Diplomarbeit

Generierung eines syntaxgesteuerten Editors für ProSet mit dem Synthesizer Generator

Andreas Bubolz

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

27. Juni 1996

Gutachter:

Prof. Dr. E.-E. Doberkat
Dr. S. Dißmann

Teil I
Schriftliche Ausarbeitung

Zusammenfassung

Die Aufgabe dieser Diplomarbeit ist die Entwicklung eines syntaxgesteuerten Editors für die Prototyping-Sprache ProSet. Dieser Editor wird zentrales Element einer integrierten, inkrementorientierten Programmentwicklungsumgebung (PEU) für ProSet sein, indem er weiteren Werkzeugen einen Zugriff auf die interne Darstellung von Programmen ermöglicht.

Da sich die Entwicklung syntaxgesteuerter Editoren auf der Abstraktionsebene universeller Programmiersprachen einerseits als äußert komplex und andererseits als recht unflexibel hinsichtlich einer möglichen Weiterentwicklung der zu unterstützenden Programmiersprache erweist, wird in dieser Arbeit der **Synthesizer Generator**, ein Werkzeug, das speziell der Generierung syntaxgesteuerter Editoren dient, eingesetzt.

Stichwörter

Programmentwicklungsumgebung, Editor, syntaxgesteuerter Editor, Synthesizer Generator, ProSet

Abstract

The object of this diploma thesis is the development of a syntax-directed editor for the prototyping language ProSet. As the editor will allow further tools access to the internal representation of programs it will be the central element of an integrated, increment-oriented program development environment for ProSet.

The development of syntax-directed editors on the abstraction level of universal programming languages is very komplex and inflexible in respect of a further development of the supported programming language. This thesis uses the **Synthesizer Generator**, a special tool for developing syntax-directed editors.

Keywords

program development environment, editor, syntax-directed editor, Synthesizer Generator, ProSet

Inhaltsverzeichnis

Kapitel 1

Einleitung

Nachdem kurz der Kontext, in den diese Arbeit einzuordnen ist, umrissen und die Arbeit selbst motiviert wurde, wird zunächst die zugrundeliegende Aufgabenstellung konkretisiert werden. Anschließend wird ein Überblick über diese Arbeit gegeben und verschiedene Konventionen vereinbart.

1.1 Kontext und Motivation

ProSet ist eine mengenorientierte Programmiersprache, welche die schnelle Konstruktion und Modifikation ausführbarer Prototypen unterstützt. Von grundlegender Bedeutung für den erfolgreichen Einsatz von Prototyping ist die Bereitstellung einer geeigneten Entwicklungsumgebung. Aus diesem Grund ist zunächst die Entwicklung einer integrierten, inkrementorientierten Programmentwicklungsumgebung (PEU) für ProSet geplant. Naheliegenderweise sollte die Entwicklung der PEU mit der Konstruktion eines syntaxgesteuerten Editors zum Erstellen und Verändern von Programmtexten beginnen.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit ist die Generierung eines derartigen Editors für ProSet mit dem **Synthesizer Generator**. Dieser erzeugt einen sprachspezifischen Editor aus einer Spezifikation, welche die abstrakte Syntax, kontextsensitive Beziehungen, Ausgabeformate, die konkrete Syntax sowie Regeln zur Restrukturierung von Programmen umfaßt. Die Spezifikationssprache basiert auf dem Konzept der attributierten Grammatiken. Zu den besonderen Merkmalen des **Synthesizer Generator** zählt die Erzeugung inkrementeller Werkzeuge.

Der syntaxgesteuerte Editor soll die komfortable und effiziente Erstellung und Manipulation von ProSet-Programmen gestatten und den vollständigen Sprachumfang unterstützen. Die Arbeit soll zudem verschiedene Benutzergruppen unterstützen und ein kontextsensitives Hilfesystem bereitstellen. Des weiteren soll die Benutzung verschiedener Sichten auf die Programmrepräsentation unterstützt werden. Beispielsweise könnte eine Sicht lediglich Fehlermeldungen anzeigen, eine weitere bestimmte Konstrukte ausblenden. Weitere Anforderungen sind durch die Analyse des Verhaltens und der Wünsche der verschiedenen Benutzergruppen zu ermitteln.

Um die Nachteile von syntaxgesteuerten Editoren zu vermeiden, soll auch die freie Eingabe geeigneter Konstrukte ermöglicht werden. Aus diesem Grund sollen zudem zeitweilige Inkonsistenzen in Bezug auf die kontextfreie Syntax und der statischen Semantik zugelassen werden. Neben der Überprüfung von kontextfreien und kontextsensitiven Bedingungen ist es wünschenswert, ggf. auf expliziten Wunsch des Benutzers zusätzliche statische Analysen anzubieten. Beispiele hierfür sind das Auffinden einer Deklaration zu einem angewendeten Auftreten einer Variablen, das Auffinden aller zu einer Prozedur nichtlokalen Variablen und das Verbot impliziter Deklarationen.

ProSet ist eine Breitbandsprache, d.h. Programme können auf unterschiedlichem Niveau formuliert werden. Des weiteren können Konstrukte, die auf einem hohen Niveau spezifiziert sind, zu äquivalenten Konstrukten auf einem niedrigeren Niveau transformiert werden. Der Editor soll derartige Transformationen und soweit möglich deren Invertierung unterstützen.

Die Konstruktion einer integrierten PEU erfordert die Berücksichtigung verschiedener Integrationsdimensionen, insbesondere der Präsentations-, Daten- und Kontrollintegration. Dies beinhaltet eine geeignete Spezifikation des attributierten abstrakten Syntaxbaums als zentrale Programmrepräsentation und somit als Schnittstelle für weitere Werkzeuge der PEU.

1.3 Überblick

Nach dieser Einleitung werden zunächst im Kapitel 2 die theoretischen Grundlagen von Editoren im allgemeinen sowie des **Synthesizer Generator** und der durch ihn erzeugten Editoren im speziellen erläutert werden. Außerdem wird ein Überblick über die Programmiersprache ProSet gewährt, wobei die für diese Arbeit wesentlichen Aspekte dieser Sprache hervorgehoben werden. Nachdem in Kapitel 3 verschiedene Anforderungen an den ProSet-Editor herausgearbeitet wurden, werden in den beiden folgenden Kapiteln 4 und 5 zunächst grundlegende Entwurfsentscheidungen getroffen und anschließend die Implementation des ProSet-Editors betrachtet. In Kapitel 6 werden solche Eigenschaften des ProSet-Editors erläutert, die sich nicht zwingend aus der Benutzung des **Synthesizer Generator** als Entwicklungswerkzeug ergeben und die daher nicht im Referenzhandbuch des **Synthesizer Generator** [RT89b] beschrieben sind. Kapitel 7 wird zunächst kurz die erweiterten Möglichkeiten der Version 5.0 des **Synthesizer Generator** darstellen. Diese Version stand leider erst kurz vor Beendigung dieser Arbeit zur Verfügung und konnte daher nur nachrangige Betrachtung finden. Desweiteren enthält dieses Kapitel einen Ausblick auf die weitere Entwicklung des Editors hinsichtlich seiner Integration mit anderen Werkzeugen zur Programmentwicklung sowie ein Resümee über die zurückliegende Arbeit. Die Spezifikation des abstrakten Syntaxbaums ist im Anhang A und die Spezifikation der Attributierung im Anhang B zu finden.

1.4 Konventionen

Innerhalb dieser Arbeit werden, um Lesbarkeit und Verständlichkeit zu fördern, verschiedene, den Textsatz betreffende Konventionen befolgt werden: Syntaktische Regeln werden vom restlichen Text abgesetzt dargestellt werden. Innerhalb dieser Regeln werden Bezeichner *kursiv* dargestellt, während feststehende Teile einer Regel, wie im restlichen Text auch in `Schreibmaschinenschrift` gesetzt werden. So stellt in folgender Regel

`left` *Phylum* ;

`left` ein feststehendes Schlüsselwort dar, während *Phylum* ein Nichtterminalsymbol bezeichnet. Beispielprogrammteile werden ebenfalls in `Schreibmaschinenschrift` geschrieben werden.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit erläutert. Zunächst werden einige grundlegende Begriffe eingeführt. Anschließend werden Editoren sowie deren konzeptionelle Unterschiede beschrieben. Darauf folgend wird der **Synthesizer Generator** vorgestellt. Eine kurze Beschreibung der Programmiersprache ProSet wird den Abschluß dieses Kapitels darstellen.

2.1 Grundlegende Begriffe

Innerhalb dieser Arbeit werden häufig die Begriffe *Syntax* und *Semantik* sowie von diesen abgeleitete Begriffe verwendet. Sie werden hier gemäß [Kas90, Kapitel 1.2] definiert:

Die *Syntax* einer Sprache wird durch eine kontextfreie Grammatik definiert. Die Regeln der Grammatik werden auch *Produktionen* genannt. Die Untersuchung der Korrektheit eines Programms hinsichtlich der Syntax der Programmiersprache heißt *syntaktische Analyse*. Die kontextabhängigen Eigenschaften einer Sprache werden *statische Semantik* genannt. Diese kann bei den meisten höheren Programmiersprachen nicht durch eine kontextfreie Grammatik beschrieben werden. In dieser Arbeit wird der Formalismus der *attributierten Grammatik* verwendet, um die statische Semantik zu spezifizieren. Eine attributierte Grammatik ist die Erweiterung der kontextfreien Grammatik um Attribute sowie um Attributregeln. Ein *Attribut* sei hier eine Information beliebiger Art, die an ein Nichtterminalsymbol oder an eine Produktion der Syntax gebunden ist. Die Menge aller Attribute sowie die Regeln zu deren Berechnung werden *Attributierung* genannt. Die statische Semantik umfaßt üblicherweise Typregeln, Regeln zur Gültigkeit von Definitionen sowie diverse lokale Kontextabhängigkeiten. Die Schritte, die ein Programm ausführt, sowie die Wirkung, die dabei jeder Schritt erzielt, werden durch die *dynamische Semantik* der Programmiersprache definiert.

Die Syntax der Programmiersprache ProSet wird durch eine *erweiterte Backus-Naur-Form* (EBNF) in [DFH+92, Anhang C] definiert. Die Semantik von ProSet wird in der Sprachbeschreibung verbal mit Bezug auf die Syntax beschrieben. Des weiteren exisitiert eine Spezifikation der Semantik in Form einer attributierten Grammatik für das Übersetzerwerkzeug *ELI* [WHK88]. Ein ProSet-Programm wird als syntaktisch und semantisch korrekt bezeichnet, wenn es diesen Regeln entspricht.

2.2 Editoren

Ein Editor ist ein Werkzeug, das der Eingabe, der Manipulation und der Darstellung verschiedenartigster Texte oder auch Grafiken dient. Hier sollen jedoch generell nur solche Editoren betrachtet werden, die die Bearbeitung von Texten ermöglichen. Im allgemeinen kann es sich bei diesen Texten um beliebige Zeichenfolgen handeln. Mit einem Editor lassen sich also z.B. Briefe ebenso erstellen wie Programme. In dem Maße, in dem sich die Erstellung eines Programms vom Schreiben eines Briefes unterscheidet, unterscheiden sich auch die Konzepte und Arbeitsweisen, auf denen verschiedene Editoren beruhen können.

2.2.1 Konventionelle Editoren

Bei konventionellen Editoren handelt es sich um Werkzeuge, für die die Struktur des zu erstellenden Textes unerheblich ist. Ein Text ist in diesem Falle nur eine Folge von Zeichen. Nur einzelne Zeichen wie der Zeilenvorschub und andere Trennzeichen erhalten hier eine besondere Bedeutung, die aber auch nur die Darstellung auf dem Bildschirm oder die Interaktion mit dem Benutzer beeinflussen.

Bei der Arbeit mit dem Editor wird dem Benutzer durch den *Cursor* angezeigt, an welcher Stelle des Textes die nächste Änderung erfolgen wird. Dieser Cursor zeigt demnach auf ein einzelnes Zeichen oder zwischen zwei Zeichen des Textes. Er kann normalerweise horizontal wie vertikal zeichenweise, wortweise bzw. zeilen- oder seitenweise bewegt werden.

Text wird einfach durch Betätigung der entsprechenden einzelnen Tasten eingegeben. Gelöscht werden kann der Text wiederum zeichen- oder wortweise. Komplexere Befehle zur Bearbeitung eines Textes stellen hier schon das Suchen und Ersetzen von Textmustern und das Markieren, Verschieben, Duplizieren oder Löschen von Textblöcken dar.

Nach Beendigung der Arbeit an einem Text, wird dieser meist direkt als Textdatei abgespeichert. Das heißt, es werden keine zusätzlichen Informationen wie Steuerzeichen, Position des Cursors o.ä. eingefügt. Mit einem konventionellen Editor erstellte Texte können so direkt von anderen Programmen wie einem Übersetzer weiterverarbeitet werden.

