern wir, wie die einzelnen Modultypen und verschiedenen Beziehungen konkret dargestellt werden.

Bei der Entwicklung der Software-Architektur für das IPSEN-System hat sich herausgestellt, daß die Architektur aus mehreren aufeinanderliegenden Schichten von Moduln zusammengesetzt ist. Wir werden im folgenden diesen **schichtenartigen Entwurf** in einer bottom-up Vorgehensweise von unten nach oben erläutern und beschreiben, welche spezifische Bedeutung die einzelnen Schichten verkapseln. Im Paragraphen 9.4 fassen wir dann noch einmal den gesamten Entwurf zusammen und stellen ihn in Abb. 9.4.2 in geschlossener Form dar.

9.1 IPSEN-spezifische Datenstrukturen

Die unterste Schicht der Architektur ist eine **datenstruktur-orientierte Schicht.** Sie enthält alle Datentyp- und Datenobjektmoduln, die für die Realisierung der einzelnen Werkzeuge benötigt werden.

Wir haben an vielen Stellen in dieser Arbeit erläutert, daß alle Werkzeuge auf der gemeinsamen Datenstruktur Modulgraph operieren. Aus diesem Grunde ist eine wesentliche Entwurfsentscheidung im IPSEN-System, nicht nur auf konzeptioneller Ebene, sondern auch auf der **Realisierungsebene** eine **graphartige Datenstruktur** zur Verfügung zu haben. Da weiterhin Graphen als zentrale Datenstukturen nicht nur im Bereich des Programmierens-im-Kleinen, sondern auch in allen anderen vom IPSEN-System unterstützten Aufgabengebieten benutzt werden, wurde ein relationales Datenbanksystem entwickelt, das speziell auf die Abspeicherung **beliebiger attributierter Graphen** zugeschnitten ist (/BL 85/). Dieses Datenbanksystem, auch **Graphenspeicher** genannt, ermöglicht, alle im IPSEN-System auftretenden graphartigen Datenstrukturen abzuspeichern. Hierzu stellt es Operationen zur Verfügung wie "Kreieren/Kopieren/Löschen von Graphen", "Einfügen/Löschen von Knoten mit einer bestimmten Markierung", "Einfügen/Löschen bestimmter Kanten mit einer bestimmten Markierung" und "Lesen/Schreiben von Knotenattributen". Weiterhin erlaubt es assoziative Anfragen der Art "Existiert eine ein-/auslaufende Kante mit einer bestimmten Markierung" bzw. "Gib Ziel-/Quellknoten einer bestimmten aus-/einlaufenden Kante".
Architektur und Realisierungskonzepte dieses Graphenspeichers sind an anderer Stelle beschrieben worden (/Ba 84/, /BL 85/). In dem hier dargestellten Entwurf verkapseln wir ihn durch einen abstrakten Datentyp im Datentypmodul "AttributedGraph", der die obigen, beispielhaft angegebenen Ressourcen exportiert.

Während in diesem Graphenspeicher beliebige attributierte Graphen abgelegt

werden können, wird im Bereich des Programmierens-im-Kleinen nur eine **eingeschränkte Klasse** attributierter Graphen benötigt, nämlich die Klasse der Modulgraphen. Aus diesem Grunde ist es sinnvoll, den abstrakten Datentyp "AttributedGraph" zu spezialisieren. Dieser abstrakte Datentyp kennt dann nur Knoten- und Kantenmarkierungen sowie Attribute, die in einem Modulgraphen enthalten sein können. Weiterhin sind alle Zugriffsoperationen auf einen Modulgraphen zugeschnitten. Mit anderen Worten heißt das, daß dieser abstrakte Datentyp "ModuleGraph" die Realisierung des durch das Graph-Ersetzungssystem "ModuleGraphOperations" auf konzeptioneller Ebene spezifierten Datentyps darstellt (vgl. Abschnitt 3.3.3).

In /Sc 86/ wird ausführlich erläutert, wie ausgehend von einer solchen Spezifikation durch Graph-Ersetzungssysteme eine zugehörige **Software-Architektur systematisch abgeleitet** werden kann. Dabei stellt sich u.a. heraus, daß die Modularisierungsüberlegungen beim Erstellen der Spezifikation unmittelbar auf die Softwarearchitektur übertragen werden können. Das bedeutet, daß die Realisierung des abstrakten Datentyps "**ModuleGraph**" aus verschiedenen Moduln zusammengesetzt ist. Wir wollen an dieser Stelle nur auf vier dieser Moduln eingehen; alles weitere findet man in /Sc 86/.
Zum **Eintragen** und **Löschen** von Graphinkrementen existiert ein Funktionsmodul "EditorOperations". Dieser Modul exportiert Ressourcen der Art "Einfügen/Löschen while-Anweisung", "Einfügen/Löschen Variablendeklaration" bzw. "Einfügen/Löschen von Bezeichnern".
Um sich über die aktuelle Umgebung eines Knotens im Modulgraphen **informieren** zu können, werden vom Funktionsmodul "MGQueries" eine Reihe von Anfrageoperationen exportiert. Hierzu zählt z.B. "Gib Markierung eines bestimmten Knotens", "Gib Nachfolger-/Vorgängerknoten über eine Kante mit einer bestimmten Markierung von einem bestimmten Knoten aus", "Gib Vaterknoten zu einem bestimmten Knoten", "Gib Anzahl auslaufender Kanten mit einer bestimmten Markierung von einem bestimmten Knoten an" usw.
Während von diesem Modul nur Ressourcen exportiert werden, mit denen man sich über den lokalen Kontext eines Knotens im Modulgraphen informieren kann, werden von dem Funktionsmodul "MGRoutines" Ressourcen exportiert, die **komplexere Anfragen** an den Modulgraphen ermöglichen. Hierzu zählt z.B. die Frage nach der Existenz zweier entfernt liegender Knoten im Modulgraphen. Alle diese Anfragen sind im Prinzip **kleinere Graphalgorithmen,** die durch eine Folge von Anfragen mit Hilfe der Ressourcen des Moduls "MGQueries" realisierbar sind. Da derartige Anfragen häufiger von verschiedenen anderen Moduln benötigt werden, werden sie durch den Modul "MGRoutines" allgemein zur Verfügung gestellt. Beispiele für exportierte Ressourcen sind "Gib nächsten Anweisungsknoten im Anweisungsteil", "Gib nächsten Bezeichnerknoten in einer Variablendeklaration" bzw. "Gib zu einem

Bezeichnerknoten in einem Ausdruck den Knoten der umgebenden Anweisung an".
Als letzten Bestandteil der Realisierung des abstrakten Datentyps "ModuleGraph" wollen wir den Funktionsmodul "MGAttributes" erwähnen, der es gestattet, modulgraphspezifische **Attribute** zu lesen bzw. zu schreiben. Hierzu gehören z.B. Ressourcen zum Lesen und Schreiben der vom Interpreter benötigten Attribute Code, Size bzw. Address.
Alle Ressourcen dieser vier Funktionsmoduln werden von dem abstrakten Datentyp "ModuleGraph" exportiert, der somit schematisch die folgende Gestalt hat:

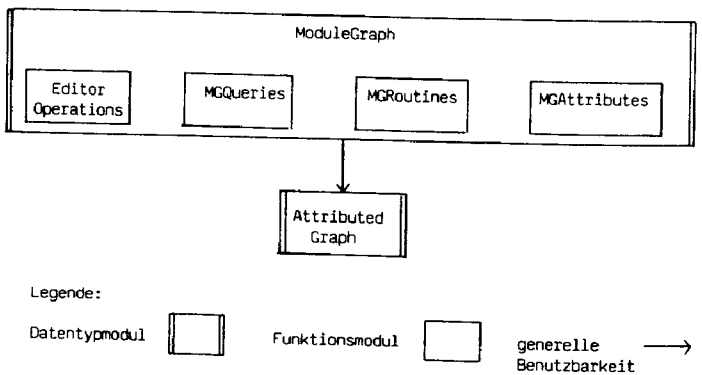


Abb. 9.1.1: Teilentwurf der Datenstruktur ModuleGraph

Die Darstellung einer weiteren Basiskomponente des IPSEN-Systems, der **Benutzerschnittstelle,** ist wesentlicher Bestandteil der Arbeit /Sc 86/. Wir wollen in diesem Abschnitt nur einige, für das weitere Verständnis notwendige Moduln erläutern.

Die Benutzerschnittstelle des IPSEN-Systems basiert wesentlich auf der Verwendung sich überlappender Fenster und der Benutzung einer Maus als zusätzlichem Eingabemedium. Aus diesem Grunde wurde im Rahmen des IPSEN-Projektes ein **allgemeines Fenstersystem** realisiert, das die Ausgabe und Manipulation sich beliebig überlappender Fenster erlaubt. Dieses Teilsystem wird durch den abstrakten Datentyp "WindowManager" repräsentiert.
Mit Hilfe dieses allgemeinen Fenstersystems wurde dann die IPSEN-spezifische Benutzerschnittstelle gestaltet. Hierzu existieren vier verschiedene Fenstertypen: Menü-, Nachrichten-, Eingabe- und Textfenster. In den zugehörigen Datentypmoduln ist z.B. verkapselt, welche Gestalt die Rahmen der einzelnen Fenster haben und welche Ein-/Ausgabeoperationen in einem Fenster erlaubt sind. So ist z.B. in einem **Eingabefenster** ("InputWindow") Lesen und Schreiben einer Zeichenkette erlaubt, während in einem **Textfenster** ("TextWindow") nur das Schreiben einer Zeichenkette und das Selektieren einer Position durch einen Mausklick erlaubt ist.

Da in einem Fenster häufig nur ein Teil eines zu bearbeitenden Textes dargestellt werden kann und der Benutzer deshalb die Möglichkeit hat, den Fensterinhalt zu rollen, wird eine weitere Datenstruktur benötigt, in der der gesamte Text gehalten werden kann. Diese Datenstruktur ist identisch mit der für die Realisierung des Unparsers benötigten Datenstruktur Textdokument (vgl. Abschnitt 6.3) und wird durch den Datentypmodul "TextDocument" verkapselt.

Das Teilsystem "IPSEN-specific I/O-System" enthält eine Reihe weiterer Moduln, die in /Sc 86/ ausführlich erläutert und motiviert werden.

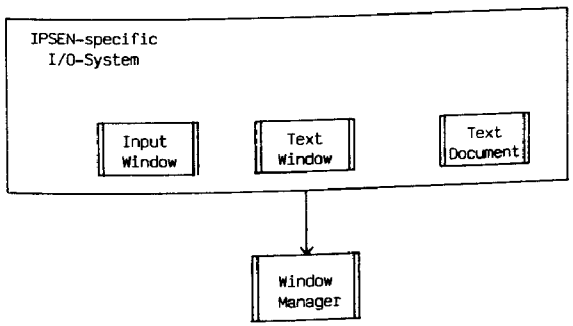


Abb. 9.1.2: Teilentwurf des IPSEN-specific I/O-System

Als weitere IPSEN-spezifische Datenstruktur benötigen wir für die Realisierung der Ausführung eine Datenstruktur, in der die Laufzeitdaten abgelegt werden können (vgl. Paragraph 7.1). Aus Effizienzgründen liegt diese Datenstruktur wie üblich im Hauptspeicher. Da nur ein einziger **Laufzeitdatenbereich** für die Ausführung benötigt wird, wird dieser durch ein abstraktes Datenobjekt mit dem Namen "RuntimeData" realisiert. Die Zugriffsoperationen dieses Datenobjektmoduls erlauben, den Inhalt einzelner Speicherzellen zu lesen bzw. zu schreiben und die für die Verwaltung des Laufzeitdatenbereichs benötigten Zeiger zu verschieben bzw. ihre Position zu lesen.

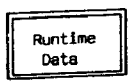


Abb. 9.1.3: Datenobjekt RuntimeData

Diese drei Komponenten "ModuleGraph", "IPSEN-specific I/O-System" und "RuntimeData" liegen gemeinsam in der **untersten Schicht** der Architektur des IPSEN-Systems (vgl. Abb. 9.4.2). Sie bilden die Grundlage für die Realisierung der in den darüberliegenden Schichten liegenden Moduln.

9.2 Transformationen

Oberhalb dieser Schicht von IPSEN-spezifischen Datenstrukturen liegt eine Schicht von Funktionsmoduln, durch die **Transformationen** zwischen den einzelnen Datenstrukturen realisiert werden. Hierbei treten insbesondere zwei Transformationen auf:

- Während der Ausführung eines Modulgraphen werden an Hand der im Modulgraphen vorliegenden Informationen die im Laufzeitdatenbereich liegenden Daten verändert.
- Um dem Benutzer den Inhalt des aktuell bearbeiteten Modulgraphen darzustellen, wird der Graph in eine textuelle Darstellung transformiert. Analog wird der durch die freie Eingabe eingegebene Quelltext in eine Modulgraphdarstellung transformiert.

Für diese Transformationen werden im einzelnen die folgenden Funktionsmoduln benötigt:

Während der **Ausführung** eines Modulgraphen hat der IPSEN-Benutzer nach jedem Ausführungsschritt, d.h. nach dem Ausführen jeder Anweisung, die Möglichkeit, durch Drücken einer speziellen Taste die Ausführung zu unterbrechen. Dies bedeutet, daß nach dem Ausführen jeder Anweisung die Kontrolle zunächst an die Instanz zurückgegeben werden muß, die derartige Benutzeraktionen empfangen kann. Wie wir noch sehen werden, liegt eine derartige Instanz für die Dialogkontrolle in einer höheren Schicht der IPSEN-Architektur. In der hier vorgestellten Schicht sollen nur Transformationen zwischen Datenstrukturen verkapselt werden. Aus diesem Grund wird von dem Funktionsmodul "MGInterpreter" im wesentlichen nur eine Ressource exportiert, die es ermöglicht, einen **einzigen Ausführungsschritt** durchzuführen. Als Ergebnis dieses Ausführungsschrittes liefert die Ressource die nächste auszuführende Anweisung und eine Statusvariable zurück. An Hand dieser Statusvariablen erkennt die diese Ressource aufrufende Stelle, ob während der Ausführung irgendwelche Besonderheiten aufgetreten sind.

Wie bereits im Paragraphen 7.1 erläutert, wird eine auszuführende Anweisung zunächst in eine P-Code-Darstellung übersetzt, wenn sie während der aktuellen Ausführung zum ersten Mal erreicht wird. Weiterhin werden bei der erstmaligen Ausführung einer Prozedur zunächst alle Werte der Attribute "Size" und "Address" neu berechnet. Alle derartigen **Attributberechnungen** werden durch Ressourcen des Funktionsmoduls "RuntimeAttributeGenerator" durchgeführt. Nach einer Berechnung des P-Codes für die aktuelle Anweisung kann dieser **P-Code** von einem entsprechenden "P-CodeInterpreter" **ausgeführt** werden, der ebenfalls durch einen Funktionsmodul realisiert wird. Hierbei wird ein P-Code-Attribut gelesen und der Inhalt des Laufzeitdatenbereichs entsprechend verändert. Da während der Berechnung des

P-Codes u.U. der Wert konstanter Ausdrücke zu ermitteln ist, benötigt der Funktionsmodul "RuntimeAttributeGenerator" zusätzlich Zugriff auf den Modul "P-CodeInterpreter", um diesen Wert unmittelbar berechnen zu lassen.

Da die beiden Funktionsmoduln "RuntimeAttributeGenerator" und "P-CodeInterpreter" nur zur Realisierung des Moduls "MGInterpreter" benötigt werden, haben wir an dieser Stelle ein Beispiel für eine **lokale Benutzbarkeitsbeziehung**. Da auf den Modulgraphen und die Laufzeitdaten auch andere Moduln zugreifen, liegt dort stets eine generelle Benutzbarkeitsbeziehung vor.

Damit hat der Teilentwurf für die Realisierung einer Anweisung die folgende Gestalt (vgl. hierzu auch /Sa 86/):

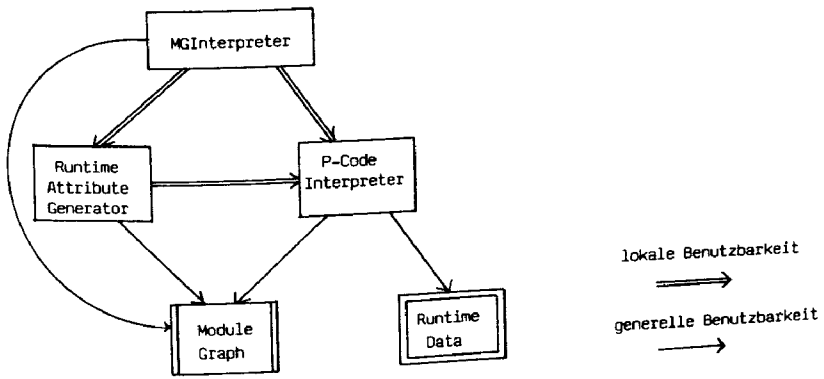


Abb. 9.2.1: Teilentwurf MGInterpreter

Für die Realisierung verschiedener Testunterstützungskommandos ist es nötig, den **Wert von Variablen**, die im Modulgraphen durch einen entsprechenden Teilgraphen repräsentiert werden, aus dem Laufzeitdatenbereich zu ermitteln. Hierzu muß zunächst im Modulgraphen die Adresse dieser Variablen ermittelt werden, bevor der Inhalt der entsprechenden Speicherzelle gelesen und ausgegeben werden kann. Falls im Laufzeitdatenbereich nur eine codierte Darstellung abgelegt ist (z.B. bei Komponenten eines Aufzählungstyps), muß anschließend aus dem Modulgraph die dort abgelegte zugehörige textuelle Darstellung ermittelt werden. Um den gesamten Speicherinhalt dem IPSEN-Benutzer anzeigen zu können, besteht zusätzlich die Möglichkeit, sich den Inhalt von Speicherzellen in einer beliebigen, existierenden dynamischen Verschachtelungstiefe anzusehen. Alle diese Transformationen werden durch den Funktionsmodul "RuntimeDataInspection" realisiert.

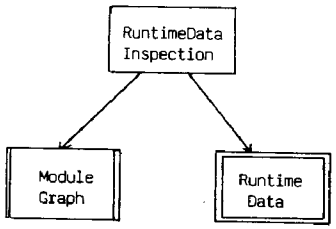


Abb. 9.2.2: Teilentwurf RuntimeDataInspection

Um den kontextfreien Anteil des Modulgraphen dem IPSEN-Benutzer in textueller Form auf dem Bildschirm anzeigen zu können, muß der Modulgraph zunächst durch einen **Unparser** in eine derartige textuelle Darstellung transformiert werden. Bei der Vorstellung des zugehörigen Entwurfs beschränken wir uns hier auf die Variante des Unparsers, die durch einen rekursiven Abstiegsalgorithmus realisiert ist (vgl. Paragraph 6.3). Ein Entwurf für die tabellengesteuerte Realisierung des Unparsers befindet sich in /Ha 85/.

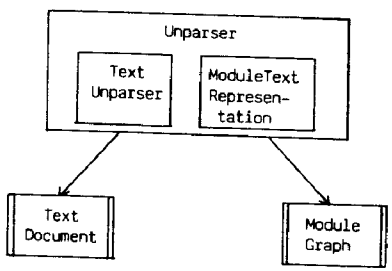


Abb. 9.2.3: Teilentwurf Unparser

Wie im Kapitel 6 bereits erläutert, wird der vom Unparser erzeugte Quelltext zunächst in einer Zwischendatenstruktur abgelegt, bevor er auf dem Bildschirm ausgegeben wird. Diese Zwischendatenstruktur wird durch den Datentypmodul "TextDocument" verkapselt. Der Unparser selbst wird durch die beiden Funktionsmoduln "TextUnparser" und "ModuleTextRepresentation" realisiert. Hierbei verkapselt der erste den **verwaltungstechnischen** Anteil des Unparsing-Prozesses und der zweite den **sprachspezifischen** Anteil. Zum verwaltungstechnischen Anteil gehört dabei insbesondere die Realisierung der Strategie, sukzessive den Startpunkt des Unparsing-Prozesses im Modulgraphen nach oben zu schieben (vgl. Paragraph 6.3). Zum sprachspezifischen Anteil gehört die Verkapselung der zu erzeugenden konkreten Syntax, die Gestaltung des Layouts (z.B. Zeilenumbruch, Einrückung) und die Reihenfolge, in der der Modulgraph durchlaufen wird. Diese Trennung von verwaltungstechnischem und sprachspezifischem Anteil hat den Vorteil, daß nur ein

Modul ausgetauscht werden muß, wenn z.B. zu einer anderen Graphstruktur Quelltext erzeugt werden soll. Der Teilentwurf des Unparsers ist in Abb. 9.2.3 dargestellt.

Der obige Unparser wird dazu benötigt, dem IPSEN-Benutzer den aktuell bearbeiteten Modul in textueller Form darzustellen. Während bei diesem Unparsing-Prozeß der Modulgraph die einzige Ausgangsdatenstruktur ist, benötigt man für die Realisierung einiger Testunterstützungskommandos einen Unparser, der zu einem Ausschnitt aus **Modulgraph und Laufzeitdatenbereich** eine textuelle Darstellung erzeugt. Ein Beispiel hierfür ist die Ausgabe eines Speicherauszugs, bei dem sowohl Variablennamen aus dem Modulgraphen erzeugt werden müssen, als auch ihr aktueller Wert aus den Laufzeitdaten ermittelt werden muß. Dieser Unparsing-Prozeß wird durch den Funktionsmodul "VariableInspector" realisiert. Auch hierbei wird die erzeugte Darstellung in einem Textdokument abgelegt. Um zu einer Variablen den Wert aus dem Laufzeitdatenbereich zu ermitteln, muß diese Variable zunächst in den Modulgraphen eingetragen werden. Dies geschieht durch Benutzung des Funktionsmoduls "M2Parser". Durch diesen Modul wird ein **Modula-2-Parser** realisiert, der einen in einem Textdokument abgelegten Ausschnitt eines Modula-2-Moduls syntaktisch analysiert und sukzessive an einer vorgegebenen Stelle in einen Modulgraphen einträgt (vgl. /Sl 86/). Dieser Modul wird insbesondere zur Realisierung der freien Eingabe benötigt. Nachdem eine Variable im Modulgraphen dargestellt ist, kann ihr Wert ermittelt werden, indem auf den oben erwähnten Modul "RuntimeDataInspection" zurückgegriffen wird. Dieser Teilentwurf hat somit folgende Gestalt:

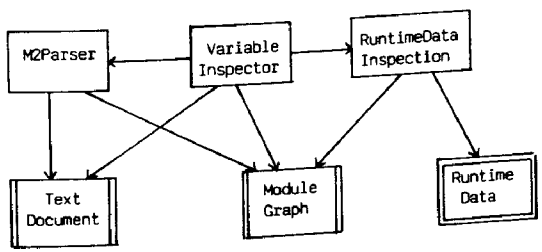


Abb. 9.2.4: Teilentwurf VariableInspector

In der in diesem Paragraphen erläuterten Transformationsschicht liegen weitere Funktionsmoduln, von denen wir zwei weitere nur kurz erwähnen wollen:

- Durch den Funktionsmodul "TextCutOut" wird ein Ausschnitt aus einem Textdokument in einem Textfenster ausgegeben.
- Durch den Funktionsmodul "TextI/O" wird in einem Eingabefenster ein Ausschnitt aus einem Textdokument dargestellt. Der Mikrodialog im Eingabefenster wird verwaltet und Veränderungen am Text im Eingabefenster im Textdokument nachgefahren.