2.2.2 Syntaxgesteuerte Editoren

Um die korrekte Erstellung von Texten, die gemäß der formalen Syntax einer bestimmten Sprache aufgebaut sein müssen, zu unterstützen, sind Editoren entwickelt worden, die speziell auf diesen Aufgabenbereich zugeschnitten sind. So werden Texte hier nicht einfach als Folgen von Zeichen betrachtet sondern hinsichtlich der syntaktischen Struktur der Sprache. Diese Editoren werden als syntaxgesteuert bezeichnet. Obwohl Programmtexte keineswegs die einzigen Texte sind, die gemäß einer formalen Syntax aufgebaut sind, sollen sie hier stellvertretend für diese betrachtet werden.

Da verschiedene Programmiersprachen sich auch in ihrem syntaktischen Aufbau unterscheiden, eignet sich ein syntaxgesteuerter Editor meistens nur zur Erstellung von Programmen einer bestimmten Programmiersprache.

Ein Programmtext wird bei einem solchen Editor intern üblicherweise als abstrakter Syntaxbaum (AST) repräsentiert, der gemäß der abstrakten Syntax der unterstützten Programmiersprache erstellt und bearbeitet wird. Dieser AST wird durch das sogenannte *Unparsing* in eine textuelle Repräsentation umgewandelt, die dem Benutzer zur Anzeige gebracht wird. Der Unparsing-Vorgang soll eine übersichtliche und konsequente Darstellung des Programmtextes als Ergebnis haben (*Pretty Printing*). Zur Modifikation des zu erstellenden Programms gibt der Benutzer eines syntaxgesteuerten Editors nicht zeichenweise Text ein, sondern er manipuliert direkt den AST. Dabei stellen die meisten solcher Manipulationen eine Ersetzung eines Teilbaums des AST durch einen neuen dar. Ein so durch einen anderen ersetzbaren Teilbaum wird als *Inkrement* bezeichnet. Die textuelle Repräsentation des aktuell ausgewählten Inkrements wird dem Benutzer hervorgehoben dargestellt und wird *strukturelle Selektion* genannt. Diese besteht immer aus einer lückenlosen Einheit des Programmtextes, da sie einen Teilbaum des AST repräsentiert. Die visuelle Hervorhebung der strukturellen Selektion kann z.B. durch Farbgebung oder durch Unterstreichung gegeben sein. Da sie im Gegensatz zu einem normalen Cursor eventuell mehrere Zeichen oder auch mehrere Zeilen umfaßt, wird diese Hervorhebung auch als *Flächen-Cursor* bezeichnet.

Während ein Programm bei einem konventionellen Editor in beliebiger Reihenfolge eingegeben werden kann, geschieht dies bei der Benutzung eines syntaxgesteuerten Editors normalerweise Top-Down von der Wurzel des AST aus. Die Bewegung der strukturellen Selektion innerhalb des Programmtextes korrespondiert immer mit einer Selektion eines Knotens des zugrunde liegenden AST. Ein syntaxgesteuerter Editor muß also über Kommandos verfügen, die eine Navigation innerhalb des AST gemäß verschiedener Strategien wie Tiefendurchlauf oder Breitendurchlauf ermöglichen.

Komplexe Manipulationsmöglichkeiten bieten die sogenannten Transformationen: Mit ihrer Hilfe ist es möglich, Teilbäume des AST abzutrennen, diese zu modifizieren und anschließend wieder in den AST einzufügen. Ein syntaxgesteuerter Editor ist damit also in der Lage, aufwendige Umstrukturierungen an einem Programm vorzunehmen.

Da ein Progamm immer durch einen AST repräsentiert wird, der gemäß der abstrakten Syntax der zu unterstützenden Sprache aufgebaut wird, ist eine syntaktische Korrektheit eingegebener Programmeinheiten zu jeder Zeit gegeben.

Wird der AST mittels Attributierung um Informationen erweitert, die die semantischen Aspekte eines Programms betreffen, kann so auch eine Korrektheit entsprechend der statischen Semantik zu jeder Zeit sichergestellt werden. Diese kann von einem syntaxgesteuerten Editor zu jeder Zeit erzwungen werden, indem die Bewegung der strukturellen Selektion verhindert wird, solange diese Selektion auf ein fehlerhaftes Inkrement weist. Ein solcher Zwang kann sich aber besonders bei der Erstellung von Prototypen als äußerst hinderlich erweisen. Daher wird bei syntaxgesteuerten Editoren oft die Ausgabe von Fehlermeldungen oder Warnungen bei semantischen Fehlern vorgezogen.

Da eine Berechnung aller Attribute eines AST oft lange dauern wird, sollte ein syntaxgesteuerter Editor Berechnung nur auf Anweisung durch den Benutzer oder aber inkrementell nach jeder Modifikation des AST ausführen.

2.2.3 Hybrid-Editoren

Da einerseits konventionelle Editoren kaum programmierspezifische Unterstützung anbieten und andererseits rein syntaxgesteuerte Editoren keine freie Texteingabe und dadurch oft nur eine umständliche Eingabe verschiedener Programmeinheiten ermöglichen, sind Hybrid-Editoren entwickelt worden. Diese bieten sowohl die Vorteile des einen als auch die des anderen Verfahrens. Hybrid-Editoren stellen eine Erweiterung der rein syntaxgesteuerten Editoren um die Möglichkeit der textuellen Eingabe zumindest an ausgewählten Inkrementen dar. Um diese textuelle Eingabe zu ermöglichen, schaltet der Hybrid-Editor entweder auf Anweisung durch den Benutzer oder aber automatisch bei Betätigung einer alphanumerischen Taste in den Texteingabemodus um. In diesem Modus kann der Benutzer, falls der Editor unter einer graphischen Benutzeroberfläche arbeitend ein neues Eingabefenster öffnet, wie mit einem konventionellen Editor arbeiten. Öffnet der Editor kein spezielles Texteingabefenster, so werden die eingegeben Zeichen an der Stelle des Programmtextes dargestellt, die den selektierten Knoten des AST repräsentiert. Wird dieser Eingabemodus verlassen, muß die neue bzw. modifizierte Programmeinheit geparst und an geeigneter Stelle in den AST eingefügt werden. Damit nach einer textuellen Eingabe nicht jedesmal der gesamte eingegebene Text geparst werden muß, sollte ein syntaxgesteuerter Editor also möglichst einen Parser verwenden, der für jeden Knoten des AST, der textuell eingegeben werden kann, einen Eintrittspunkt bietet.

2.2.4 Pro und Contra

Konventionelle Editoren sind, da sie nicht an einer speziellen Syntax orientiert arbeiten, offensichtlich die wesentlich flexibleren Werkzeuge: Sie lassen sich zur Eingabe aller denkbaren Texte oder Zeichenfolgen einsetzen. Darüber hinaus sind sie (meistens) einfach zu bedienen und geben dem Benutzer alle Freiheiten bei der Bearbeitung eines Textes.

Syntaxgesteuerte Editoren sind dagegen „nur“ zur Erstellung von Programmen in einer bestimmten Programmiersprache zu gebrauchen. Die Arbeit mit ihnen stellt wegen ihren (noch) geringen Verfügbarkeit für die meisten Programmierer eine große Umgewöhnung dar. Des weiteren sind sie wesentlich aufwendiger zu entwickeln und benötigen auch mehr Systemressourcen.

Demgegenüber wird dem Benutzer eines syntaxgesteuerten Editors viel weniger Wissen bezüglich der Syntax der verwendeten Sprache abverlangt. Ist der Benutzer erst mit der Bedienung eines solchen Editors vertraut, so kann er sich auf die Formulierung eines Algorithmus konzentrieren, ohne auf syntaktische Eigenschaften der benutzten Programmiersprache und ohne z.B. auf eine ästhetische, übersichtliche und evtl. vorgegebenen Programmierrichtlinien entsprechende Form des Textes achten zu müssen. Zusätzlich kann er z.B. durch verschiedene Darstellungen seines Programmtextes hinsichtlich unterschiedlicher Kriterien in seiner Arbeit unterstützt werden. Außerdem werden viele Programmierfehler bei der Eingabe von Programmen durch die Verwendung syntaxgesteuerter Editoren vermieden oder frühzeitig erkannt werden.

Auf die interne strukturelle Darstellung eines Programmtextes können auch andere Werkzeuge zugreifen. So können z.B. inkrementelle Übersetzer verwendet werden, um die Turn-Around-Zeiten zu verkürzen. Ebenso sind symbolische Debugger oder verschiedenste CASE-Werkzeuge denkbar, die auf denselben Daten wie der Editor arbeiten, um so mit ihm zu einer kompletten Softwareentwicklungsumgebung zu verschmelzen.

Die Nachteile der starken Inflexibiltät bei der Programmentwicklung, die rein syntaxorientierten Editoren vorgeworfen werden können, sind bei der Verwendung von Hybrid-Editoren aufgehoben oder zumindest stark vermindert.

2.3 Der Synthesizer Generator

Die Entwicklung des **Synthesizer Generator** beruht auf dem seit 1978 an der Cornell University, Ithaca, USA erstellten *Cornell Program Synthesizer*, einer Programmentwicklungsumgebung für eine kleine Untermenge der Programmiersprache PL/I. Diese Entwicklungsumgebung enthielt neben einem syntaxgesteuerten Editor einen inkrementellen Übersetzer sowie einen Debugger. 1984 wurde aus dem Cornell Program Synthesizer der **Synthesizer Generator** entwickelt. Dieser ist nunmehr nicht auf eine bestimmte Sprache zugeschnitten, sondern stellt vielmehr ein generisches System dar, das mittels verschiedener formaler Spezifikationen zur Generierung von Programmentwicklungsumgebungen für verschiedenste Programmiersprachen benutzt werden kann. Zentrales Element dieser Programmierentwicklungen stellt der syntaxgesteuerte Editor dar. Zu Beginn dieser Arbeit stand die Version 4.2 des **Synthesizer Generator** zur Verfügung. Die wesentlich erweiterte Version 5.0 konnte erst kurz vor Abschluß dieser Arbeit verwendet werden. Daher wird hier, soweit nicht explizit anders ausgewiesen, immer auf die Version 4.2 Bezug genommen. Die Neuheiten der Version 5.0 werden im Kapitel 7 vorgestellt.

Ein mittels des **Synthesizer Generator** entwickelter Editor weist folgende äußerliche Merkmale auf:

Multiple Puffer: Zur selben Zeit können mit dem Editor verschiedene Programme bearbeitet werden.

Multiple Fenster: Während einer Editorsitzung können beliebig viele Fenster für eventuell verschiedene Programme oder verschiedene Sichten dargestellt werden.

Multiple Sichten: Ein und dasselbe Programm kann gleichzeitig in verschiedenen Fenstern gemäß verschiedener Sichten dargestellt werden. Hierdurch ergibt sich z.B. die Möglichkeit, in einem Fenster Fehlermeldungen bezüglich des in einem anderen Fenster dargestellten Programms zu sehen. Die Selektion einer Fehlermeldung durch den Benutzer führt dann zu einer automatischen Selektion des korrespondierenden Programmteils im anderen Fenster.

Graphische Benutzungsoberfläche: Mittels des **Synthesizer Generator** erzeugte Editoren für Fenstersysteme (z.B. X11) enthalten eine graphische Benutzungsoberfläche. Eine solche Oberfläche bietet neben Pull-Down-Menüs und farbigen Knöpfen für eine einfache Bedienung via Maus die Möglichkeit der Verwendung verschiedener Farben und Schrifttypen für die Darstellung von Programmen.

EMACS-**ähnliches Äußeres und Bedienung:** Sowohl das Aussehen als auch die Bedienung eines mit dem **Synthesizer Generator** erstellten Editors sind stark an den weitverbreiteten konventionellen Editor *Emacs* [SBA92] angelehnt. Dadurch wird für viele Benutzer des Editors die Einarbeitungszeit erheblich verkürzt werden.

kontextsensitives Hypertext-Hilfesystem: Der **Synthesizer Generator** bietet die Möglichkeit, ein kontextsensitives Hilfesystem in generierte Editoren zu integrieren. Der Benutzer eines Editors kann von diesem Hilfesystem neben einer Erläuterung der Bedienung des Editors auch die Beantwortung von Fragen erwarten, die sowohl Syntax als auch Semantik der unterstützten Programmiersprache betreffen.

Konfigurierbarkeit über X-Ressourcen: Via Ressourcen können wesentliche Merkmale des Editors (z.B. Farben, Schriften, Tabulatorweite, Standardsicht.) vom Benutzer beeinflußt werden.

2.3.1 Aufbau

Es soll nun zunächst die Arbeitsweise des **Synthesizer Generator** und anschließend der Aufbau eines generierten Editors erläutert werden.

Arbeitsweise des Synthesizer Generator

Der **Synthesizer Generator** erwartet als Eingabe eine oder mehrere Dateien, die eine formale Spezifikation des zu generierenden Editors enthalten. Diese Spezifikation muß in der Sprache *Synthesizer-Specification-Language* (*SSL*)

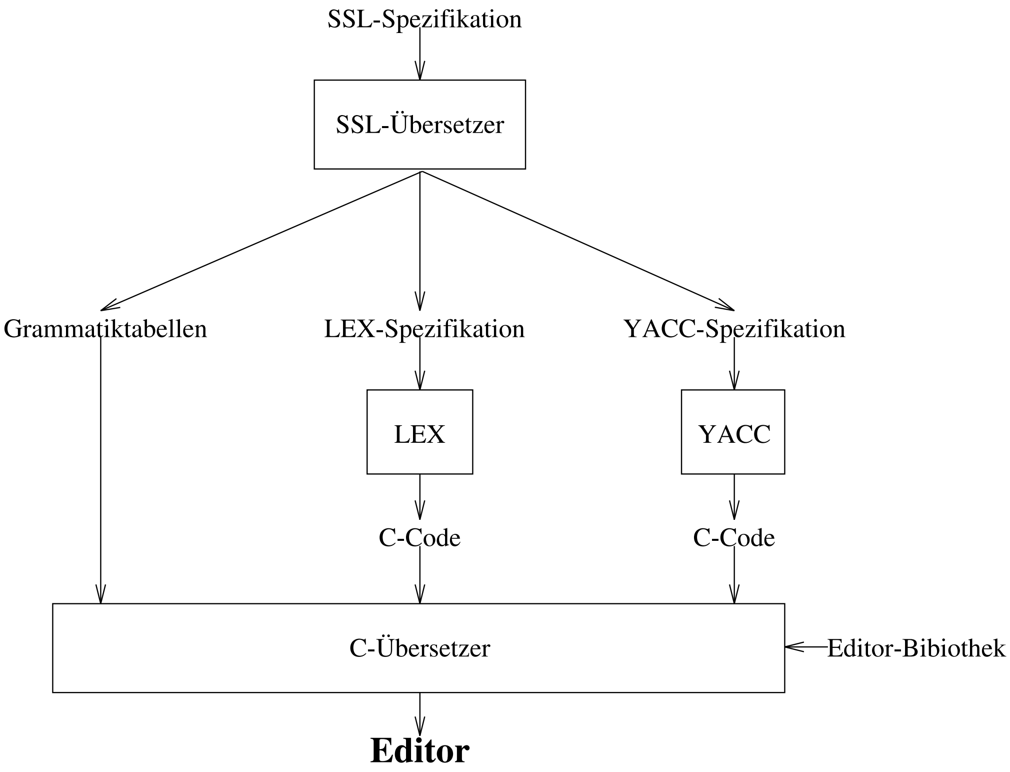

Abbildung 2.1: Die Arbeitsweise des **Synthesizer Generator**

formuliert werden. Basierend auf dieser Spezifikation ist es die Aufgabe des **Synthesizer Generator**, einen vollständigen Editor zu generieren. Dieser Vorgang kann in verschiedene Einzelschritte unterteilt werden: Zunächst gewinnt der SSL-Übersetzer aus der ursprünglichen Spezifikation neben verschiedenen Grammatiktabellen jeweils eine Spezifikation für den Scanner-Generator *LEX* [Les75] oder *FLEX* sowie für den Parser-Generator *YACC* [Joh75] bzw. *BISON*. Anschließend werden *LEX*/*FLEX* und *YACC*/*BISON* gestartet. Diese beiden Werkzeuge generieren aus den ihnen eingegebenen Spezifikationen C-Code. Dieser Code sowie die schon erwähnten Grammatiktabellen, die ebenso in C vorliegen, werden an den nun gestarteten C-Übersetzer übergeben. Im letzten Schritt werden zu den Ausgaben des C-Übersetzers verschiedene, den sprachenunabhängigen Code für den Editor enthaltende Editor-Bibliotheken gebunden (z.B. Oberfläche, AST-Verwaltung). Das Ergebnis dieses Vorgangs ist der ausführbare, syntaxgesteuerte Editor. Der gesamte Vorgang der Generierung ist in Abbildung 2.1 graphisch dargestellt. In der Abbildung sind die verwendeten Werkzeuge und die verschiedenen Dokumente, die von den Werkzeugen erzeugt bzw. verwendet werden, dargestellt. Werkzeuge sind von Dokumenten durch eine rechteckige Umrahmung unterschieden. Die Pfeile kennzeichnen den Datenfluß innerhalb des Systems.

Arbeitsweise eines generierten Editors

Im Kapitel 2.2.2 wurde bereits erläutert, daß syntaxgesteuerte Editoren meistens einen AST zur internen Darstellung eines Programmtextes verwenden. Auch ein mittels des **Synthesizer Generator** erzeugter Editor (im folgenden **Synthesizer Generator**-Editor genannt) verwendet diese Darstellung. Ebenfalls die im selben Kapitel erwähnte Attributierung des AST kann bei **Synthesizer Generator**-Editoren verwendet werden. So können an beliebige Knoten des AST semantische Informationen gebunden sein. Der **Synthesizer Generator** analysiert die Abhängigkeiten der Attribute voneinander und vom AST selbst und erstellt verschiedene Tabellen, die von einem inkrementellen Algorithmus des **Synthesizer Generator**-Editor verwendet werden, um teure Neuberechnungen von Attributwerten auf ein Minimum zu reduzieren.

2.3.2 Spezifikation eines Editors

Zur Generierung eines auf eine bestimmte Programmiersprache abgestimmten Editors benötigt der **Synthesizer Generator** verschiedene Informationen bezüglich dieser Sprache. Ebenso müssen spezielle Darstellungsformen und Benutzerschnittstellen spezifiziert werden. Die komplette Spezifikation eines mittels des **Synthesizer Generator** zu generierenden Editors läßt sich in fünf Kategorien einteilen, die in SSL formuliert werden müssen:

- Abstrakte Syntax
- Konkrete Syntax
- Attributierung
- Unparsing
- Transformationen

Während diese Kategorien unter SSL syntaktisch klar voneinander getrennt sind, können Syntax und Semantik der zugehörigen Programmiersprache nicht eindeutig diesen Einheiten zugeordnet werden. So stellt die Teilung der Syntax einer Programmiersprache in abstrakte und konkrete Syntax eine Entwurfsentscheidung dar, die nicht zwingend durch die Sprache selbst vorgegeben ist, sondern u.a. in Bezug auf die Bedienung des zu erstellenden Editors gefällt werden muß. Auch lassen sich oft syntaktische Eigenschaften der Sprache einfacher unter Zuhilfenahme der Attributierung formulieren, als alleine mit abstrakter und konkreter Syntax. Die größte Entscheidungsfreiheit bietet jedoch die Spezifikation von Unparsing und Transformationen. Hier sind dem Editor-Entwickler im Rahmen der obengenannten Anforderungen an den Editor alle kreativen Freiheiten gegeben.

Prinzipiell ist es möglich, alle obengenannten Teile der Spezifikation eines Editors beliebig textuell miteinander zu vermischen. Es ist z.B. erlaubt, die Spezifikation für arithmetische Ausdrücke zusammenzufassen und in einer separaten

```
expr: ExprNil()
    | IntConst(INT)
    | FloatConst(REAL)
    | Sum,Diff,Prod,Div(expr expr);
```

Abbildung 2.2: Beispiel 1 für die Spezifikation der abstrakten Syntax von arithmetischen Ausdrücken

Datei abzulegen. Dadurch wird es möglich, die SSL-Spezifikation eines Editors zu modularisieren. Hier sind dieselben Vorteile zu erwarten, die durch die Modularisierung von Programmen gegeben sind.

Abstrakte Syntax

Um einen Editor für eine spezifische Programmiersprache zu erstellen, ist es notwendig, die abstrakte Syntax dieser Sprache für den **Synthesizer Generator** verständlich zu spezifizieren. Jede künftige Editorsitzung wird auf diese abstrakte Syntax in der Form aufbauen, daß alle zu editierenden Objekte durch abstrakte Syntaxbäume repräsentiert werden, die gemäß der abstrakten Grammatik aufgebaut werden. Jede Änderung eines Objektes korrespondiert mit einer Änderung des repräsentierenden AST, die immer durch Anwendung einer Produktion der abstrakten Syntax beschrieben werden kann. In der Terminologie des **Synthesizer Generator** entsprechen die Nichtterminalsymbole einer Produktion einer in EBNF gegebenen Syntax den sogenannten *Phyla*. Jede Produktion der abstrakten Syntax erhält einen identifizierenden Namen. Eine Produktion mit ihrem Namen wird *Operation* genannt. Die Operanden einer Operation entsprechen der rechten Seite einer Produktion und sind selbst wiederum Phyla. Die Deklaration einer Produktion der abstrakten Syntax geschieht in folgenden Form:

$$\textit{Phylum-Name}\texttt{:}\ \ \textit{Operator-Name}\texttt{(}\textit{Phylum}_1\ \textit{Phylum}_2\ \ldots \textit{Phylum}_k\texttt{);}$$

Diese Deklaration entspricht der Produktion

$$\textit{Phylum-Name} \rightarrow \textit{Phylum}_1\ \textit{Phylum}_2\ \ldots \textit{Phylum}_k$$

In Abbildung 2.2 wird die abstrakte Syntax einfacher arithmetischer Ausdrücke als erläuterndes Beispiel in SSL spezifiziert. Hier wird eine in SSL erlaubte stark verkürzte Schweibweise verwendet, die verschiedene alternative Produktionen für ein Phylum durch den Senkrechtstrich oder Komma voneinander trennt.

So existieren für das Phylum `expr` sieben Produktionen. Die Bezeichner `INT` und `REAL` sind vordefinierte Phyla und entsprechen den Terminalsymbolen der konkreten Syntax. Die Operation `ExprNil()` spiegelt sich demgegenüber nicht in der konkreten Syntax wieder. Sie ist vielmehr eine Produktion, durch die ein

```
expr      : ExprNil()
          | IntExpr(intExpr)
          | FloatExpr(floatExpr);

intExpr   : IntExprNil()
          | IntConst(INT)
          | IntSum,IntDiff,IntProd,IntDiv(intExpr intExpr);

floatExpr: FloatExprNil()
          | FloatConst(REAL)
          | FloatSum,FloatDiff,FloatProd,
            FloatDiv(floatExpr floatExpr);
```

Abbildung 2.3: Beispiel 2 für die Spezifikation der abstrakten Syntax von arithmetischen Ausdrücken

nicht-expandierter Ausdruck im AST dargestellt wird. Die in diesem Beispiel gegebene abstrakte Syntax erlaubt die Anwendung der vier Rechenoperationen auf ganzzahlige wie auf reelle Konstanten. Die Spezifikation einer streng typisierten Programmiersprache kann diese abstrakte Syntax für arithmetische Ausdrücke enthalten. In diesem Falle wird jedoch die eigentlich unzulässige Anwendung der Operatoren auf Konstanten verschiedener Typen nicht von vornherein durch die abstrakte Syntax unterbunden. Durch geeignete semantische Regeln, realisiert über die Attributierung (siehe Kapitel 2.3.2), können jedoch diese Typfehler trotzdem vermieden werden oder zumindest kann durch die Ausgabe von Fehlermeldungen auf sie hingewiesen werden. In Abbildung 2.3 wird die Spezifikation so erweitert, daß die korrekte Verwendung der Typen schon durch die abstrakte Syntax erzwungen wird.

Dieses Beispiel zeigt, daß die Spezifikation der abstrakten Syntax einer Sprache nicht nur durch die Sprache selbst bestimmt ist, sondern vielmehr eine komplexe Entwurfsentscheidung darstellt.

Attributierung

Wie bereits in Kapitel 2.3.1 erläutert kann eine kontextfreie Grammatik um an Nichtterminalsymbole oder an Produktionen gebundene Attribute und um Attributgleichungen zu einer attributierten Grammatik (AG) erweitert werden, um semantische Regeln der zugrunde liegenden Sprache zu formulieren.

Während bei attributierten Grammatiken [Kas80, Kas90] nur zwei Typen von Attributen (geerbte oder synthetisierte) existieren, gibt es in SSL derer vier: Geerbte Attribute, synthetisierte Attribute, lokale Attribute sowie Nichtterminalsymbolattribute. Eine Einschränkung gegenüber AGs stellen zwei Anforderung an die Attributgleichungen in SSL dar: Wie in *ordered attribute grammars*(OAGs) [Knu68] müssen Attributgleichungen wohldefiniert sein und dürfen keine zyklischen Abhängigkeiten aufweisen. Die neu eingeführten Attributtypen erhöhen die Mächtigkeit der Attributierung nicht, sie dienen lediglich einer vereinfachten Spezifikation.