Beide Funktionsmoduln sind weitere Bestandteile des "IPSEN-specific I/O-System" (vgl. /Sc 86/).

9.3 Ablaufsteuerung

Oberhalb der im letzten Paragraphen vorgestellten Schicht von Transformationsmoduln liegt eine Schicht von Moduln, durch die die **Ablaufsteuerung** der einzelnen Werkzeugaktivitäten realisiert ist.

Da durch nahezu alle Werkzeugaktivitäten eine Ein-/Ausgabe initiiert wird, kommt dem Funktionsmodul "ViewManager" eine entscheidende Bedeutung zu. Durch diesen Modul werden alle Funktionsmoduln koordiniert, die zur **Ein-/Ausgabe** benötigt werden. Die Existenz einer derartigen zentralen Kontrollinstanz ist aus verschiedenen Gründen notwendig, wie die folgenden Erläuterungen zeigen:

- Zu einem beliebigen Zeitpunkt während der Arbeit mit dem IPSEN-System können sich mehrere Fenster auf dem Bildschirm befinden, in dem einzelne werkzeugspezifische Ausgaben dem IPSEN-Benutzer angezeigt werden. Da den einzelnen Werkzeugen nicht bekannt ist, welche anderen Aktivitäten zuvor ausgeführt wurden, muß an einer Stelle verkapselt sein, **welche Fenster** bereits auf dem Bildschirm geöffnet sind, damit ein neu zu öffnendes Fenster an einer geeigneten Stelle geöffnet werden kann.

- Zu einem Graphen können sich zu derselben Zeit mehrere textuelle Darstellungen, d.h. "Views", auf dem Bildschirm in verschiedenen Textfenstern befinden. Zu jeder einzelnen textuellen Darstellung gehört ein Exemplar eines Textdokuments. Die Zuordnung **Textfenster zu Textdokument** muß bekannt sein, wenn der IPSEN-Benutzer z.B. den Text in einem Textfenster blättern will. Weiterhin muß der zugehörige Modulgraph bekannt sein, wenn durch einen Mausklick ein neues Inkrement selektiert werden soll.

- Bei der Ein-/Ausgabe eines Teils eines Modula-2-Moduls müssen die verschiedenen **Transformatoren** geeignet **koordiniert** werden: Bei der Ausgabe muß zunächst durch den Unparser Quelltext erzeugt und in einem Textdokument abgelegt werden. Aus demselben Textdokument muß mit Hilfe des Moduls "TextCutOut" ein Ausschnitt auf dem Bildschirm dargestellt werden. Bei der freien Eingabe wird zunächst durch den Modul "Text-I/O" der Inhalt eines Textdokuments modifiziert. Dieser Text wird dann vom Parser analysiert und in den Modulgraphen eingetragen. Anschließend muß die Quelltextdarstellung des veränderten Modulgraphen aktualisiert werden (vgl. /Sc 86/).

- **Nachrichtenfenster** sollen häufig in der Nähe der Darstellung eines Inkrements in einem bestimmten View angezeigt werden. Hierzu benötigt man die Information,

wo das zu diesem View gehörende Fenster auf dem Bildschirm liegt und an welcher Stelle in diesem Fenster sich die Darstellung des Inkrements befindet.

Zur Realisierung dieser verschiedenen Koordinierungsaufgaben wird ein lokales Gedächtnis benötigt, das sich in dem Datenobjektmodul "ViewTable" befindet. In dieser Tabelle wird im wesentlichen abgelegt, welche Textdokumente und Textfenster zu einem Modulgraphen derzeit existieren und welche Textfenster derzeit geöffnet sind.

Der Teilentwurf des "ViewManagers" hat die folgende Gestalt:

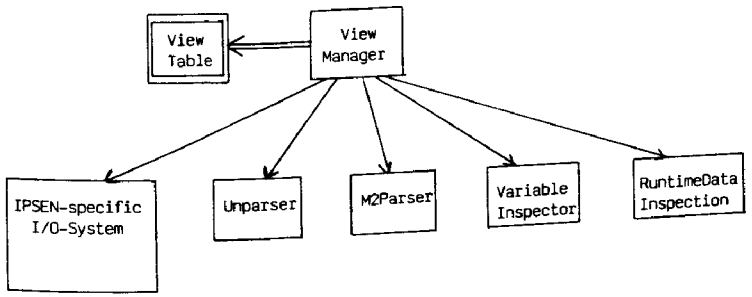


Abb. 9.3.1: Teilentwurf ViewManager

Da alle Werkzeuge eine Ein-/Ausgabe anstoßen, wird der Modul "ViewManager" von allen Moduln benutzt, die die "Ablaufsteuerung der einzelnen Werkzeugaktivitäten" realisieren. Bei der Realisierung dieser Moduln sind zwei Aspekte zu berücksichtigen:

- Die Zusammenfassung logisch zusammengehörender Aktivitäten zu Aktivitäten eines Werkzeugs kann auch bei der Einteilung in Moduln zur Ablaufsteuerung berücksichtigt werden. Die Ablaufsteuerung der Aktivitäten eines Werkzeugs kann jeweils in einem eigenen Funktionsmodul verkapselt werden.

- Andererseits ist zu beachten, daß bei einer Unterbrechung einer Werkzeugaktivität, an der eine Benutzerentscheidung erwartet wird, in der Regel auch Kommandos anderer Werkzeuge aktiviert werden können. Aus diesem Grunde muß bei einer solchen Unterbrechung einer Werkzeugaktivität die Ablaufkontrolle von dem aktuell aktiven Werkzeug an eine **zentrale Instanz** zurückgegeben werden, die auch auf Kommandos anderer Werkzeuge reagieren kann.

Speziell der zweite Aspekt hat erhebliche Auswirkungen auf die für die Ablaufsteuerung eines Werkzeugs zu wählende Realisierungsstrategie. Da die Ausführung einer Werkzeugaktivität zu einem bestimmten Zeitpunkt unterbrechbar und u.U. zu einem späteren Zeitpunkt fortsetzbar sein muß, muß bei einer Unterbrechung sämtliche, für die Fortsetzung notwendige Information geeignet

aufgehoben werden. Dies schließt insbesondere eine **rekursive Realisierung** der Ablaufsteuerung aus. Denn nach einer Unterbrechung auf einer beliebigen Rekursionstiefe ist es (nahezu) unmöglich, bei einer Fortsetzung denselben Ausführungszustand zu erreichen. In diesem Fall muß deshalb bei der Realisierung der einzelnen Aktivitäten explizit ein Keller verwaltet werden, in dem sämtliche bei einer Fortsetzung benötigten Informationen abgelegt werden. In einfacheren Fällen genügt an Stelle eines Kellers auch eine kleinere Tabelle mit entsprechenden Einträgen. Die Ablaufsteuerung der einzelnen Werkzeuge hat somit die folgende Gestalt:

Die Realisierung der Kommandos des Werkzeugs der statischen Analyse bestehen im wesentlichen aus der Aktivierung mehr oder weniger aufwendiger Algorithmen auf dem Modulgraphen. Aus diesem Grunde werden für die Realisierung in erster Linie Ressourcen der Moduln "MGQueries" und "MGRoutines" benötigt. Da während einer Analyseaktivität die Analyse auf andere Prozedurdeklarationen nur auf expliziten Benutzerwunsch ausgedehnt wird, muß bei Erreichen einer Prozedurdeklaration bzw. eines Prozeduraufrufs die Analyse zunächst unterbrochen werden und der IPSEN-Benutzer gefragt werden. Werden für die Fortsetzung der Analyse bereits ermittelte Informationen benötigt, werden diese aufgehoben. Da sich der Benutzer wiederholt für eine Ausdehnung der Analyse auf statisch tiefer liegende Prozedurdeklarationen entscheiden kann, eignet sich am besten eine kellerartige Datenstruktur für eine Verwaltung solcher Informationen. In jedem Kellerelement wird die zu einer bestimmten Prozedurdeklaration noch zu verarbeitende Information abgelegt. Hierzu dient der Datenobjektmodul "StaticAnalysisStack", der in einer lokalen Benutzbarkeitsbeziehung zum Funktionsmodul "StaticAnalysis" steht, durch den die Ablaufsteuerung des Werkzeugs der statischen Analyse realisiert wird. Für die Ausgabe der Analyseergebnisse benötigt dieser Modul Zugriff auf den Modul "ViewManager". Damit hat der Teilentwurf für dieses Werkzeug die folgende Gestalt:

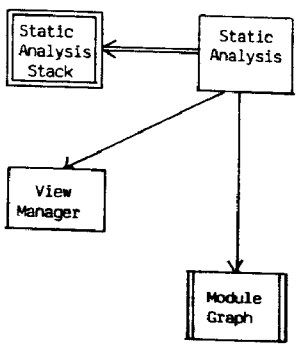


Abb. 9.3.2: Teilentwurf der Ablaufsteuerung der statischen Analyse

Auch für die Realisierung des Werkzeugs der Testvorbereitung wird nur Zugriff auf die beiden Moduln "ModuleGraph" und "ViewManager" benötigt. Im Gegensatz zur obigen statischen Analyse werden hierbei jedoch auch graphverändernde Ressourcen des Moduls "EditorOperations" aufgerufen, um eine während der Ausführung des Modulgraphen implizit zu aktivierende Tätigkeit in den Modulgraphen einzutragen (vgl. Kapitel 8). Für die Realisierung der Eingabe von Variablenbezeichnern wird auf die im Modul "ViewManager" enthaltene Ressource zur freien Eingabe zurückgegriffen. Ebenso wird der veränderte Modulgraph durch Aufruf einer Ressource des Moduls "ViewManager" dem Benutzer in textueller Darstellung angezeigt. Da die bisherigen Kommandos des Werkzeugs Testvorbereitung nicht unterbrochen werden können, wird keine lokale Tabelle (bzw. Keller) benötigt. Ansonsten ist der Teilentwurf für die Ablaufsteuerung des Werkzeugs Testvorbereitung analog zu Abb. 9.3.2.

Wir haben oben bereits erläutert, daß durch den Funktionsmodul "MGInterpreter" im wesentlichen die Ausführung einer einzelnen Anweisung realisiert ist. Die Aufgabe der **Ablaufsteuerung für die gesamte Ausführung,** realisiert durch den Funktionsmodul "InterpreterControl", besteht nun darin, die Ausführung einer einzelnen Anweisung solange anzustoßen, bis eine Unterbrechung auftritt.