Die auch in AGs bekannten, in [Kas90] *erworben* oder *abgeleitet* genannten, geerbten oder synthetisierten Attribute sind immer an Phyla gebunden. Der Wert eines synthetisierten Attributs wird durch eine Attributregel bestimmt, die zu einer Produktion gehört, auf deren linker Seite das Nichtterminalsymbol steht, an das das Attribut gebunden ist. Demgegenüber sind geerbte Attribute immer an Symbole gebunden, die auf der rechten Seite der jeweiligen Produktionen stehen.

Im Gegensatz zu den synthetisierten und den geerbten Attributen sind die lokalen Attribute nicht an Phyla sondern an Produktionen der abstrakten Syntax gebunden. Dies ist z.B dann sinnvoll, wenn die semantische Korrektheit der Anwendung einer Produktion festgehalten werden soll. Es ist wohl möglich, eine derartige Information mittels eines synthetisierten Attributes, das an das auf der linken Seite der Produktion auftretende Nichtterminalsymbol gebundenen ist, zu speichern. Dies hat jedoch den Nachteil, daß für jedes linksseitige Auftreten dieses Nichtterminalsymbols Attributgleichungen zu spezifizieren sind, die während der Ausführung des Editors natürlich auch ausgewertet werden, obwohl die gewonnene, zu speichernde Information evtl. unnötig ist.

Der vierte in SSL bekannte Attributtyp ist das Nichtterminalsymbolattribut. Diese Attribute sind ebenfalls an Produktionen gebunden. Im Gegensatz zu den lokalen Attributen sind sie jedoch selbst Teil der Produktion und damit des AST, indem sie selbst ein Nichtterminalsymbol darstellen. Dieser Attributtyp ist jedoch für die Spezifikation des ProSet-Editors von geringer Bedeutung und soll daher hier nicht weiter motiviert oder erläutert werden.

Nachdem nun die Attribute gemäß ihrer Zuordnung zum AST unterschieden wurden, soll nun kurz auf die Wertebereiche von Attributen eingegangen werden: Attribute können nicht nur Werte einfacher Typen wie `CHAR`, `STR` oder `INTEGER` aufnehmen. Vielmehr ist es möglich, einem Attribut ein in der abstrakten Grammatik definiertes Phylum als Typ zuzuordnen. Dadurch wird es möglich, Attribute zu definieren, die ganze Teil-ASTs referenzieren. Diese Möglichkeit wird besonders in der Spezifikation der konkreten Syntax ihre Verwendung finden.

Attribute müssen in SSL zunächst explizit deklariert werden, erst danach kann auf sie zugegriffen werden. Zur Attributberechnung stellt SSL Fallunterscheidungen, verschiedene vordefinierte SSL Funktionen und Operatoren sowie die Möglichkeit, in C geschriebene Funktionen aufzurufen, zur Verfügung. Innerhalb der Berechnungsvorschriften für Attribute ist es nicht nur möglich, über sogenannte *remote attribute sets* auf Attribute, die sich im AST oberhalb des zu berechnenden Attributs befinden, zuzugreifen. Durch die Verwendung von *Patterns* oder *Mustern* ist es möglich, auch auf Attribute von Nachfahren zuzugreifen. Ein Muster besteht hierbei aus der verschachtelten Anwendung von Produktionen auf eine oder mehrere Variablen. Bei der Attributberechnung wird versucht, den Teil des AST, dessen Wurzel zu der Produktion bzw. zu dem Nichtterminalsymbol gehört, dem das zu berechnende Attribut zugeordnet

```
/* Attributdeklaration */

expr  { synthesized ETYPE type; };

/* abstrakte Syntax */     /* Attributierung */

expr: ExprNil()            { expr.type=ET_UNKNOWN; }
    | IntConst(INT)        { expr.type=ET_INT; }
    | FloatConst(REAL)     { expr.type=ET_REAL; }
    | Sum,Diff,Prod,Div(expr expr)
    { expr$1.type=(TypesCompatible(expr$1.type,expr$2.type))
                  ? (expr$1.type!=ET_UNKNOWN)
                    ? expr$1.type
                    : expr$2.type
                  : ET_UNKNOWN;
      local STR error;
      error=(TypesCompatible(expr$1.type,expr$2.type))
            ? ""
            : "type-mismatch"; };
```

Abbildung 2.4: Beispiel der Attributierung für streng typisierte arithmetische Ausdrücke

ist, auf ein Muster abzubilden. Die benutzten Variablen werden dann mit den aktuellen Werten belegt. Auf diese kann schließlich für weitere Berechnungen zugegriffen werden. Dieser leistungsfähige Mechanismus des *Pattern Matching* wird auch bei der Spezifikation von Transformationen verwendet, bei deren Erläuterung auch ein einfaches Beispiel hierfür zu finden sein wird.

Als Beispiel für die Attributierung diene nun die Typinferenz bzw. Typprüfung der schon in Kapitel 2.3.2 erwähnten Ausdrücke, wie in Abbildung 2.4 dargestellt. Der Typ eines zusammengesetzten Ausdrucks soll anhand der Typen seiner Teilausdrücke bestimmt werden. Ein zusammengesetzter Ausdruck habe den Typ `ET_UNKNOWN`, genau dann, wenn die Typen beider Teilausdrücke `ET_UNKNOWN` ist oder die Teilausdrücke nicht typverträglich sind. Aus Gründen der Übersichtlichkeit wird hier auf die Einführung eines zusätzlichen Werts `ET_BADTYPE` zur Indikation eines Typfehlers verzichtet. Jedem Phylum `expr` wird ein Attribut `type` des Typs `ETYPE` zugeordnet. `ETYPE` sei ein enumerativer Typ und an anderer Stelle definiert. Nicht-expandierte Ausdrücke erhalten den Typ `ET_UNKNOWN`. Sie sind typkompatibel zu den anderen Typen. Konstanten erhalten den Typ `ET_INT` bzw. `ET_REAL`.

Der Typ eines Ausdrucks, der durch die Anwendung der Produktionen `Sum`, `Diff`, `Prod` oder `Div` gebildet wird, ergibt sich folgendermaßen: Die Funktion `TypesCompatible` sei an anderer Stelle definiert und gebe genau dann den booleschen Wert `false` (in SSL existiert ein Typ `BOOL`) zurück, wenn der eine

Operator der Typ `ET_INT` und der andere der Typ `ET_REAL` ist. Falls diese Funktion die beiden Operanden als typverträglich ausweist, wird der Typ des ersten Operanden zugewiesen (sofern dieser nicht `ET_UNKNOWN` ist, ansonsten wird der Typ des zweiten Operanden zurückgegeben). Sind die beiden Operanden jedoch nicht typverträglich, so wird `ET_UNKNOWN` ermittelt.

Den Produktionen `Sum`, `Diff`, `Prod` oder `Div` ist außerdem ein lokales Attribut `error` zugeordnet. Diesem wird im Falle einer Typunverträglichkeit eine Fehlermeldung zugewiesen.

Wie im Kapitel 2.3.2 noch erläutert werden wird, kann dieses Attribut mittels geeigneten Unparsings vom Editor ausgegeben werden.

Konkrete Syntax

Die abstrakte Syntax einer Sprache dient im wesentlichen der internen Darstellung von Texten dieser Sprache. Der Vorgang der Überführung einer textuellen Darstellung eines Programms in einen AST wird *Parsing* genannt. Hierfür müssen weitere Regeln bzw. Produktionen, in ihrer Form und Semantik den Regeln der abstrakten Syntax ähnlich, aufgestellt werden. Unter anderem diese Produktionen bilden die konkrete Syntax einer Sprache. Sie werden jedoch nicht auf einzelne Zeichen eines Textes angewendet: Zuvor werden die einzelnen Zeichen des Textes mittels eines *Scanners* in atomare Einheiten (*Lexeme*) aufeinanderfolgender Zeichen gruppiert. Der Scanner verwendet hierfür reguläre Ausdrücke (Lexem-Deklarationen), die Bestandteil der Spezifikation der konkreten Syntax einer Sprache sind.

In SSL können Lexemdeklarationen folgender Syntax entsprechend angegeben werden:

Phylum-Name`:<` *regulärer Ausdruck* `>;`

Ein ProSet-Bezeichner wird also folgendermaßen spezifiziert:

```
IDENTIFIER: < [A-Za-z][A-Za-z0-9_]* >;
```

Der obige reguläre Ausdruck wird vom **Synthesizer Generator** direkt an den unterliegenden Scanner-Generator *LEX* bzw. *FLEX* übergeben. Ein Bezeichner ist in diesem Beispiel eine Zeichenkette, bestehend aus einem Buchstaben, gefolgt von einer beliebigen Anzahl von Buchstaben oder Ziffern oder dem Unterstrich `"_"`.

Die einzelnen so gewonnenen Lexeme sind Operanden der Produktionen der konkreten Syntax. Die Syntax der Spezifikation dieser Produktionen entspricht in SSL im wesentlichen der Spezifikation der abstrakten Syntax. So unterscheiden die Parsing-Regeln sich von den Regeln zur abstrakten Syntax nur in der Verwendung von `"::="` statt `":"` zur Trennung von linker und rechter Seite. So entspricht hier analog die Deklaration

$$\textit{Phylum-Name}\texttt{::=(}\textit{Phylum}_1\ \textit{Token}_1\ \textit{Phylum}_2\ \ldots\ \textit{Phylum}_n\texttt{);}$$

der EBNF-Produktion

$$\textit{Phylum-Name} \rightarrow \textit{Phylum}_1\ \textit{Token}_1\ \textit{Phylum}_2\ \ldots\ \textit{Phylum}_n\,.$$

Durch Anwendung der Produktionen der konkreten Syntax wird aus einer Folge von Lexemen ein sogenannter *Parsing-Baum* aufgebaut. Dieser Parsing-Baum muß anschließend in den AST überführt werden. Dies geschieht beim **Synthesizer Generator** ebenfalls mittels Attributierung:

Jedem Phylum der Parsing-Regeln können synthetisierte Attribute zugeordnet werden, deren Werte bei Anwendung von Produktionen der konkreten Syntax, auf deren linker Seite das Phylum steht, bestimmt werden. Hierfür können die Operationen der abstrakten Syntax (und auch andere, z.B. vordefinierte SSL-Operationen) auf die rechtsseitigen Phyla oder deren Attribute angewendet werden. Auf diese Weise kann aus dem Parsing-Baum sukzessive der korrespondierende AST konstruiert werden. Dieser Vorgang sollte anhand des in Abbildung 2.5 dargestellten Beispiels deutlich werden. Hier wird die konkrete Syntax der schon mehrfach betrachteten arithmetischen Ausdrücke spezifiziert.

Zunächst wird das synthetisierte Attribut `e` deklariert. Dieses Attribut kann eine Referenz auf einen AST für das Phylum `expr` der abstrakten Syntax aufnehmen und ist an das Phylum `Expr` der konkreten Syntax gebunden. Die beiden mit dem Schlüsselwort `left` beginnenden Zeilen des Beispiels geben die Assoziativität und die Präzedenz der vier Grundrechenarten an. Anschließend werden die einzelnen alternativen Produktionen für das Phylum `Expr` der konkreten Syntax angegeben. Jeder Alternative folgt eine Attributgleichung für einen korrekten Aufbau des Teil-AST. Die Phyla `INTEGER` und `FLOAT` seien bereits definiert, während `STRtoINT` und `STRtoREAL` vordefinierte SSL-Funktionen zur Umwandlung der Lexeme nach `INT` bzw. `REAL` sind. Die Verwendung des Dollarzeichen innerhalb einer Attributgleichung entspricht der von *YACC*: `"$$"` bezeichnet das Phylum auf der linken Seite einer Produktion und die Anhängung von `"$n"` an den Namen eines Phylums entspricht dem n-ten in der Produktion auftretenden Phylum dieses Namens. Die letzte Zeile des Beispiels gibt an, an welcher Stelle des AST der neu gewonnene Teil-AST einzufügen ist.

Unparsing

Abstrakte wie konkrete Grammatik sowie die Attributierung dienen der Spezifikation der internen Darstellung eines Programmes mittels eines attributierten Syntaxbaums. Ein auf einem solchen AST arbeitendender Editor muß in der Lage sein, eine textuelle Repräsentation des aktuellen AST zu erzeugen. Dieser Vorgang wird *Unparsing* genannt. Da es für die textuelle Darstellung eines Programms je nach Spezifikation der konkreten Syntax (z.B. der gültigen Trennzeichen) beliebig viele Möglichkeiten gibt, und da die konkrete Syntax der zugrunde liegenden Programmiersprache nicht vollständig spezifiziert sein muß,

```
/* Attributdeklaration */
Expr { synthesized expr e; };

/* Assoziativitaet, Praezedenz */
left '+','-';
left '*'.'/';

/* konkr. Syntax */     /* Aufbau des Baums
Expr::= (INTEGER)       { $$.e=IntConst(STRtoINT(INTEGER)); }
     |   (FLOAT)        { $$.e=FloatConst(STRtoREAL(FLOAT)); }
     |   (Expr '+' Expr) { $$.e=Sum(Expr$2.e,Expr$3.e; }
     |   (Expr '-' Expr) { $$.e=Diff(Expr$2.e,Expr$3.e; }
     |   (Expr '*' Expr) { $$.e=Prod(Expr$2.e,Expr$3.e; }
     |   (Expr '/' Expr) { $$.e=Div(Expr$2.e,Expr$3.e; };

/* Zuordnung zum AST */
expr ~ Expr.e;
```

Abbildung 2.5: Beispiel für die konkrete Syntax für arithm. Ausdrücke

wird mittels der Aufstellung geeigneter Unparsing-Regeln die textuelle Darstellung spezifiziert. Da eventuell mehrere verschiedene Ansichten eines Programms erwünscht sind, ist es möglich, verschiedene *Sichten* oder *Views* zu definieren und für verschiedene Sichten verschiedene Unparsing-Regeln zu spezifizieren. Standardmäßig ist durch den **Synthesizer Generator** jedoch nur eine Sicht namens `BASEVIEW` vorgegeben. Für jede Produktion der abstrakten Syntax kann wiederum für jede Sicht eine Unparsing-Regel spezifiziert werden. Eine solche Unparsing-Regel hat wie die Regeln der abstrakten Syntax eine linke und eine rechte Seite. Diese beiden Seiten werden durch die Zeichen(-folgen) `":"` oder `"::="` getrennt. Diese Trennzeichen geben an, ob für das linksseitige Phylum der korrespondierenden Regel der abstrakten Syntax eine textuelle Eingabe möglich sein soll. Jede Regel enthält mehrmals die Zeichen `"@"` oder `"^"`. Diese repräsentieren die Phyla der entsprechenden Regel der abstrakten Syntax. Sie geben an, ob das jeweilige Phylum durch den Benutzer selektierbar sein soll oder nicht. Schließlich kann eine Unparsing-Regel verschiedene Terminalzeichen und Steuersequenzen enthalten. Letztere können z.B. einen Zeilenumbruch oder eine Einrückung des Textes einleiten.

Zur Verdeutlichung diene nun die in Abbildung 2.6 gezeigte Spezifikation der Unparsing-Regeln für die einfachen arithmetischen Ausdrücke. Hier ist es möglich, sowohl einen kompletten Ausdruck, wie auch jeden syntaktischen Teil eines Ausdrucks einzeln zu selektieren und textuell einzugeben bzw. zu modifizieren. Für die Funktionen auf zwei Operanden wird zusätzlich nach einem eventuellen Zeilenumbruch der Wert des Attributes `error` ausgegeben. Die Steuersequenz `"%o"` erzeugt einen Zeilenumbruch hinter einem mathematischen

```
expr: ExprNil              [@::="<Expr>"]
      IntConst,FloatConst  [@::=@]
    | Sum                  [@::=@ "+%o" @ error]
    | Diff                 [@::=@ "-%o" @ error]
    | Prod                 [@::=@ "*%o" @ error]
    | Div                  [@::=@ "/%o" @ error];
```

Abbildung 2.6: Beispiel für das Unparsing der arithm. Ausdrücke

```
transform <expr> on "+":Sum(<expr>,<expr>),
                 on "-":Diff(<expr>,<expr>),
                 on "*":Prod(<expr>,<expr>),
                 on "/":Div(<expr>,<expr>);
```

Abbildung 2.7: Beispiel für Transformationen für arithm. Ausdrücke

Operator, falls die Ausgabe ansonsten die gewünschte Zeilenbreite überschreiten würde.

Transformationen

Falls ein Programm menügesteuert erstellt werden soll oder die komplexe Umstrukturierung einzelner Programmteile ermöglicht werden soll, müssen für den Editor *Transformationsregeln* spezifiziert werden. Falls die aktuelle strukturelle Selektion des Programms dem Muster der linken Seite einer solchen Transformationsregel entspricht, wird dem Benutzer ein zusätzlicher Menüpunkt angeboten. Wird dieser gewählt, so wird das selektierte Programmstück durch die rechte Seite der Transformationsregel ersetzt.

Abbildung 2.7 zeigt einige Transformationen zum menügesteuerten Aufbau der arithmetischen Ausdrücke. Die spitzen Klammern um `expr` kennzeichnen hier einen nicht-expandierten Ausdruck. Falls der Benutzer einen solchen selektiert, werden ihm vom Editor die Transformationen `"+"`, `"-"` usw. zum Erzeugen einer Summe, einer Differenz usw. angeboten. Dieses Beispiel zeigt nicht die leistungsfähigen Möglichkeiten, die der Transformationsmechanismus des **Synthesizer Generator** bietet. Denn sowohl die linke als auch die rechte Seite einer Transformationsregel kann auch verschachtelte Funktionsaufrufe enthalten.

2.3.3 Bedienung eines Editors

Ein mithilfe des **Synthesizer Generator** erstellter Editor ähnelt sowohl bezüglich des Aussehens seiner Oberfläche als auch in seiner Bedienung sehr

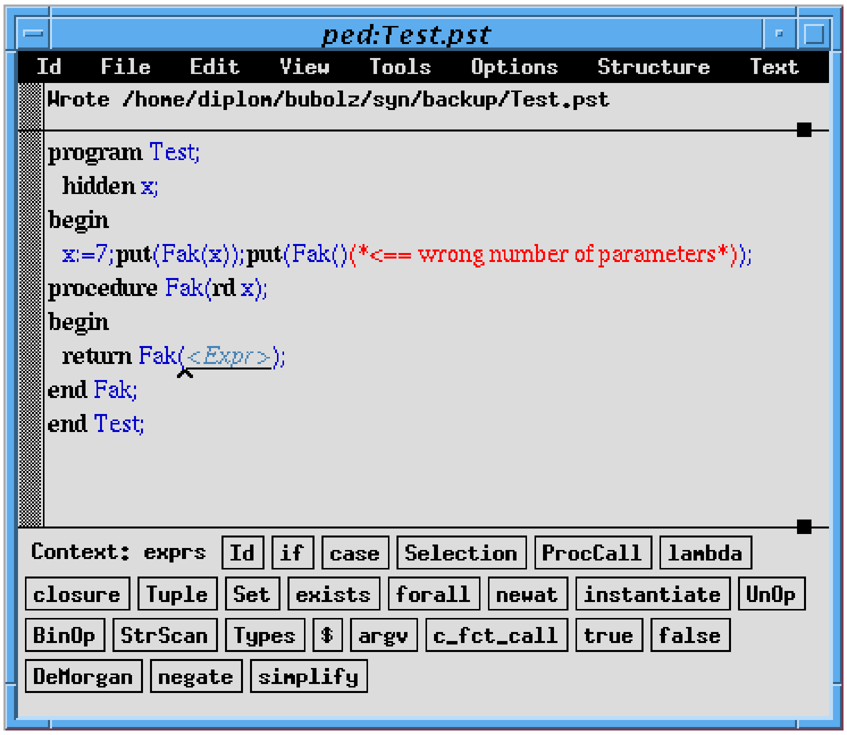

Abbildung 2.8: Die Oberfläche des ProSet-Editors

stark dem Editor *EMACS*. So unterteilt sich die Benutzungsoberfläche des Editors wie in Abbildung 2.8 zu sehen in vier vertikale Bereiche: Oben ist die Menüleiste des Editors zu finden, in der sich das Hauptmenü befindet. Bei Anklicken eines Auswahlpunktes mit der Maus klappt dort jeweils ein Pull-Down-Menü herunter, über das die wichtigsten Funktionen des Editors zu erreichen sind. Unterhalb der Menüleiste befindet sich ein auf der Abbildung freier Raum, innerhalb dessen der Editor eventuelle Meldungen ausgibt. Der eigentliche Programmtext befindet sich im größten Bereich in der vertikalen Mitte der Oberfläche. Neben der üblichen Einfügemarke, die dem Benutzer anzeigt, wo (falls möglich) die nächste textuelle Eingabe das Programm modifizieren wird, wird ein ganzer die Einfügemarke umschließender Bereich des Programmtextes durch Unterstreichung hervorgehoben. Dieser Bereich korrespondiert immer mit einem Inkrement, also einem Teilbaum des aktuellen, das Programm repräsentierenden AST und wird strukturelle Selektion genannt. Strukturorientierte Änderungen des Programmtextes werden immer im Kontext der aktuellen strukturellen Selektion vorgenommen. Der Name der aktuellen strukturellen Selektion gemäß der Spezifikation der abstrakten Syntax wird im letzten unteren Bereich der Benutzungsoberfläche dargestellt. In diesem Bereich werden außerdem die Namen der Operationen und Transformationen, die in dem aktuellen Kontext erlaubt sind, als Auswahlknöpfe angezeigt. Diese Ansammlung von Knöpfen wird innerhalb dieser Arbeit ebenfalls als Menü bezeichnet werden. Durch Anklicken einer dieser Knöpfe wird die Selektion dann entsprechend expandiert bzw. gemäß der Transformationsregeln transformiert. Da die strukturelle Selektion immer die textuelle Einfügemarke umschließt, kann sie mit dieser, also mithilfe normaler Cursor-Bewegung verschoben werden. Einfacher ist es jedoch, spezielle Befehle zu benutzen, die eine Bewegung der strukturellen Selektion gemäß dem korrespondierenen AST erlauben. So ist es z.B. möglich, den nächsten oder vorigen Haltepunkt, d.h. einen Knoten im AST, der Wurzel einer strukturellen Selektion sein kann, gemäß der Unparsing-Regeln innerhalb oder außerhalb der aktuellen strukturellen Selektion auszuwählen und zur aktuellen strukturellen Selektion zu machen. Wird vom Benutzer eine alphanumerische Taste gedrückt, wechselt der Editor, falls an dieser Stelle erlaubt, in den textuellen Eingabemodus. Ansonsten wird der Benutzer durch eine Fehlermeldung informiert. Beendet der Benutzer z.B. durch Auswahl einer neuen strukturellen Selektion den textuellen Modus, wird die aktuelle strukturelle Selektion geparst. Ist dies nicht erfolgreich möglich, informiert hierüber eine Fehlermeldung. In diesem Falle bleibt die fehlerhafte Struktur farblich hervorgehoben.

Um dem Benutzer die Wahl der Darstellung des Programmtextes zu ermöglichen, gibt es einen Befehl, der alle verfügbaren Ansichten auflistet, und dann den Wechsel in die ausgewählte Darstellung ermöglicht. Es kann dabei ein Programm in verschiedenen Fenstern verschieden dargestellt werden. Das Programm kann außerdem (z.B. zum kompilieren) in der jeweiligen textuellen Dastellung abgespeichert oder gedruckt werden.

Die normalen Hilfefunktionen wie sie auch vom *EMACS* angeboten werden, können um sprachspezifische Hilfetexte erweitert werden. Es ist möglich für alle selektierbaren Kontexte, alle möglichen Darstellungsformen sowie für alle Transformationen Hilfetexte anzulegen, die dann kontextsensitiv vom Benutzer eingesehen werden können.

2.3.4 Vergleich mit anderen Systemen

Neben dem **Synthesizer Generator** existieren noch verschiedene andere Systeme, die der Generierung von Programmentwicklungsumgebungen dienen. Die bekanntesten dieser Systeme sind *Centaur* [BC88], *Gandalf* [HN86], *Mjolner* [KLLMM94], *Pan* [BGdV92], *PECAN* [Rei85] und *Progres* [ELN+92].

Zwei dieser Systeme sollen nun stellvertretend kurz charakterisiert werden. Es wurden hierfür *Centaur* und *Pan* gewählt. Diese beiden Produkte zeichnen sich durch einen fortgeschrittenen Grad der Entwicklung aus. Sie sind ebenso wie der **Synthesizer Generator** für Unix verfügbar und besitzen eine graphische, menügesteuerte und fensterorientierte Benutzungsoberfläche. Daher können sie als echte Alternativen zum **Synthesizer Generator** betrachtet werden.

Die beiden Systeme sollen nun jeweils innerhalb vierer Abschnitte vorgestellt werden. Der erste Abschnitt wird jeweils eine allgemeine Charakterisierung darstellen. Der zweite Abschnitt wird auf grundlegende Aspekte der Spezifikation, die das jeweilige Werkzeug als Eingabe verwendet, eingehen. Der dritte Abschnitt wird die Benutzungsschnittstelle einer mit diesen Systemen generierten Programmentwicklungsumgebung betrachten. Im letzten Abschnitt wird jeweils die Möglichkeit der Integration von Werkzeugen untersucht. Eine Gegenüberstellung dieser beiden Systeme mit dem **Synthesizer Generator** ist schließlich in Tab.2.1 dargestellt.