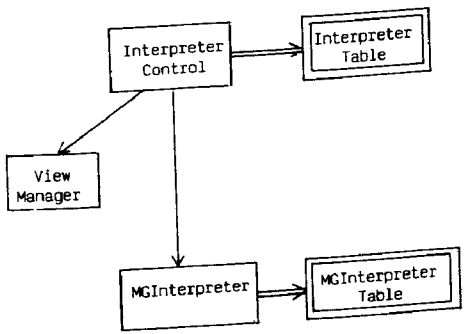


Abb. 9.3.3: Teilentwurf der Ablaufsteuerung der Ausführung

Je nach Art der Unterbrechung wird dem Benutzer durch Aufruf einer Ressource des Moduls "ViewManager" in einem Nachrichtenfenster zunächst eine Meldung ausgegeben (z.B. bei Laufzeitfehlern). Anschließend wird durch eine spezielle Statusvariable der aufrufenden Instanz mitgeteilt, warum und an welcher Stelle die Ausführung unterbrochen wurde. Damit bei einer Fortsetzung der Ausführung gewährleistet ist, daß die Ausführung an der unterbrochenen Stelle fortgesetzt wird, wird der Unterbrechungspunkt in einer Variablen im Datenobjekt "InterpreterTable" abgelegt. Außerdem muß während der Ausführung erkannt werden, wann ein automatischer Unterbrechungspunkt bzw. das Ende der Ausführung erreicht worden ist. Da die nächste auszuführende Anweisung innerhalb des Moduls "MGInterpreter"

bestimmt wird, wird diese Information dort in einem lokal benutzbaren Datenobjektmodul "MGInterpreterTable" abgelegt. Die aktuell ausgeführte Anweisung wird dem Benutzer auf dem Bildschirm durch eine besondere Darstellung dieser Anweisung sichtbar gemacht. Dieses Verschieben des Ausführungszeigers auf dem Bildschirm geschieht durch eine Ressource des Moduls "ViewManager".
Insgesamt hat der Teilentwurf für die Realisierung der Ablaufsteuerung der Ausführung somit die in Abb. 9.3.3 angegebene Gestalt.

Wie wir im Kapitel 8 erläutert haben, wird die Realisierung der Testunterstützungskommandos im wesentlichen auf die bereits im IPSEN-System vorhandenen Komponenten zurückgespielt. Das bedeutet, daß für die Realisierung des Funktionsmoduls "TestingSupport" Ressourcen der Moduln "ModuleGraph", "MGInterpreter" und "ViewManager" benutzt werden.

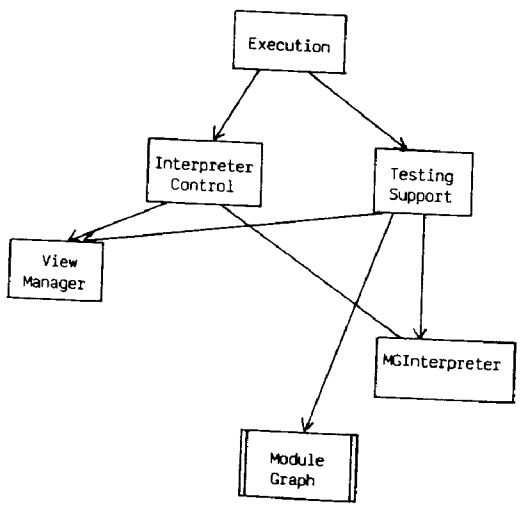


Abb. 9.3.4: Teilentwurf der Ablaufsteuerung der Ausführung mit Testumgebung

Ist während der Ausführung eines Modulgraphen die Testumgebung angeschaltet, wird bei Aufruf der Ressource des Moduls "InterpreterControl" die Ausführung unterbrochen, wenn ein Prozeduraufruf einer der zusätzlichen vordefinierten Prozeduren (z.B. Inspec, Dump) erreicht wird. Bei einem Aufruf einer derartigen implizit zu aktivierenden Tätigkeit muß anschließend eine entsprechende Ressource des Moduls "TestingSupport" aktiviert werden. Nach Beendigung dieser implizit auszuführenden Aktivität wird die Ausführung an der Anweisung fortgesetzt, die dem Aufruf dieser implizit zu aktivierenden Tätigkeit folgt. Eine derartige **Integration** zweier verschiedener Werkzeuge wird durch den Funktionsmodul "Execution"

realisiert, der sich in der Software-Architektur des IPSEN-Systems oberhalb der bisher vorgestellten Moduln befindet. In dieser Ebene befinden sich die Funktionsmoduln zur Ablaufsteuerung von Kommandos, durch die **verschiedene Werkzeugaktivitäten integriert** werden.
Der Teilentwurf für die Ausführung mit eingeschalteter Testumgebung hat somit die in Abb. 9.3.4 angegebene Gestalt.

Die IPSEN-spezifische Vorgehensweise bei der **Kommandoeingabe** wird durch den ebenfalls in dieser Schicht liegenden Funktionsmodul "StructuralCmdInput" realisiert. Bei Aufruf einer entsprechenden Ressource wird ein in der aktuellen Situation gültiges Kommando eingelesen und eine interne Kommandobezeichnung zurückgeliefert (vgl. /Sc 86/).

Die **gesamte Ablaufsteuerung** wird realisiert durch den ganz oben in der Architektur liegenden Funktionsmodul "IPSENControl". Hier wird ein eingelesenes Kommando empfangen und die entsprechende Werkzeugaktivität angestoßen. Falls eine Werkzeugaktivität unterbrochen werden kann, wird eine Statusvariable zurückgeliefert, die angibt, ob die Ausführung der Aktivität abgeschlossen oder unterbrochen worden ist. Falls durch die Ausführung der Werkzeugaktivität ein anderes Inkrement im Modulgraphen zum aktuellen Inkrement wird, wird das neue aktuelle Inkrement zurückgeliefert. Diese Informationen werden benötigt, um bei der Eingabe des nächsten Kommandos bzw. beim Anzeigen eines Kommandomenüs (dies ist enthalten in "StructuralCmdInput") die in der aktuellen Situation gültigen Kommandos bestimmen zu können. Auf diesen Aspekt wird in /Sc 86/ ausführlich eingegangen. Weiterhin muß der Ausführungszustand einer Werkzeugaktivität bekannt sein, wenn aufgrund eines vom Benutzer eingegebenen Kommandos (z.B. Editorkommandos) eine andere, z.Zt. unterbrochene Aktivität (z.B. Ausführung eines Modula-2-Moduls) explizit beendet werden muß.
Derzeitiger Zustand sowie eine Kennzeichnung des aktuell bearbeiteten Modulgraphen und des aktuellen Inkrements werden in dem Datenobjekt "ControlTable" abgelegt. Weiterhin wird hier vermerkt, ob derzeit die Testumgebung eingeschaltet ist.

9.4 Gesamtentwurf

Wir haben in diesem Kapitel die **Komponenten der Software-Architektur** des IPSEN-Systems vorgestellt, durch die die in dieser Arbeit erläuterten Werkzeugaktivitäten realisiert werden. Wir haben erläutert, daß die gesamte Architektur aus verschiedenen Schichten zusammengesetzt ist, von denen wir hier die Schicht der IPSEN-spezifischen Datenstrukturen, die Schicht der Transformationsmoduln und die Schicht für die Ablaufsteuerung beschrieben haben:

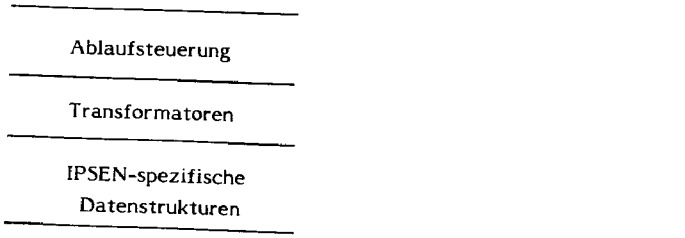


Abb. 9.4.1: Schichten der Software-Architektur des IPSEN-Systems

Die in den letzten Paragraphen in Teilentwürfen dargestellten Moduln werden in Abb. 9.4.2 noch einmal in einer Gesamtdarstellung zusammengefaßt.

In /Sc 86/ wird die Software-Architektur des gesamten IPSEN-Systems für den Bereich des Programmierens-im-Kleinen vorgestellt. Dabei wird gezeigt, daß sowohl die unterste als auch die oberste Schicht in mehrere Schichten aufgespalten werden kann. So kann z.B. die oberste Schicht weiter verfeinert werden in Schichten für die Ablaufsteuerung von einzelnen Werkzeugen, integrierten Werkzeugen und des gesamten IPSEN-Systems.

Weiterhin wird in /Sc 86/ erläutert, daß bei der Entwicklung dieser Software-Architektur von uns großes Gewicht darauf gelegt wurde, nicht ausschließlich eine Architektur für den Aufgabenbereich des Programmieren-im-Kleinen-Anteils im IPSEN-System zu entwickeln. Es ist unsere Hoffnung, daß viele der **Entwurfsentscheidungen unmittelbar übernommen** werden können, wenn die Architektur für die **anderen Aufgabenbereiche** im IPSEN-Projekt entwickelt wird. Diese Hoffnung hat sich mittlerweile bei der Entwicklung eines Editors für den Programmieren-im-Großen-Anteil des IPSEN-Systems bereits bestätigt (vgl. /Le 85/). Außerdem hat sich herausgestellt, daß sich diese Architektur auch unmittelbar auf andere, analoge Softwaresysteme übertragen läßt. So wurde diese Architektur ebenso erfolgreich auf die Entwicklung eines ebenfalls in unserer Arbeitsgruppe im Rahmen von Diplomarbeiten entstandenen strukturbezogenen Textsystems übertragen (/Er 86/, /Ti 86/).