Informationen für diesen Vergleich wurden [BC88, BGdV92, SL93] und [Bud92] entnommen.

Centaur

Allgemeine Charakterisierung: *Centaur* ist ein mittlerweile kommerzielles Produkt, welches aus dem seit 1974 am Institut National de Recherche en Informatique et en Automatique (INRIA) in Frankreich entwickelten *MENTOR*-System [DGGKL84, DGKLM84] hervorgegangen ist.

Eine Programmentwicklungsumgebung, die mit *Centaur* erstellt wurde, besteht aus drei nicht inkrementell arbeitenden Einzelwerkzeugen, die der Verwaltung von abstrakter und konkreter Syntax, dem Unparsing und der Analyse von statischer und dynamischer Semantik dienen. Diese drei Werkzeuge arbeiten voneinander unabhängig unter einer gemeinsamen Oberfläche.

Spezifikation: Die Spezifikation für *Centaur* kann aus bis zu drei Teilspezifikationen für die drei erwähnten Einzelwerkzeuge bestehen.

Die Spezifikation von abstrakter und konkreter Syntax sowie deren Assoziation geschieht entweder mittels der Sprache *METAL* oder mit *SDF*. Während mittels *METAL* abstrakte und konkrete Syntax voneinander unabhängig spezifiziert werden können, wird die abstrakte Syntax bei Verwendung der neueren Sprache *SDF* von der konkreten Syntax abgeleitet.

Die Spezifikation des Unparsing geschieht mittels der Sprache *PPML*. Diese Spezifikation ist für jedes Phylum optional. Nicht-spezifizierte Phyla werden gemäß vorgegebener Unparsing-Regeln ausgegeben.

Die Semantik wird in der Sprache *TYPOL* beschrieben. *TYPOL* stellt die Implementation der *Natural Semantics* [Kah87] dar. Hier werden im Prinzip eine Reihe von Axiomen sowie Inferenzregeln für diese angegeben. Diese werden dann vom *TYPOL*-Übersetzer in Prolog-Terme umgewandelt.

Die Gesamtspezifikation fur *Centaur* kann recht kurz gehalten werden, wenn keine gesonderten Unparsing-Regeln angegeben werden und wenn *SDF* als Beschreibungssprache für abstrakte und konkrete Syntax zugleich verwendet wird. Allerdings dürfte die Verwendung von insgesamt drei verschiedenen Sprachen eine umfangreichere Spezifikation ziemlich erschweren.

Benutzungsschnittstelle: Da die Einzelwerkzeuge von *Centaur* voneinander unabhängig arbeiten, müssen sie einzeln vom Benutzer aktiviert werden. Dies bedeutet, daß z.B. die Überprüfung der Korrektheit der statischen Semantik eines Programmes vom Benutzer explizit angefordert werden muß. Die textuelle Eingabe von Programmeinheiten geschieht in einem separaten Fenster. Fehlerhafte Programmteile werden markiert innerhalb des restlichen Programmtextes dargestellt. Ein strukturorientiertes Edieren ist nur durch Kopieren des selektierten Inkrements in die Zwischenablage, anschließende textuelle Bearbeitung und Zurückkopieren möglich. Damit erscheint die Oberfläche eines mittels *Centaur* generierten Editors sehr unpraktisch.

Integration von Werkzeugen: Die drei Einzelwerkzeuge von *Centaur* kommunizieren über Unix-Pipes miteinander. Daher wird sich die Anbindung bzw. Integration weiterer Werkzeuge nur mäßig aufwendig gestalten.

Pan

Allgemeine Charakterisierung: *Pan* ist ein an der University of California, Berkeley entwickeltes Werkzeug, das als syntaxgesteuerter Editor und Browser die Benutzungsoberfläche einer integrierten Entwicklungsumgebung darstellt. *Pan* basiert wie der *Emacs* auf einem Lisp-Kern und ist wie dieser konfigurierbar und erweiterbar.

Spezifikation: Eine Spezifikation für *Pan* besteht aus bis zu drei Einzelspezifikationen für die konkrete Syntax, abstrakte Syntax sowie für semantische Regeln. Die konkrete Syntax muß hierbei jedoch nicht zwingend vollständig angegeben sein. Nicht-spezifizierte Teile der konkreten Syntax generiert *Pan* mittels einer Technik namens „Grammatical Abstraction" automatisch aus der entsprechenden abstrakten Syntax. Regeln für das Unparsing können nicht explizit vorgegeben werden, da diese ebenfalls automatisch aus der abstrakten Grammatik gewonnen werden. Somit besteht eine Spezifikation für Pan nur aus der abstrakten Grammatik, sofern keine semantischen Regeln aufgestellt werden.

Die abstrakte und konkrete Syntax (und damit das Unparsing) geschieht mittels der Sprache *Ladle*, während die semantische Spezifikation in der Sprache *Colander* vorliegen muß. *Colander* basiert auf den *Logical Constraint Grammars*. Hier werden den Produktionen einer kontextfreien Grammatik verschiedene Zielaussagen (*goals*) zugeordnet. Weitere solche *goals* können unabhängig von diesen Produktionen angegeben werden. Das Laufzeitsystem von *Pan* versucht schließlich alle *goals* mittels weiterer spezifizierter Regeln zu erfüllen. Damit beruht *Colander* wie *TYPOL* auf den Prinzipien der logischen Programmierung.

Benutzungsschnittstelle: Die Benutzungsoberfläche von *Pan* ist anders als z.B. die des **Synthesizer Generator** nicht syntaxgesteuert sondern syntaxerkennend. Dies bedeutet, daß der Benutzer ein Programm nicht menügeführt, sondern wie bei konventionellen Editoren textuell eingibt. *Pan* analysiert auf Anforderung durch den Benutzer diese textuelle Repräsentation des Programmes anhand der konkreten und abstrakten Syntax. Der Anwender wird von *Pan* nie dazu gezwungen, ein syntaktisch

Eigenschaft	*Cenatur*	*Pan*	**Synthesizer Generator**
Anzahl der Spezifikations-sprachen	3	2	1
Erforderliche Spezifikationen	konkr. Syntax	konkr. Syntax	konkr., abstr. Syntax, Unparsing
Spezifikation der Semantik beruht auf	logische Programmierung	logische Programmierung	Attributierung
Analyse des Programmtextes	auf Anforderung	auf Anforderung	inkrementell
Verwendung inkrementeller Algorithmen	nein	ja	ja
Integration/ Anbindung von Werkzeugen	ja	ja	bedingt

Tabelle 2.1: *Centaur*, *Pan* und **Synthesizer Generator** im Vergleich

wie semantisch korrektes Programm zu schreiben. Abweichungen von der Syntax werden von *Pan* lediglich als solche visuell gekennzeichnet.

Integration von Werkzeugen: *Pan* ist in Common-Lisp implementiert. Der Lisp-Interpreter wie auch das gesamte Laufzeitsystem von *Pan* ist dem Entwickler der Software-Entwicklungsumgebung zugänglich. Damit hat dieser die Möglichkeit, sowohl auf die interne Darstellung als auch auf die verschiedensten Systemroutinen zuzugreifen, um weitere Werkzeuge zu implementieren. Auch eine Anbindung externer Werkzeuge sollte über eine mittels Lisp implementierte Schnittstelle problemlos möglich sein.

Tabelle 2.1 stellt zusammenfassend die wesentlichen Unterschiede zwischen den drei Systemen dar. Der Vergleich zeigt, daß nur *Pan* als konkurrierendes System zu betrachten ist. Jedoch erst der praktische Einsatz von *Pan* wird zeigen, ob dieses System dieselbe Leistungsfähigkeit und Stabilität wie der **Synthesizer Generator** aufweist.

2.4 ProSet

Es soll nun eine kurze Einführung in die Programmiersprache ProSet folgen. Die Sprache wird hier keinesfalls vollständig beschrieben oder definiert. Es wird hier nur ein allgemeiner Überblick über die Sprache und deren Möglichkeiten, sowie ein Überblick über die Besonderheiten von ProSet hinsichtlich dieser Arbeit gegeben werden. Eine detaillierte Beschreibung der Sprache ist unter [DFH+92] zu finden.

2.4.1 Allgemeine Merkmale von ProSet

Die Programmiersprache ProSet ist entwickelt worden, um eine schnelle Konstruktion und Modifikation von ausführbaren Prototypen zu ermöglichen. Zu diesem Zweck bietet ProSet leistungsfähige Datentypen, die auf der mathematischen Theorie endlicher Mengen beruhen. Auf diesen beiden Eigenschaften beruht der Name von ProSet : Prototyping With Sets. Die Sprache bietet jedoch nicht nur die Möglichkeit der Programmierung auf einer sehr hohen Abstraktionsebene, vielmehr kann mit ProSet auch auf einem niedrigen Niveau der Abstraktion gearbeitet werden, wie dies z.B. mit den Programmiersprachen Pascal, Ada oder C möglich ist. ProSet ist also nicht nur zwecks Prototyping einsetzbar, sondern kann als Breitbandsprache bezeichnet werden. Darüber hinaus werden von ProSet Mechanismen zur Modularisierung von Programmen sowie Persistenzmechanismen angeboten, was das Programmieren im Großen ermöglicht.

ProSet ist eine schwach typisierte Programmiersprache: Variablen müssen nicht explizit deklariert werden und können zur Laufzeit ihren Typ wechseln. Nicht initialisierte Variablen enthalten den undefinierten Wert `om`.

Datentypen

Es soll nun ein kurzer Überblick über die von ProSet angebotenen Datentypen gegeben werden.

Primitive Datentypen. ProSet bietet die einfachen Datentypen `integer`, `real`, `boolean`, `string` und `atom`. Lediglich die beiden letzten Datentypen bedürfen hier der Erläuterung: Der Typ `string` dient nicht nur der Darstellung von Zeichenketten beliebiger Länge sondern auch der einzelner Zeichen. Ein Datentyp `char` wird daher nicht benötigt. Werte des Datentyps `atom` werden weltweit nur einmal vergeben und sind daher zur eindeutigen Identifikation von Objekten über die Laufzeit eines Programmes hinaus geeignet.

ProSet stellt eine umfangreiche Anzahl von Operatoren und vordefinierten Funktionen für diese Datentypen bereit. Unäre Operatoren werden in ProSet in Präfix- und binäre Operatoren in Infixnotation verwendet. Viele Operatoren sind in ProSet überladen. Sie können daher auf verschiedene Datentypen angewendet werden.

Für die Typen `integer`, `real` und `boolean` stellt ProSet die üblichen Operatoren und Funktionen zur Verfügung. Auf den Typ `atom` dürfen die Vergleichsoperatoren `=` und `/=`, sowie der Operator `type` angewendet werden. Letzterer ist ebenfalls auf allen anderen Datentypen definiert und erlaubt nach Anwendung den Vergleich von Typen von Objekten. Für den Datentyp `string` sind über die allgemein üblichen Operatoren und Funktionen hinaus verschiedene, hier nicht weiter zu erläuternde Funktionen definiert, die eine komfortable Verarbeitung von Zeichenketten erlauben.

Zusammengesetzte Datentypen. ProSet bietet als zusammengesetzte Datentypen die Typen `set` und `tuple` an. Diese finden ihre Entsprechung in den mathematischen, endlichen Mengen bzw. Tupeln. Werte dieser beiden Datentypen können enumerativ, in Form eines Intervalls oder aber auch deskriptiv konstruiert werden. Es sollen an dieser Stelle nur einzelne Beispiele dargestellt werden, die die Mächtigkeit dieser Datentypen demonstrieren:

```
S1:={10 .. 100};
T1:=["a" .. "z"];
```

In diesem Beispiel wird `S1` die Menge aller ganzen Zahlen von `10` bis `100` und `T1` ein alle Kleinbuchstaben enthaltendes Tupel zugewiesen.

```
S2:={1,2,3,"x","y","z"};
T2:=[T1,["A" .. "Z"],5];
```

Hier wird `S2` die heterogene Menge mit den ganzen Zahlen `1`, `2` und `3` und den Kleinbuchstaben `x`, `y` und`z` zugewiesen. `T2` erhält ein Tupel, das aus dem Tupel `T1`, einem alle Großbuchstaben enthaltenden weiteren Tupel und der Zahl `5` besteht.

```
S3:={[x,y] : x in {2..10}, y in {2..10} | x<y};
T3:=[x:x in {0 .. 10000}|x mod 10=0];
```

`S3` wird in diesem Beispiel die Menge aller 2-Tupel zugewiesen, deren Elemente jeweils aus den Zahlen `2` bis `10` bestehen und deren erstes Element jeweils kleiner als das zweite ist. `T3` erhält schließlich ein Tupel bestehend aus allen durch `10` teilbaren Zahlen von `0` bis `10000`.

Datentypen höherer Ordnung. Die Datentypen `function`, `modtype` und `instance` werden in ProSet als Datentypen höherer Ordnung bezeichnet. Sie besitzen Bürgerrechte erster Klasse, d.h. Werte dieser Typen besitzen eine Identität und können zugewiesen werden. Die Verwendung der Begriffe Prozedur und Funktion mag in diesem Zusammenhang etwas verwirrend erscheinen. Im Gegensatz zu verschiedenen anderen Sprachen können Prozeduren durch die Verwendung der `return`-Anweisung einen Wert zurückgeben. Durch Anwendung des `closure`-Konstrukts auf eine Prozedur wird eine Funktion erzeugt. Die erzeugte Funktion unterscheidet sich dann von der ursprünglichen Prozedur dadurch, daß im Rumpf verwendete nichtlokale Bezeichner „eingefroren" werden, d.h ihre Werte bei späterem Aufruf der Funktion immer dieselben sein werden wie bei Anwendung des `closure`-Konstrukts. Mit dem Datentyp `module` ist es möglich, abstrakte Datentypmodule zu implementieren.

Kontrollstrukturen

In ProSet existieren die in algol-ähnlichen Sprachen üblichen Konstrukte zur Bildung von Programmschleifen. Diese sind jedoch an mengentheoretische Grundprinzipien angepaßt worden.

Neben diesen Schleifenkonstrukten bietet ProSet die ebenfalls üblichen bedingten und fallgesteuerten Anweisungen.

Aufbau eines ProSet-Programms

Ein ProSet-Programm beginnt nach einer einleitenden Kopfzeile mit einem Deklarationsteil für Variablen und Konstanten. Hier können Variablen und Konstanten initialisiert werden und ihre Sichtbarkeit kann modifiziert werden (später näher beschrieben). Schließlich können Variablen und Konstanten innerhalb dieses Deklarationsteils als persistent vereinbart werden.

Auf diesen Deklarationsteil folgt der Anweisungsteil. Innerhalb dieses Anweisungsteils kann schon auf Prozeduren, Handler und Module zugegriffen werden, die im nachfolgenden Deklarationsteil definiert werden. Der Aufbau von Prozeduren, Handlern und Modulen entspricht dabei genau dem des Hauptprogramms, d.h. in ihnen können Variablen und Konstanten und auch wiederum Prozeduren, Handler und Module definiert werden.

Prozeduren

Eine Prozedur wird in ProSet über ihren Namen aufgerufen. Dabei können ihr verschiedene Parameter übergeben werden. Für jeden zu übergebenden Parameter kann bei der Deklaration der Prozedur bestimmt werden, ob „call-by-value“, „call-by-result“ oder “call-by-value/result“ als Übergabemechanismus verwendet werden soll. In ProSet werden grundsätzlich alle Parameter via Copy-Semantik übergeben, d.h. es werden jeweils die Werte von Bezeichnern und nicht Referenzen auf entsprechende Speicherzellen übergeben.

Jede Prozedur liefert bei ihrem Aufruf einen Wert zurück. Dieser kann mittels der `return`-Anweisung innerhalb der Prozedur bestimmt werden. Wird dies unterlassen, gibt die Prozedur den undefinierten Wert `om` zurück.

In ProSet ist es wie in Lisp möglich, anonyme Prozeduren zu definieren. Ihre Definition entspricht im wesentlichen der einer benannten Prozedur, wobei der Name der Prozedur durch das Schlüsselwort `lambda` ersetzt wird.

Module

Um das Programmieren im Großen zu unterstützen, besteht in ProSet die Möglichkeit, generische Module zu definieren. Der Aufbau von Modulen ist in ProSet dem Aufbau von Prozeduren und Handlern sehr ähnlich. Statt einer Liste von Übergabeparametern wird hier jedoch eine Liste importierter und eine Liste exportierter Bezeichner angegeben. Aus Modulen können wiederum durch

Verwendung des `closure`-Konstrukts ProSet-Werte erster Klasse gewonnen werden, die u.a. persistent gemacht werden können. Dadurch wird es möglich, Bibliotheken von Modulen anzulegen.

Sichtbarkeitsregeln

Alle in einem Deklarationsteil für Variablen und Konstanten befindlichen Definitionen sind nur im darauffolgenden Anweisungsteil verfügbar, sofern ihr Sichtbarkeitsbereich nicht mittels des Schlüsselwortes `visible` bei ihrer Deklaration erweitert wurde. In diesem Fall kann auch innerhalb der diesem Anweisungsteil folgenden Definitionen von Prozeduren, Handlern und Modulen auf sie zugegriffen werden.

Nur die einem Anweisungsteil folgenden, auf oberer Ebene definierten Prozeduren, Handler und Module sind innerhalb dieses Anweisungsteils verfügbar. Auf Definitionen innerhalb dieser kann von hier aus nicht zugegriffen werden.

Für formale Parameter von Prozeduren gelten dieselben Sichtbarkeitsregeln wie für `visible` deklarierte Variablen oder Konstanten: Sie sind nur innerhalb des Anweisungsteils der entsprechenden Prozedur, sowie in allen untergeordneten Prozeduren, Handlern und Modulen sichtbar.

Weitere Sprachmerkmale

Persistenz. In ProSet ist es möglich, Daten über die Laufzeit eines Programmes hinaus zu erhalten. Solche Daten werden als persistent bezeichnet. Sie werden in einem sogenannten P-File abgelegt und stehen bei zukünftigen Programmläufen zur Verfügung. ProSet bietet darüber hinaus Transaktionen und unterstützt den Mehrbenutzerbetrieb.

Jedes Datenobjekt erster Klasse, also auch Funktionen, können persistent gemacht werden. Dies geschieht durch eine entsprechende, das Schlüsselwort `persistent` sowie eine das P-File identifizierende Zeichenkette enthaltende Deklaration der jeweiligen Variablen oder Konstanten.

Parallelität. ProSet unterstützt die Programmierung paralleler Applikationen. Funktionen können als eigene Prozesse parallel zu anderen abgearbeitet werden. Die Kommunikation und Synchronisation von Prozessen wird durch die Verwendung von Tupelräumen ermöglicht, für deren Verwaltung ProSet spezielle sprachliche Konstrukte bereitstellt.

Ausnahmebehandlung. Um die Zuverlässigkeit eines Programmes zu erhöhen, sowie um eine bessere Strukturierung desselben zu ermöglichen, bietet ProSet einen Mechanismus zur flexiblen Ausnahmebehandlung an. Hier besteht die Möglichkeit, verschiedene Programmroutinen zu spezifizieren, die beim Eintreten verschiedener Ausnahmesituationen abgearbeitet werden. Diese Routinen werden im folgenden als *Exception-Handler* oder als Handler bezeichnet werden, während die Ausnahmesituationen nur Ausnahmen genannt werden.

Handler entsprechen in ihrem Aufbau und ihrer Funktionsweise weitgehend den Prozeduren. Bei der Definition eines Handlers ist es zusätzlich möglich, eine Liste von Bezeichnern von Ausnahmen anzugeben, für deren Behandlung der Handler verwendet werden soll.

Eine Ausnahme wird entweder bei vordefinierten Ausnahmesituationen wie einer Division durch Null vom Laufzeitsystem oder aber explizit über spezielle Anweisungen ausgelöst. Anschließend wird das Programm dann durch den entsprechenden, der Ausnahme zugeordneten Handler fortgeführt. Nach der Behandlung der Ausnahme wird wiederum mittels verschiedener Anweisungen der weitere Kontrollfluß bestimmt.

2.4.2 Derzeitige Implementationsbeschränkungen

Die derzeitige Implementation von ProSet enthält noch verschiedene Beschränkungen, die nicht der Sprachbeschreibung entsprechen und später einmal entfallen werden. Verletzungen dieser Beschränkungen wird der Editor mit einer Warnung quittieren. Diese Warnung wird einen Hinweis darauf enthalten, daß es sich nicht um einen eigentlichen syntaktischen oder semantischen Fehler, sondern um die Verletzung einer Implementationsbeschränkung handelt.

Die Implementation des ProSet-Übersetzer enthält zur Zeit folgende Einschränkungen:

1. Bei Funktionen und Handlern sind nur `rd`-Parameter erlaubt.
2. Module dürfen nur Prozeduren oder wiederum Module exportieren.
3. Import- und Exportlisten von Modulen dürfen keine gemeinsamen Elemente enthalten. (Dies ist eine Folge aus der vorigen Einschränkung.)
4. L-Values, d.h. Ausdrücke, die auf der linken Seite von Zuweisungen stehen können, dürfen keine verschachtelten Selektoren enthalten.

Kapitel 3

Anforderungen an den ProSet-Editor

Um eine zielgerichtete Entwicklung des Editors zu ermöglichen, soll nun ein Anforderungskatalog ausgearbeitet werden, der zum einen allgemeine, sprach-un-ab-hän-gi-ge Anforderungen enthalten wird und zum anderen die ProSet-spezifische Problematik der Programmentwicklung berücksichtigen wird. Dieser Anforderungskatalog ist Ergebnis verschiedener Befragungen, Diskussionen, eines Fragebogens sowie eigener Erfahrung im Umgang mit Editoren sowie in der Programmierung im allgemeinen. Auch das Studium verschiedener in der Literatur beschriebener Systeme, sowie die praktische Erprobung verschiedener Editoren und eines frühen Prototyps des ProSet-Editors haben wesentlichen Einfluß gehabt.

Bei dem Einsatz des erwähnten Prototyps ist deutlich geworden, daß einige Abweichungen von der ursprünglichen Aufgabenstellung zu dieser Arbeit notwendig sind. Die entsprechend modifizierten Anforderungen werden in der folgenden Aufzählung einen entsprechenden Hinweis enthalten.

3.1 Allgemeine Anforderungen

Die allgemeinen Anforderungen umfassen folgende Punkte:

1. Der Editor muß einfach zu bedienen sein

   Dies ist eine Anforderung, die prinzipiell an jedes interaktive Software-Produkt gestellt wird. Sie ist im Zusammenhang mit dieser Arbeit jedoch von besonderer Bedeutung, da nur wenige Programmierer mit der Benutzung syntaxgesteuerter Editoren vertraut sind. Soll ein potentieller Benutzer von den Vorteilen eines syntaxgesteuerten Editors überzeugt werden, so darf der erste von ihm gewonnene Eindruck nicht der sein, daß die Arbeit mit einem solchen Werkzeug sich wesentlich komplizierter gestaltet als die mit einem konventionellen Editor. Diese Anforderung ist allgemeinster Art, sie soll jedoch in verschiedenen folgenden Anforderungen konkretisiert werden.

2. Die Bedienbarkeit des Editors muß einheitlich und konsequent sein.

   Der zu erstellende Editor muß (auch im Sinne der vorigen Anforderung) ein konsequentes, in sich homogenes Konzept der Bedienung bieten. Hierbei ist unter dem Begriff Bedienung nicht nur die Aktion des Benutzers, sondern auch die Reaktion des Editors zu verstehen. So sollten beispielsweise optionale Elemente immer gleichartig als solche (z.B. farblich) gekennzeichnet werden. Auch sollte immer derselbe Mechanismus angeboten werden, um optionale Elemente in den Programmtext aufzunehmen (z.B. Mausklick auf den Platzhalter oder Transformation der gesamten enthaltenden syntaktischen Struktur).

3. Der Editor muß eine übersichtliche Menüstruktur besitzen.

   In dieser Arbeit wird, wie bereits erwähnt, auch eine dauerhaft sichtbare Ansammlung von Knöpfen innerhalb eines abgetrennten Bereichs der Benutzungsoberfläche als Menü bezeichnet. Die Menge der Auswahlmöglichkeiten verfügbarer Menüs können als Knoten eines *Menübaums* betrachtet werden. Die Söhne eines jeden Knotens sind diejenigen Auswahlmöglichkeiten, die ein Folgemenü bietet, das bei Selektion des Auswahlpunktes, der mit dem betrachteten Knoten assoziiert ist, erscheint.

   Der Menübaum des Editors sollte einerseits flach gehalten werden, d.h. die Blätter (und damit die mit diesen verbundenen Aktionen) sollten über wenige Zwischenmenüs zu erreichen sein, um eine schnelle Arbeit zu ermöglichen.

   Andererseits sollten die Menüs nicht zu viele verschiedene Auswahlmöglichkeiten gleichzeitig anbieten, da dies die Übersichtlichkeit des Menüs stark vermindert und die Wahrscheinlichkeit der Fehlbedienung erheblich vergrößert.

4. Der Editor muß verschiedene Benutzergruppen unterstützen.

   Anfänger müssen bei der Erstellung syntaktisch wie semantisch korrekter Programme dadurch unterstützt werden, daß der Editor einen weitgehend menügeführten Aufbau von Programmen ermöglicht, so daß die Programmierung keine genaue Kenntnis der Syntax erfordert.