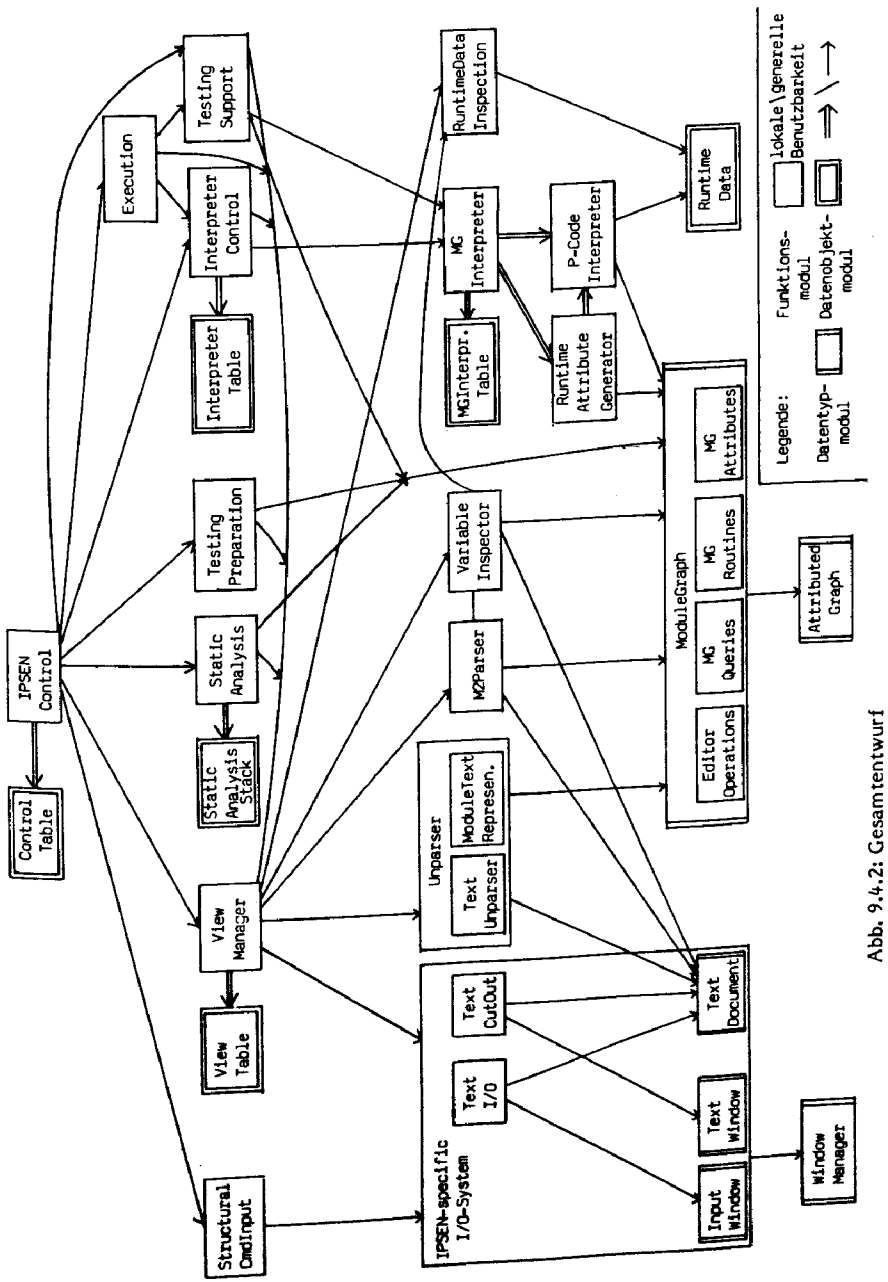


Abb. 9.4.2: Gesamtentwurf

9.5 Bemerkungen zur Implementierung

Ein wichtiges Ziel des IPSEN-Projekts ist neben der Entwicklung neuer konzeptioneller Ideen diese Überlegungen unmittelbar auf eine konkrete **Prototyp-implementierung** anzuwenden. Für die erste Phase des Projekts hieß das konkret, die Werkzeuge des Programmieren-im-Kleinen-Anteils zu implementieren und in einer integrierten Umgebung zum Laufen zu bringen. Hierbei war es nicht unser Ziel, ein vermarktungsfähiges Softwareprodukt zu erstellen. Es sollte vielmehr die **Machbarkeit** der konzeptionellen Ideen an Hand einer konkreten Implementierung gezeigt werden.

Kurz vor Ende der ersten fünf Jahre des IPSEN-Projekts dürfen wir sagen, daß wir unser Ziel erreicht haben! Nach einem etwas zähen Anfang in der Implementierungsphase stehen wir derzeit in der Integrationsphase, in der nacheinander die Implementierungen der einzelnen Werkzeuge in eine Gesamtumgebung eingefügt werden. Die meisten Schwierigkeiten und auch der zähe Anfang der Implementierungsphase hatten ihre Ursache weniger in Schwächen der konzeptionellen Vorüberlegungen als vielmehr in mangelnder Hardware- und Software-Aussstattung.

Die Implementierung wurde Anfang 1984 mit der Implementierung der Basiskomponenten auf einem Rechner (TA 1630 von Triumph-Adler) begonnen, dessen Grenze von 64 kB-Hauptspeichergröße für Code und Daten schnell erreicht war. Die dadurch bedingte Verwendung von Overlay-Konzepten brachte vor allem bei der zu der Zeit laufenden Implementierung des Graphenspeichers (/Ba 84/) viele zusätzliche Probleme.

Ende 1984 standen dann zwei IBM XT mit 512 kB Hauptspeicher zusammen mit einem Modula-2-Entwicklungssystem zur Verfügung (Volition Systems /GI 82/). Nach einer Portierung des ursprünglich in Pascal geschriebenen Graphenspeichers auf diesen neuen Rechner wurden im Laufe des Jahres 1985 die übrigen Komponenten des IPSEN-Systems codiert und getestet. Da durch den Compiler des Volition-Systems Modula-2-Programme zunächst in P-Code übersetzt werden, war die Ausführungszeit der übersetzten Programme sehr hoch. Aus diesem Grunde haben wir uns im Spätherbst 85 entschieden, für die weitere Programmentwicklung ein anderes Entwicklungssystem zu benutzen. Der Aufwand dieser Umstellung war nicht allzu hoch, da die Entwicklungsmaschine und die Programmiersprache identisch blieben. Dieses Entwicklungssystem ist das "Modula-2 Software Development Systems (M2SDS)" der Firma Interface Technologies (/IT 85/). Der hierin enthaltene Compiler erzeugt unter dem Betriebssystem MS-DOS ausführbaren Maschinencode.

Allerdings lagen auch bei dieser Konfiguration die Antwortzeiten des IPSEN-Systems immer noch in unzumutbaren Größen. Dies hat sich erst geändert, seitdem uns mit Beginn dieses Jahres ein IBM AT/02 zur Verfügung steht, auf den wir derzeit die

gesamte Implementierung übertragen.

Da bei der jetzigen Implementierung nur die Machbarkeit gezeigt werden sollte, haben wir den **Leistungsumfang** der einzelnen Werkzeuge **eingeschränkt.** Insbesondere ist beim syntaxgestützten Editor und Hybrid-Interpreter im Moment nur die Verarbeitung von Standarddatentypen vorgesehen.

Der jetzige Prototyp umfaßt ca. 45.000 Zeilen Quelltext. Hierin enthalten sind alle selbstentwickelten Basiskomponenten (Fenstersystem, Graphenspeicher) sowie die Implementierung sämtlicher Transformationsbausteine. Alle diese Komponenten sind getestet und bereits in einem Programm integriert. Weiterhin sind die Werkzeuge syntaxgestützter Editor, statische Analyse und Testvorbereitung getestet und Schritt für Schritt integriert worden. Zur Zeit wird die Ausführung (InterpreterControl) integriert, so daß anschließend noch das Werkzeug Ausführung und Test (Execution bzw. TestingSupport) getestet bzw. integriert werden muß. Es ist unser Ziel, bis zum Frühsommer 1986 eine lauffähige Prototypimplementation für den Programmieren-im-Kleinen-Anteil vorliegen zu haben. Hierbei soll dann auch nahezu der gesamte Sprachumfang von Modula-2 unterstützt werden.

Die gesamte Implementierung wurde in erster Linie betreut und koordiniert von C. Lewerentz, W. Schäfer und dem Autor dieser Arbeit. Alle drei haben auch große Teile der Implementierung selbst erstellt. Vom Autor dieser Arbeit wurden insgesamt ca. 8.500 Zeilen Quelltext geschrieben, die sich wie folgt aufteilen (vgl. hierzu Abb. 9.4.2): die Moduln "MGQueries" und "MGRoutines" ca. 2.500 Zeilen, die rekursive Abstiegsvariante des Unparsers ca. 2.500 Zeilen, der Modul "VariableInspector" ca. 600 Zeilen, der "ViewManager" ca. 900 Zeilen, das Werkzeug der statischen Analyse ca. 1.300 Zeilen und die z.T. noch in der Testphase befindlichen Moduln "Execution", "TestingPreparation" und "TestingSupport" ca. 700 Zeilen Quelltext.
Eine kommentierte Fassung dieses Quelltexts liegt als interner Projektbericht vor.

10 Zusammenfassung und Ausblick

Wir haben in dieser Arbeit gezeigt, daß eine graphartige Datenstruktur, der sogenannte **Modulgraph**, eine geeignete zentrale Datenstruktur für die Realisierung aller Werkzeuge im Bereich des Programmierens-im-Kleinen innerhalb einer Software-Entwicklungsumgebung darstellt. Hierzu haben wir zunächst erläutert, wie aus einer normierten Beschreibung der kontextfreien Syntax einer Programmiersprache ein Grundgerüst eines Modulgraphen systematisch abgeleitet werden kann. Die dabei entstehenden abstrakten Syntaxgraphen wurden dann um Knoten und Kanten ergänzt, um weitere strukturelle Beziehungen, z.B. kontextsensitive Abhängigkeiten in einem Programm, innerhalb desselben Datenmodells darstellen zu können. Zusätzliche nicht-strukturelle Informationen, die für die Realisierung der einzelnen Werkzeuge benötigt werden, wurden in Knotenattributen abgelegt.

Um in der Spezifikationsphase die Funktionalität der einzelnen Werkzeuge beschreiben zu können, muß die Bedeutung von Zugriffsoperationen auf diese gemeinsame, sehr komplexe Datenstruktur Modulgraph festgelegt werden. Hierzu haben wir in dieser Arbeit die Methode des "**Graph Grammar Engineering**" vorgestellt, mit der es möglich ist, eine derartige Spezifikation systematisch und schrittweise zu erstellen. Grundidee dieser Vorgehensweise ist, für jedes Werkzeug getrennt durch ein Graph-Ersetzungssystem zu beschreiben, wie bei einer beliebigen Werkzeugaktivität im Modulgraphen abgelegte, werkzeugspezifische Informationen zu aktualisieren ist. Diese einzelnen Graph-Ersetzungssysteme wurden dann in einem zweiten Schritt in einem weiteren Graph-Ersetzungssystem koordiniert, das dann eine formale Spezifikation der Klasse der Modulgraphen darstellte.

Die Methode des Graph Grammar Engineering wurde im Rahmen des IPSEN-Projektes bisher angewandt, um im Bereich des Programmierens-im-Kleinen einen **syntaxgestützten Editor** für die Programmiersprache Modula-2 (/Sc 86/) bzw. Ada (/Do 84/) zu spezifizieren. Außerdem wurde sie auch bei der Spezifikation des syntaxgestützten Editors im Bereich des Programmierens-im-Großen eingesetzt (/LN 84/).

Im weiteren Verlauf dieser Arbeit wurde für verschiedene Werkzeuge des Programmieren-im-Kleinen-Anteils konkret erläutert, wie einige, zuvor im Stil einer Anforderungsdefinition vorgestellte Kommandos unter Benutzung dieser gemeinsamen Datenstruktur realisiert werden können. Hierbei wurden exemplarisch Kommandos des Werkzeugs der **statischen Analyse** diskutiert, durch die es für den IPSEN-Benutzer möglich ist, Informationen über den im aktuell bearbeiteten Modula-2-Modul verkapselten Kontroll- und Datenfluß zu erhalten. Weiterhin wurden Kommandos zum Ausführen und Testen eines Modula-2-Moduls vorgestellt, mit denen es dem IPSEN-Benutzer möglich ist, der aktuellen Situation angemessene **Testumgebungen** aufzubauen. Die Beschreibung der Realisierung aller dieser Kommandos geschah weiterhin auf konzeptioneller Ebene, in dem erläutert wurde, welche

werkzeugspezifischen Informationen zusätzlich im Modulgraphen abzulegen und zu verwalten sind.

Für die Realisierbarkeit umfangreicher, komfortabler Testunterstützungsmöglichkeiten ist im IPSEN-System in erster Linie der Einsatz eines **Hybrid-Interpreters** für die Ausführung eines Modula-2-Moduls verantwortlich. Dieses Werkzeug besteht im wesentlichen aus zwei Interpretern, die abwechselnd die im Modulgraphen abgelegten Kontrollstrukturen bzw. in eine maschinennähere Form übersetzte Darstellung von Zuweisungsanweisungen und Ausdrücken ausführen. Das Konzept dieses Hybrid-Interpreters wurde in dieser Arbeit vorgestellt.

Um Modulgraphen auf dem Bildschirm darstellen zu können, müssen sie zunächst in eine für den Benutzer lesbare Form übersetzt werden. Hierzu wurden in dieser Arbeit zwei verschiedene Methoden vorgestellt, um eine **textuelle Darstellung** aus der Modulgraphdarstellung zu erzeugen. Es wurde gezeigt, daß durch den Einsatz einer weiteren, ebenso graphartigen Zwischendatenstruktur hohe Anspüche an Layout und Formatierung des zu erzeugenden Quelltextes befriedigt werden können.

Am Schluß der Arbeit wurde ein Überblick über den Teil der **Software-Architektur** des IPSEN-Systems gegeben, der für die Realisierung der in dieser Arbeit vorgestellten Werkzeuge relevant ist.

Durch /Sc 86/ und diese Arbeit wurden bisher im wesentlichen nur die offenen Fragen im Aufgabenbereich des Programmierens-im-Kleinen untersucht. Im Vergleich zu dem angestrebten Gesamtziel des IPSEN-Projekts scheint dies ein sehr kleiner Schritt gewesen zu sein. Andererseits war es bei allen bisher durchgeführten Untersuchungen stets unser Ziel, die jeweilige Fragestellung auf **systematische** Art und Weise zu lösen und zugrundeliegende **prinzipielle Eigenschaften** herauszuarbeiten. Aus diesem Grunde sind wir der festen Überzeugung, daß sich viele der in /Sc 86/ bzw. in dieser Arbeit vorgestellten Vorgehensweisen übertragen lassen, wenn die Werkzeuge der anderen Aufgabenbereiche realisiert werden sollen. Das gilt natürlich auch für zahlreiche denkbare Erweiterungen der in dieser Arbeit vorgestellten Werkzeuge. Hierzu wurden bereits eine Reihe von Möglichkeiten angesprochen. Weitere sinnvolle Ergänzungen des Leistungsumfangs der einzelnen Werkzeuge werden deutlich werden, wenn die ersten Erfahrungen im Umgang mit der Prototypimplementierung des IPSEN-Systems vorliegen.

Neue, interessante Probleme werden zu lösen sein, wenn es um die Realisierung von Werkzeugen geht, durch die Tätigkeiten aus verschiedenen Aufgabenbereichen **integriert und koordiniert** werden. An dieser Stelle seien als Beispiel nur umfangreiche statische Analysen genannt, durch die über Modulgrenzen hinweg bestimmte Eigenschaften untersucht werden. Ein weiteres Beispiel ist die Ausführung ganzer Software-Systeme, für die der in dieser Arbeit vorgestellte Hybrid-

Interpreter geeignet zu erweitern ist. So weit es bisher möglich war, wurde dies durch eine entsprechende Laufzeitdatenstruktur für die Verwaltung verschiedener modullokaler Datenbereiche schon vorgesehen (vgl. /Sa 86/).

Die in dieser Arbeit vorgestellte Spezifikationsmethode des Graph Grammar Engineering ist gut geeignet, alle Zugriffsoperationen zur **Modifikation einer graphartigen Datenstruktur** zu spezifizieren. Dies hat sich insbesondere bei der Anwendung dieser Methode zur Spezifikation verschiedener Editoren erwiesen. Für die Spezifikation von Werkzeugen, die im Prinzip nur lesenden Zugriff auf diese graphartige Datenstruktur haben, muß überlegt werden, inwieweit diese Spezifikationsmethode ergänzt werden kann. Bei der Spezifikation derartiger Werkzeuge werden neben der graphartigen Datenstruktur häufig weitere Datenstrukturen benötigt, die während der Ausführung einer Werkzeugaktivität verändert werden. Als Beispiel haben wir in dieser Arbeit die Laufzeitdaten bei der Vorstellung des Interpreters kennengelernt. Die naive Idee, alle zusätzlichen Datenstrukturen ebenfalls als graphartig aufzufassen, scheint auf die Dauer nicht tragbar zu sein. In diesem Punkt sollte überlegt werden, inwieweit andere, u.U. auch operationelle Spezifikationsmethoden mit der Methode des Graph Grammar Engineering kombiniert werden können. Das Ziel sollte sein, eine Spezifikationsmethode zu entwickeln, mit der es möglich ist, alle Teile eines Softwaresystems zu spezifizieren, die auf durchaus unterschiedlichen Datenstrukturen operieren, aber deren **zentrale Datenstrukturen Graphen** sind.

Literatur

/Al 83/ Allison, L.: Syntax Directed Program Editing, in: Software - Practice and Experience, Vol. 13, 453-465

/AU 79/ Aho, A.V./Ullmann, J.D.: Principles of Compiler Design, Reading: Addison-Wesley

/Ba 84/ Brandes, Th.: Entwurf und Implementierung eines Graphenspeichers, Diplomarbeit, Universität Dortmund, Abt. Informatik

/BL 85/ Brandes, Th./Lewerentz, C.: GRAS - A Nonstandard Data Base System within a Programming Support Environment, in: /Pr 85a/, 113-121

/Br 84/ Brendel, W.: Funktionaler Entwurf und automatische Synthese hochintegrierter Schaltkreise, Dissertation, Universität Erlangen-Nürnberg, Arbeitsberichte des IMMD, Bd. 17, Nr. 4

/Di 72/ Dijkstra, E.W.: Notes on Structured Programming, in: Dahl/Dijkstra/Hoare: Structured Programming, Academic Press

/DH 84/ Donzeau-Gouge, V./Huet, G./Kahn, G./Lang, B.: Programming Environments Based on Structured Editors: The Mentor Experience, in: Barstow, D.R. et al. (eds.): Interactive Programming Environments, McGraw-Hill

/DK 84/ Donzeau-Gouge, V./Kahn, G./Lang, B./Melese, B.: Document structure and modularity in Mentor, in: /He 84/, 141-148

/DL 84/ Dal Cin, M./ Lutz, J./ Risse, Th.: Programmierung in Modula-2, Stuttgart: Teubner

/Do 84/ Dorka, H.-J.: Spezifikation eines syntaxgesteuerten Editors für die Programmiersprache Ada, mit einer Graphgrammatik, Diplomarbeit, Ruhr-Universität Bochum, Abt. Mathematik

/EG 83/ Engels, G./Gall, R./Nagl, M./Schäfer, W.: Software Specification Using Graph Grammars, in: Computing 31, 317-346, Wien: Springer

/EL 86/ Engels, G./Lewerentz, C./Nagl, M./Schäfer, W.: On the Structure of an Incremental and Integrated Software Development Environment, in: Proc. of the 19th Hawaii Intern. Conference On System Sciences

/EM 85/ Ehrig, H./Mahr, B.: Fundamentals of Algebraic Specification 1, EATCS Monographs on Theoretical Computer Science, Berlin : Springer

/ES 85a/ Engels, G./Schäfer, W.: Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment, in Ehrig et al (eds.): Formal Methods and Software Development, Proc. of TAPSOFT Conference, Berlin, LNCS 186, 179-193, Berlin: Springer

/ES 85b/ Engels, G./Schäfer, W.: The Design of an Adaptive and Portable Programming Support Environment, in Valle, G./Bucci, G. (eds.): Proc. of the International Computing Symposium 1985, Florence, Italy, 297-308, Amsterdam: North-Holland

/Er 86/ Erdtmann, F.: Aufbau und Verwaltung einer internen Datenstruktur für Texte in strukturbezogenen Editoren, Diplomarbeit, Universität Osnabrück, Fachbereich Mathematik/Informatik, in Vorbereitung

/Fr 83/ Fritzon, P.: Adaptive Prettyprinting of Abstract Syntax applied in Ada and Pascal, Research Report LiTH-IDA-R-83-08, Linköping University

/Fr 84/ Fritzson, P.: Preliminary Experience from the DICE System, A Distributed Incremental Compiling Environment, in: /He 84/, 113-123

/Fr 85/ Fritzson, P.: The Architecture of an Incremental Programming Environment and some Notions of Consistency, in: /Pr 85a/, 64-79

/Ga 83/ Gall, R.: Formale Beschreibung des inkrementellen Programmierens-im-Grossen mit Graph-Grammatiken, Dissertation, Universität Erlangen, Arbeitsberichte des IMMD, Bd. 16, Nr. 1

/Ga 85/ Garlan, D.: A Framework for Unparse Specification Languages, Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh

/Ge 83/ Geissmann, L.B.: Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith, Dissertation, ETH Zürich, Diss. ETH No. 7286

/Gl 82/ Gleaves, R.: Modula-2 on the UCSD Pascal System, Del Mar: Volition Systems

/Go 73/ Goldstein, I.: Pretty-printing, converting list to linear structure, Artificial Intelligence Laboratory Memo, No. 279, M.I.T., Cambridge, Mass.

/Go 84/ Göttler, H.: Implementation of Attributed Graph-Grammars, in: Proceedings of the WG '84, Intern. Workshop on Graphtheoretic Concepts in Computer Science, Berlin. Linz: Trauner

/GR 83/ Goldberg, A./Robson, D.: Smalltalk-80, The Language and its Implementation, Reading: Addison-Wesley

/GT 78/ Goguen, J.A./ Thatcher, J.W./ Wagner, E.G./ Wright, E.G.: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in Yeh, R.T. (ed.): Current Trends in Programming Methodology, Vol. IV, Englewood Cliffs: Prentice Hall

/Ha 71/ Hansen, W.J.: User Engineering Principles for Interactive Systems, AFIP FJcc, 39, 523-532

/Ha 82/ Habermann, N. et al.: The Second Compendium of Gandalf Documentation, Technischer Bericht, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh

/Ha 85/ Haverkamp, H.: Quelltexterzeugung durch einen Unparser innerhalb einer Software-Entwicklungsumgebung, Diplomarbeit, Universität Dortmund, Abt. Informatik

/He 77/ Hecht, Matthew S.: Flow Analysis of Computer Programs, New York: North-Holland

/He 84/ Henderson, P. (ed.): Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, ACM SIGPLAN Notices Vol. 19, No. 5

/Hu 81/ Huenke, H. (Ed.): Software Engineering Environments, Amsterdam: North-Holland

/IT 85/ Modula-2 Software Development System, User's Guide, Interface Technologies Corporation

/Ja 86/ Janning, Th.: Integration in IPSEN, Diplomarbeit, Universität Osnabrück, Fachbereich Mathematik/Informatik, in Vorbereitung

/JF 82/ Johnson, G.F./Fischer, C.N.: Non-syntactic Attribute Flow in Language-Based Editors, in: Proc. of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico

/KE 82/ Kaiser, G.E./Ellison, R./Garlan, D.R./Notkin, D.S./Popovich, St.: Gandalf Environment User's Manual and Tutorial, Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh

/Ki 79/ Kimm, R. et al.: Einführung in Software-Engineering, Berlin: Walter de Gruyter

/KT 85/ Kirslis, P.A./Terwilliger, R.B./Campbell, R.H.: The SAGA Approach to Large Program Development in an Integrated Modular Environment, Dept. of Computer Science, University of Illinois at Urbana-Champaign, in: /Pr 85a/, 44-53

/Le 85/ Lewerentz, C.: Inkrementelles Programmieren im Großen: Syntaxgestützte Erstellung und Wartung von Systemspezifikationen, in: Morgenbrod/ Remmele (eds.): Entwurf großer Software-Systeme, Bericht German Chapter ACM, Bd. 19, 68-93, Stuttgart: Teubner

/Le 86/ Lewerentz, C.: Entwurf und Implementierung eines syntax-gesteuerten Editors für Software-Architekturen, Universität Osnabrück, eingereicht zur Veröffentlichung

/LN 84/ Lewerentz, C./ Nagl, M.: A Formal Specification Language for Software Systems Defined by Graph Grammars, in U. Pape (Ed.): Proc. WG '84, Linz: Trauner

/LN 85/ Lewerentz, C./ Nagl, M.: Incremental Programming in the Large: Syntax-aided Specification Editing, Integration and Maintenance, in Proc. of the 18th Hawaii International Conference on System Sciences, Vol. 2, 638-649

/Me 82/ Medina-Mora, R.: Syntax-Directed Editing: Towards Integrated Programming Environments, PhD Thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh

/Mi 81/ Mikelsons, M.: Prettyprinting, ACM Transactions on Program Languages and Systems, Vol. 2, No. 4

/ML 84/ Madhavji, N./ Leoutsarakos, N.: A Technique for Folding Program Structures, Technischer Bericht SCOS-84.18, School of Computer Science, McGill University, Montreal

/ML 85/ Madhavji, N./Leoutsarakos, N./Vouliouris, D.: Software Construction Using Typed Fragments, McGill University, Montreal, in: Ehrig, H. et al. (Eds.): Formal Methods and Software Development, TAPSOFT Conference, Berlin, LNCS 186, 163-178, Berlin: Springer

/MN 84/ Meyer, B./Nerson, J.-M./Ko, S.H.: Showing Programs on a Screen, Technischer Bericht TRCS84-08, Dept. of Computer Science, University of California, Santa Barbara

/MJ 81/ Muchnik, St.S./ Jones, N.D.: Program Flow Analysis: Theory and Applications, Englewood Cliffs: Prentice-Hall

/Na 79/ Nagl, M.: Graph-Grammatiken. Theorie, Anwendungen, Implementierung, Braunschweig: Vieweg

/Na 80/ Nagl, M.: GRAPL - A Programming Language for Handling Dynamic Problems on Graphs, in U. Pape (ed.): Discrete Structures and Algorithms, 25-45, München: Hanser

/Na 81/ Nagl, M.: Application of Graph Rewriting to Optimization and Parallelization of Programs, in Computing, Suppl. 3, 105 - 124 (1981), Wien: Springer Verlag 1981

/Na 85a/ Nagl, M.: An Incremental Programming Support Environment, in Computer Physics Communications 38, 245-276, Amsterdam: North-Holland

/Na 85b/ Nagl, M.: Graph Technology Applied to a Software Project, in: Rozenberg, G./Salomaa, A.: The Book of L, 303-312, Berlin: Springer

/Op 80/ Oppen, D.: Prettyprinting, ACM Transactions on Program Languages and Systems, Vol. 2, No. 4

/Pe 82/ Pemberton, St./ Daniels, M. : Pascal Implementation: The P4 Compiler/ Compiler and Assembler Interpreter, Chichester: Ellis Horwood Ltd.

/Pr 85a/ Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large, Cape Cod, Mass.

/Pr 85b/ Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Seattle, ACM SIGPLAN Notices Vol. 20, No. 7

/Qu 86/ Quante, R.: Strukturbezogener Editor und Interpreter für Graph-Grammatiken, Diplomarbeit, Universität Dortmund, Abt. Informatik, in Vorbereitung

/Re 85/ Reiss, St.P.: PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering, SE-11, 3, 276 - 285

/Re 82a/ Reps, T.: Generating Language-based Environments, Technical Report 82-514, Cornell University, Ithaca

/Re 82b/ Reps, T.: Optimal-time Incremental Semantic Analysis for Syntax-Directed Editors. Ninth Annual Symposium on Principles of Programming Languages

/Re 84/ Reiss, St.P.: Graphical Program Development with PECAN Program Development Systems, in: /He 84/, 30-41

/RT 83/ Reps, T./Teitelbaum, T./Demers, A.: Incremental Context-dependent Analysis for Language-based Editors, ACM Transactions on Programming Languages and Systems, Vol. 5, 449-477

/Sa 86/ Sandbrink, A.: Entwurf und Implementierung eines Test- und Laufzeitunterstützung berücksichtigenden Interpreters, Diplomarbeit, Fachbereich Mathematik/Informatik, Universität Osnabrück, in Vorbereitung

/Sc 74/ Schneider, H.J.: Syntax-Directed Description of Incremental Compilers, LNCS 26, 192-201, Berlin: Springer

/Sc 77/ Schneider, H.J.: Graph Grammars, LNCS 56, 314-331, Berlin: Springer

/Sc 86/ Schäfer, W.: Eine integrierte Softwareentwicklungsumgebung: Konzepte, Entwurf und Implementierung, Dissertation, Universität Osnabrück, Fachbereich Mathematik/Informatik, in Vorbereitung

/Sl 86/ Schleef, U.: Ein inkrementell arbeitender Parser als Teil eines syntaxgesteuerten Editors, Diplomarbeit, Universität Osnabrück, Fachbereich Mathematik/Informatik, in Vorbereitung

/Sn 85/ Snelting, G.: Experiences with PSG - Programming System Generator, in Ehrig, H. et al. (Eds.): Formal Methods and Software Development, TAPSOFT Conference, Berlin, LNCS 186, 148-162, Berlin: Springer

/Ti 86/ Tillmann, P.: Erzeugen und Verwalten der physischen Repräsentation von Texten in strukturbezogenen Editoren, Diplomarbeit, Universität Osnabrück, Fachbereich Mathematik/Informatik, in Vorbereitung

/TM 81/ Teitelman, W./Masinter, L.: The Interlisp Programming Envronment, in Computer, Vol. 14, No. 4

/TR 81a/ Teitelbaum, T./Reps, Th.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, in: CACM, Vol. 24, No. 9, 563-573

/TR 81b/ Teitelbaum, T./Reps, T./Horwitz, S.: The Why and Wherefore of the Cornell Program Synthesizer, in ACM SIPLAN Notices, Vol. 16, No. 6

/Wi 74/ Wirth, N.: On the Composition of Well-Structured Programs, in: Computing Surveys, Vol. 6, No. 4

/Wi 81/ Wirth, N: The Personal Computer Lilith, Institut für Informatik, ETH Zürich

/Wi 82/ Wirth, N.: Programming in Modula-2, Berlin: Springer

/Za 83/ Zadeck, F.K.: Incremental Data Flow Analysis in a Structured Program Editor, PhD dissertation, Mathematical Sciences Department, Rice University

/Za 84/ Zadeck, F.: Incremental Data Flow Analysis in a Structured Program Editor, in: Proceedings of the ACM SIPLAN '84 Symposium on Compiler Construction, published as ACM SIPLAN Notices, Vol. 19, No. 6

Stichwortverzeichnis

Abkürzungsverzeichnis

In diesem Verzeichnis werden die in den Modulgraphbeispielen verwendeten Abkürzungen für Knoten- und Kantenmarkierungen aufgeführt.

Knotenmarkierungen:

AssStat	:	assignment_statement
CallByRef	:	call_by_reference_parameter
ConstDecl	:	constant_declaration
ConstDeclList	:	constant_declaration_list
DeclList	:	declaration_list
Expr	:	expression
EndIfStat	:	end_if_statement
EndWhileStat	:	end_while_statement
FieldCompList	:	field_component_list
FormParPart	:	formal_parameter_part
FormParList	:	formal_parameter_list
GreExpr	:	'>'_expression
Ident	:	identifier
IdentList	:	identifier_list
IfStat	:	if_statement
ImplMod	:	implementation_module
MinSimpExpr	:	'-'_simple_expression
OptImpList	:	opt_import_list
OptStatList	:	opt_statement_list
OptStatPart	:	opt_statement_part
PlusSimpExpr	:	'+'_simple_expression
ProcCall	:	procedure_call
ProcDecl	:	procedure_declaration
ProgMod	:	program_module
RepeatStat	:	repeat_statement
RecordComp	:	record_component
SimpleExpr	:	simple_expression
StatList	:	statement_list
SubrType	:	subrange_type
TypeDecl	:	type_declaration
TypeDeclList	:	type_declaration_list
TypeDef	:	type_definition
Var	:	variable
VarDecl	:	variable_declaration
VarDeclList	:	variable_declaration_list
WhileStat	:	while_statement

Kantenmarkierungen:

EConstExpr	:	edge_constant_expression
EEndStat	:	edge_end_statement
EExpr	:	edge_expression
EFieldCompList	:	edge_field_component_list
EFirstConstExpr	:	edge_first_constant_expression
EFirstSimpExpr	:	edge_first_simple_expression
EFormParList	:	edge_formal_parameter_list
EFormParPart	:	edge_formal_parameter_part
EFormParType	:	edge_formal_parameter_type
EIdent	:	edge_identifier
EIdentList	:	edge_identifier_list
ESecConstExpr	:	edge_second_constant_expression
ESecSimpExpr	:	edge_second_simple_expression
EStatList	:	edge_statement_list
ETypeDef	:	edge_type_definition
EUnsiSimpExpr	:	edge_unsigned_simple_expression
EUnsiSimpConstExpr	:	edge_unsigned_simple_constant_expression
EVar	:	edge_variable

Anhang A

Normierte EBNF

Bei der im folgenden angegebenen EBNF, die die für das Programmieren-im-Kleinen benötigte Teilmenge von Modula-2 beschreibt, ist die Aufteilung der nichtterminalen Symbole in drei disjunkte Teilmengen gegeben (vgl. Paragraph 3.1).

Die Produktionen zu optionalen Nonterminals werden nicht aufgeführt, da sie stets dieselbe, im folgenden exemplarisch angegebene Gestalt haben:

```
<opt_dummy>  ::=  ε | <dummy>
```

Alle erklärten nichtterminalen Symbole sind somit stets Alternativen-, Struktur- oder Listen-Nonterminals.

```
<module>          ::= <program_module> | <implementation_module>

<program_module>  ::= MODULE <ident> ;
                        <opt_import_list>
                        <opt_declaration_list>
                        <opt_statement_part>
                      END <ident> .

<implementation_module> ::= IMPLEMENTATION MODULE <ident> ;
                              <opt_import_list>
                              <opt_declaration_list>
                              <opt_statement_part>
                            END <ident> .

<import_list>  ::=  <import> <import_list> | <import>
<import>  ::=  <opt_import_module> IMPORT <ident_list> ;
<import_module>  ::=  FROM <ident>

<declaration_list> ::= <declaration> <declaration_list> | <declaration>
<declaration>      ::= <constant_declaration_part> |
                       <type_declaration_part> |
                       <var_declaration_part> |
                       <procedure_declaration>

<constant_declaration_part> ::= CONST <opt_constant_declaration_list>
<constant_declaration_list> ::= <constant_declaration>
                                <constant_declaration_list> |
                                <constant_declaration>
<constant_declaration>      ::= <ident> = <constant_expression> ;

<type_declaration_part> ::= TYPE <opt_type_declaration_list>
<type_declaration_list> ::= <type_declaration> <type_declaration_list> |
                            <type_declaration>
<type_declaration>      ::= <ident> = <type_definition> ;

<var_declaration_part>  ::= VAR <opt_var_declaration_list>
<var_declaration_list>  ::= <var_declaration> <var_declaration_list>  |
```

```
                              <var_declaration>
<var_declaration>          ::= <ident_list> : <type_definition> ;

<procedure_declaration> ::= PROCEDURE <ident> <opt_formal_parameter_part> ;
                               <opt_declaration_list>
                               <opt_statement_part>
                            END <ident> ;

<formal_parameter_part> ::= ( <opt_formal_parameter_list> ) <opt_result_type>
<formal_parameter_list> ::= <formal_parameter> ; <formal_paramter_list>
                            <formal_parameter>
<formal_parameter>         ::= <call_by_reference_parameter> |
                            <call_by_value_parameter>
<call_by_reference_parameter> ::= VAR <ident_list> : <formal_parameter_type>
<call_by_value_parameter> ::= <ident_list> : <formal_parameter_type>
<formal_parameter_type>    ::= <ident> | <open_array_type>
<open_array_type>          ::= ARRAY OF <ident>


<type_definition>    ::= <ident>          | <enumeration_type> |
                         <subrange_type> | <array_type>      |
                         <record_type>   | <set_type>        |
                         <pointer_type>  | <procedure_type>

<enumeration_type>   ::= ( <ident_list> )

<subrange_type>      ::= [ <constant_expression> .. <constant_expression>]

<array_type>         ::= ARRAY <simple_type_list> OF <type_definition>

<simple_type_list>   ::= <simple_type> , <simple_type_list> |
                         <simple_type>
<simple_type>        ::= <qualident>   | <subrange_type> |
                         <enumeration_type>

<record_type>        ::= RECORD
                            <opt_field_component_list>
                         END

<field_component_list> ::= <field_component> ; <field_component_list> |
                           <field_component>
<field_component>    ::= <record_component> | <variant_component>

<record_component>   ::= <ident_list> : <type_definition>

<variant_component> ::= CASE <opt_tag_field> <qualident> OF
                          <case_component_list>
                          <opt_else_component>
                        END
<tag_field>          ::= <ident> :

<case_component_list> ::= <case_component> "|" <case_component_list> |
                          <case_component>

<case_component> ::= <case_label_list> : <field_component_list>

<case_label_list> ::= <case_label> , <case_label_list>  | <case_label>
<case_label>      ::= <constant_expression> | <constant_expression_range>
```



```
<constant_expression_range> ::= <constant_expression> .. <constant_expression>

<else_component> ::= ELSE <opt_field_component_list>


<set_type>          ::= SET OF <simple_type>
<pointer_type>      ::= POINTER TO <type_definition>
<procedure_type>    ::= PROCEDURE <opt_formal_type_part>

<formal_type_part> ::= ( <opt_formal_type_list> ) <opt_result_type>
<formal_type_list> ::= <formal_type> , <formal_type_list> |  <formal_type>
<formal_type>      ::= <call_by_reference_formal_parameter_type> |
                       <call_by_value_formal_parameter_type>

<call_by_reference_formal_parameter_type> ::= VAR <formal_parameter_type>
<call_by_value_formal_parameter_type>     ::= <formal_parameter_type>

<result_type>      ::= : <ident>


<statement_part>       ::= BEGIN <opt_statement_list>
<statement_list>       ::= <statement> ; <statement_list> | <statement>

<statement>            ::= <assignment_statement> | <procedure_call>  |
                           <if_statement>         | <case_statement>  |
                           <while_statement>      | <repeat_statement>|
                           <loop_statement>       | <for_statement>   |
                           <with_statement>       | <exit_statement>  |
                           <return_statement>


<assignment_statement> ::= <variable> := <expression>

<variable>             ::= <ident> <opt_selector_list>
<selector_list>        ::= <selector> <selector_list> | <selector>
<selector>             ::= <index> | <field> | <reference>

<index>                ::= [ <expression_list> ]
<field>                ::= . <ident>
<reference>            ::= ^

<procedure_call>   ::= <variable> <opt_parameter_part>
<parameter_part>   ::= ( <opt_expression_list> )

<if_statement>     ::= IF <expression> THEN <opt_statement_list>
                          <opt_elsif_part_list>
                          <opt_else_part>
                       END

<elsif_part_list>  ::= <elsif_part> <elsif_part_list> |
                          <elsif_part>
<elsif_part>       ::= ELSIF <expression> THEN <opt_statement_list>
<else_part>        ::= ELSE  <opt_statement_list>

<case_statement>   ::= CASE <expression> OF
                          <case_alternative_list>
                          <opt_else_part>
                       END
```

```
<case_alternative_list> ::= <case_alternative> "|" <case_alternative_list> |
                            <case_alternative>
<case_alternative> ::= <case_label_list> : <opt_statement_list>

<while_statement>  ::= WHILE <expression> DO
                           <opt_statement_list>
                       END

<repeat_statement> ::= REPEAT
                           <opt_statement_list>
                       UNTIL <expression>

<for_statement>    ::= FOR <ident> := <expression> TO <expression>
                          <opt_inc_expression> DO
                          <opt_statement_list>
                       END
<inc_expression>   ::= BY <constant_expression>

<loop_statement> ::= LOOP
                         <opt_statement_list>
                     END

<with_statement> ::= WITH <variable> DO
                         <opt_statement_list>
                     END

<exit_statement> ::= EXIT

<return_statement> ::= RETURN <opt_expression>


<constant_expression>   ::= <simple_const_expression> | <'='_const_expression> |
                            <'<>'_const_expression>   | <'<'_const_expression> |
                            <'<='_const_expression>   | <'>'_const_expression> |
                            <'>='_const_expression>   | <'in'_const_expression>


<'='_const_expression>  ::= <simple_const_exp>  =   <simple_const_exp>
<'<>'_const_expression> ::= <simple_const_exp>  <>  <simple_const_exp>
<'<'_const_expression>  ::= <simple_const_exp>  <   <simple_const_exp>
<'<='_const_expression> ::= <simple_const_exp>  <=  <simple_const_exp>
<'>='_const_expression> ::= <simple_const_exp>  >=  <simple_const_exp>
<'>'_const_expression>  ::= <simple_const_exp>  >   <simple_const_exp>
<'in'_const_expression> ::= <simple_const_exp>  IN  <simple_const_exp>

<simple_const_exp>      ::= <opt_sign> <unsigned_simple_const_exp>

<sign>                  ::= + | -

<unsigned_simple_const_exp>  ::= <const_term>          |
                                 <'+'_simple_const_exp> |
                                 <'-'_simple_const_exp> |
                                 <'or'_simple_const_exp>


<'+'_simple_const_exp>  ::= <const_term>  +   <unsigned_simple_const_exp>
<'-'_simple_const_exp>  ::= <const_term>  -   <unsigned_simple_const_exp>
<'or'_simple_const_exp> ::= <const_term>  OR  <unsigned_simple_const_exp>
```



```
<const_term>   ::= <const_factor>      | <'*'_const_term>   |
                   <'/'_const_term>    | <'div'_const_term> |
                   <'mod'_const_term>  | <'and'_const_term>

<const_factor>          ::= <ident>    | <number>  |
                            <string>   | <bracketed_const_exp> |
                            <set>      | <not_const_factor>


<bracketed_const_exp>   ::= ( <constant_expression> )

<set>                   ::= <opt_ident> { <element_list> }

<element_list>  ::= <element> , <element_list>  | <element>
<element>       ::= <constant_expression>  | <constant_expression_range>

<not_const_factor>      ::= NOT <const_factor>


<'*'_const_term>        ::= <const_factor>  *    <const_term>
<'/'_const_term>        ::= <const_factor>  /    <const_term>
<'div'_const_term>      ::= <const_factor>  DIV  <const_term>
<'mod'_const_term>      ::= <const_factor>  MOD  <const_term>
<'and'_const_term>      ::= <const_factor>  AND  <const_term>


<expression_list> ::= <expression> , <expression_list>  | <expression>

<expression> ::=  <simple_expression>  | <'='_expression>  |
                  <'<>'_expression>    | <'<'_expression>  |
                  <'<='_expression>    | <'>='_expression> |
                  <'>'_expression>     | <'in'_expression>

<'='_expression>  ::= <simple_expression>  =   <simple_expression>
<'<>'_expression> ::= <simple_expression>  <>  <simple_expression>
<'<'_expression>  ::= <simple_expression>  <   <simple_expression>
<'<='_expression> ::= <simple_expression>  <=  <simple_expression>
<'>='_expression> ::= <simple_expression>  >=  <simple_expression>
<'>'_expression>  ::= <simple_expression>  >   <simple_expression>
<'in'_expression> ::= <simple_expression>  IN  <simple_expression>

<simple_expression> ::= <opt_sign> <unsigned_simple_expression>

<unsigned_simple_expression> ::=  <term>  |
                                  <'+'_simple_expression>  |
                                  <'-'_simple_expression>  |
                                  <'or'_simple_expression>


<'+'_simple_expression>  ::= <term>  +   <unsigned_simple_expression>
<'-'_simple_expression>  ::= <term>  -   <unsigned_simple_expression>
<'or'_simple_expression> ::= <term>  OR  <unsigned_simple_expression>


<term>      ::=  <factor>      |  <'*'_term>  |
                 <'/'_term>    |  <'div'_term> |
                 <'mod'_term>  |  <'and'_term>
```

```
<factor>     ::=  <variable>     |  <number>  |
                  <string>       |  <set>     |
                  <not_factor>   |  <function_designator>  |
                  <bracketed_expression>

<not_factor>              ::= NOT <factor>

<function_designator>     ::= <variable> <opt_parameter_part>

<bracketed_expression>    ::= ( <expression> )


<'*'_term>                ::= <factor>  *   <term>
<'/'_term>                ::= <factor>  /   <term>
<'div'_term>              ::= <factor>  DIV <term>
<'mod'_term>              ::= <factor>  MOD <term>
<'and'_term>              ::= <factor>  AND <term>

<ident_list>              ::= <ident> , <ident_list>  | <ident>
<ident>                   ::= <letter> <opt_letter_or_digit_list>

<letter_or_digit_list>    ::= <letter_or_digit> <letter_or_digit_list>|
                              <letter_or_digit>
<letter_or_digit>         ::= <letter>  |  <digit>
<letter>                  ::= A | B | C | ... | Z

<digit_list>              ::= <digit> <digit_list>  | <digit>
<digit>                   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<number>                  ::= <decimal_natural>     |  <octal_natural>  |
                              <character_natural>   |  <hexadecimal_natural>|
                              <rational>

<decimal_natural>         ::= <digit_list>
<octal_natural>           ::= <oct_digit_list> B
<character_natural>       ::= <oct_digit_list> C

<oct_digit_list>          ::= <oct_digit> <oct_digit_list>   | <oct_digit>
<oct_digit>               ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hexadecimal_natural>     ::= <digit> <opt_hex_digit_list> H

<hex_digit_list>          ::= <hex_digit> <hex_digit_list>  | <hex_digit>
<hex_digit>               ::= 0 | 1 | ... | 9 | A | B | C | D | E | F

<rational>                ::= <digit> <opt_digit_list> .
                                    <opt_digit_list> <opt_scale_factor>
<scale_factor>            ::= E <opt_sign> <digit> <opt_digit-list>

<string>                  ::= <string_in_quote_marks>  |  <string_in_apostrophes>
<string_in_quote_marks>   ::= " <opt_character_list> "
<string_in_apostrophes>   ::= ' <opt_character_list> '
```

Lebenslauf

10.04.1955	geboren in Gelsenkirchen
1961 - 1965	Marienschule Gelsenkirchen
1965 - 1970	Schalker Gymnasium Gelsenkirchen
1970 - 1973	Gesamtschule Gelsenkirchen
22.05.1973	Abitur
1973 - 1974	Grundwehrdienst
1974 - 1980	Studium an der Universität Dortmund
04.12.1980	Diplom in Informatik mit Nebenfach Mathematik
1980 - 1981	wissenschaftliche Hilfskraft an der Abteilung Informatik der Universität Dortmund
1981 - 1986	wissenschaftlicher Mitarbeiter am Fachbereich Mathematik/Informatik der Universität Osnabrück