   Fortgeschrittene Programmierer sollen durch automatisierte, komplexe Transformationen sowie durch verschiedene Werkzeuge zur semantischen Analyse von Programmen unterstützt werden.

   Anders als in der Aufgabenstellung dieser Arbeit gefordert, sollten jedoch Anfänger und Fortgeschrittene zur Programmerstellung keine verschiedenen Sichten standardmäßig verwenden. Ursprünglich war für den Anfänger eine Sicht geplant, die alle optionalen, nicht-expandierten Inkremente permanent als solche darstellt. Dies führte jedoch zu einer sehr unübersichtlichen, weil mit optionalen Inkrementen überladenen, Darstellung des Programms. Statt dessen sollen nun optionale, nicht-expandierte Inkremente erst bei ihrer Selektion dargestellt werden. Für die Selektion von optionalen Elementen stellt der ProSet-Editor verschiedene Kommandos zur Verfügung. Daher stören diese optionalen Inkremente die Darstellung nicht, können jedoch problemlos auch vom Anfänger aufgesucht werden. Es können also sowohl Anfänger als auch Fortgeschrittene mit ein und derselben Sicht als Standardsicht arbeiten.

5. Die Bedienung des Editors soll an die Philosophie von ProSet angelehnt sein.

   Beispielsweise sollte der Editor den Prototyping-Charakter von ProSet ebenso berücksichtigen, wie den mathematischen Hintergrund verschiedener Datentypen.

3.2 Konkrete funktionale Anforderungen

Die konkreten funktionalen Anforderungen an den ProSet-Editor sind folgende:

1. Der ProSet-Editor soll eine menügeführte, graphische Benutzungsoberfläche besitzen.

   Dies ist z.Zt. Stand der Technik und nur solche Programme werden von vielen Benutzern (auch Programmierern) akzeptiert. Dies findet seine Begründung einerseits im zweifellos erhöhten Bedienungskomfort solcher Programme.

2. Menüs dürfen nur solche Auswahlmöglichkeiten enthalten, die einer sowohl syntaktisch als auch (soweit möglich) semantisch korrekten Vervollständigung eines Programmes dienen können.

   Es ist wünschenswert, den Benutzer des Editors davor zu bewahren, aus einer unübersichtlich großen Anzahl von Alternativen eine auszuwählen, um anschließend vom Editor über die Unzulässigkeit dieser Wahl informiert zu werden. Beispielsweise wird eine `quit`- oder `continue`-Anweisung nur innerhalb einer Schleife angeboten werden.

3. Verschiedene syntaktische Kategorien müssen auf verschiedene Art und Weise hervorgehoben werden.

   Es ist z.B. wünschenswert, daß neben Schlüsselwörtern auch Platzhalter, Kommentare usw. farblich oder durch besondere Schrifttypen hervorgehoben werden.

4. Die Struktur eines ProSet-Programmes soll durch sukzessives Ausblenden verschiedener syntaktischer Kategorien dargestellt werden können.

   Vorstellbar ist in diesem Zusammenhang z.B. eine Ansicht, die nur den blockstrukturierten Aufbau eines Programmes zeigt. Hier würden also neben dem Kopf des Programmes die Deklarationen von Prozeduren, Handlern und Modulen dargestellt. Sehr hilfreich ist auch ein Mechanismus, der es erlaubt, Anweisungsblöcke (z.B. innerhalb einer Programmschleife) zu verbergen.

5. Verschiedene Inkremente müssen auch textuell einzugeben sein.

   Beispielsweise sollen alle Literale über Tastatur einzugeben sein: Ein menügeführter Aufbau von z.B. Zeichenketten würde eine Abbildung der Tastatur auf ein Menü erfordern und wäre dem Bedienkomfort des Editors sehr abträglich. Daher werden, von obengenannten Ausnahmen abgesehen alle Konstanten nur textuell einzugeben sein.

   Auch zusammengesetzte Ausdrücke wie mathematische Terme oder Mengenausdrücke müssen auf konventionelle Art und Weise erstellt werden können. So ist die Summe `(3*4)+(5*6)` über Tastatur mit wenig Aufwand zu erstellen. Eine menügeführte Erstellung hingegen erweist sich hier als umständlich: Einerseits muß eine größere Anzahl von Auswahlen aus verschiedenen Menüs getroffen werden, um anschließend die Konstanten doch wieder textuell einzugeben. Andererseits müßte obiger Term nicht etwa von links nach recht aufgebaut werden, sondern gemäß seiner Bedeutung als Summe zweier Produkte jeweils zweier Konstanten.

   Trotzdem ist im Gegensatz zum Fall der Konstanten für Ausdrücke auch eine menügesteuerte Eingabe vorzusehen: Besonders für Anfänger wird es hilfreich sein, aus den im jeweiligen Kontext zur Verfügung stehenden Operatoren und Funktionen auswählen zu können. Dies trifft insbesondere auf ProSet-spezifische Elemente der Sprache wie Mengenausdrücke zu.

6. Der ProSet-Editor muß den gesamten Sprachumfang unterstützen.

   ProSet soll in dem Umfang unterstützt werden, der in [DFH+92] dargestellt ist. Auch die Funktionen der Standardbibliothek werden miteinbezogen werden.

7. Fertige ProSet-Programme im Textformat müssen eingelesen und geschrieben werden können.

   Das Einlesen ist einerseits erforderlich, um eine Akzeptanz des Editors nicht nur bei Neulingen zu finden. Andererseits sollten ProSet-Programme, die auf anderen Systemen entwickelt oder von etwaigen Hilfsprogrammen erzeugt wurden, mit dem ProSet-Editor zu pflegen sein. Dabei sollte der Benutzer auf etwaige syntaktische oder semantische Fehler im Quelltext hingewiesen werden. Hier ist besonderes Augenmerk auf solche semantischen Fehler zu richten, die bei einer Programmerstellung mittels des syntaxgesteuerten Editors nicht auftreten können. Beispielsweise ist das Auftreten einer `quit`-Anweisung außerhalb einer Programmschleife mittels einer Warnung anzumerken.

   Das Schreiben ist erforderlich, um mit dem **Synthesizer Generator** bearbeitete Programme mit anderen, nicht integrierten Werkzeugen weiterverarbeiten zu können.

8. Die semantische Korrektheit eines Programmes sollte vom Editor untersucht werden.

   Dies ist besonders bei einer schwach typisierten Sprache wie ProSet nur in sehr eingeschränktem Maße möglich. Generell soll aber ein Programm vom Editor genau dann als fehlerfrei dargestellt werden, wenn der ProSet-Übersetzer dieses Programm akzeptiert.

9. Der Editor sollte verschiedene Mechanismen anbieten, die eine Untersuchung verwendeter Bezeichner erlauben.

   Neben dem gezielten Auffinden der Deklaration eines gegebenen Bezeichners entsprechend der Syntax von ProSet wären z.B. auch Warnungen bei der Einführung eines neuen Bezeichners in einem Anweisungsteil sehr hilfreich zur Vermeidung semantischer Fehler. Da ProSet nicht die explizite Deklaration von Variablen erwartet, werden etwaige Tippfehler bei der Eingabe von Bezeichnern nicht vom Übersetzer erkannt. Sie führen auch beim anschließenden Programmlauf nicht zwangsläufig zu einem Abbruch. Daher sind solche Fehler häufig nur sehr schwierig zu lokalisieren.

10. Leistungsfähige Transformationen müssen zur Verfügung stehen.

    Nicht nur Transformationen zur komplexen Umstrukturierung von Programmen wie in Kapitel 2.2.2 motiviert sollen angeboten werden, sondern auch solche, die der Aufschlüsselung von Mengen- und Tupelausdrücken dienen. Damit sollen dem Benutzer einerseits eventuell recht kompliziert aufgebaute Mengen- und Tupelausdrücke transparent gemacht werden. Andererseits wird hierdurch eine schrittweise Ausführung entsprechender Programmteile bei der Fehlersuche mit einem Debugger ermöglicht.

11. Ein kontextsensitives Hilfesystem ist vorzusehen.

    Dabei sollten neben den verschiedenen syntaktischen Kategorien von ProSet auch die angebotenen Programmsichten, Transformationen wie auch die allgemeine Bedienung des Editors erläutert werden. Dies ist gerade deshalb wichtig, weil nur wenige Programmierer im Umgang mit syntaxgesteuerten Editoren Erfahrung haben.

12. Der Editor muß weitgehend konfigurierbar sein.

    Besonders von den Präferenzen des Benutzers abhängige Dinge wie verwendete Farben und Schriftsätze aber auch die bevorzugte, voreingestellte Programmsicht u.a. sollten einfach konfigurierbar sein.

13. Kommentare müssen (zumindest an ausgezeichneten Stellen) möglich sein.

    Anders als bei üblichen Übersetzerwerkzeugen dürfen Kommentare nicht vernachlässigt werden. Sie müssen zumindest in der internen Programmdarstellung repräsentiert werden.

Kapitel 4

Entwurfsentscheidungen

Es sollen nun verschiedene grundlegende Entscheidungen getroffen werden, die die Spezifikation des ProSet-Editors maßgeblich beeinflussen. Diese ergeben sich einerseits aus den obengenannten Anforderungen an den Editor, sind aber andererseits von dem verwendetem Entwicklungswerkzeug, dem **Synthesizer Generator** abhängig.

4.1 Textuelle vs. menügeführte Eingaben

Da der zu entwickelnde Editor auch dem ProSet-Neuling eine zügige Programmentwicklung ermöglichen soll, ist eine menügeführte Eingabe für alle syntaktischen Einheiten eines Programms vorzusehen. Die einzigen Ausnahmen stellen hier die Eingabe von benutzerdefinierten Bezeichnern und von Literalen einfacher Datentypen dar: Diese sind einerseits syntaktisch so einfach aufgebaut und würden einen Menübaum andererseits so stark aufblähen, daß hier nur eine textuelle Eingabe vorzusehen ist. Auch die menügeführte Eingabe von Ausdrücken gestaltet sich zumindest für den erfahrenen Programmierer oft umständlicher und aufwendiger als die textuelle. Daher sollte die Eingabe von Ausdrücken generell auch textuell möglich sein. Für alle anderen syntaktischen Einheiten wäre es möglich, eine menügeführte Eingabe zu erzwingen. Da dies jedoch den Benutzer des Editors eventuell einschränkt und die Möglichkeit der textuellen Eingabe keine Nachteile bringt, kann eine textuelle Eingabe überall geschehen.

4.2 Optionale Elemente vs. Transformationen

Für die Behandlung optionaler Elemente wie dem Schlüsselwort `constant` oder der Markierung von Schleifen gibt es grundsätzlich zwei grundverschiedene Möglichkeiten:

Entweder enthält die umgebende syntaktische Einheit in jedem Fall das optionale Element, und nur dieses wird eventuell durch einen Nullstring oder einen

Kommentarstring dargestellt, oder es gibt jeweils zwei, bis auf das Vorhandensein des optionalen Elementes äquivalente Einheiten. Beispielsweise sind für Schleifen diese Ansätze denkbar:

loop	$\rightarrow$ *optlabel statements*
optlabel	$\rightarrow$ *nulllabel* | *label*

bzw.

loop	$\rightarrow$ *labeled-loop* | *unlabeled-loop*
labeled-loop	$\rightarrow$ *label unlabeled-loop*

Die beiden Möglichkeiten unterscheiden sich wesentlich in der Art, in der sie sich dem Benutzer präsentieren: Bei der ersten Version muß ein optionales Element ersetzt werden, indem der zugehörige (eventuell auf dem Bildschirm nicht sichtbare) Platzhalter z.B. mit dem Befehl *Forward-with-optionals* selektiert und anschließend überschrieben bzw. ersetzt wird. Bei der anderen Möglichkeit muß die umgebende syntaktische Einheit selektiert werden und mit einer Transformation in eine, das optionale Element enthaltende Einheit, umgewandelt werden. Da letztere Möglichkeit zu einer Vergrößerung des jeweiligen aktuellen Menüs führt und außerdem eine Zweckentfremdung des Mechanismus der Transformation darstellt, wird bei der Spezifikation des ProSet-Editors konsequent der ersten Methode der Vorzug gewährt. Beiden Methoden gemeinsam ist die Möglichkeit, gemäß Unparsing-Regel das optionale Element z.B. in Kommentarzeichen als solches anzuzeigen oder zu verbergen. Da das Anzeigen aller optionalen Elemente die Übersichtlichkeit der Darstellung des Programmtextes vermindert, wird der Editor optionale Elemente nur dann darstellen, wenn sie selektiert oder expandiert sind.

4.3 Menüaufbau: tief vs. breit

Bei der menügeführten Eingabe eines Programmes beeinflußt der Aufbau des Menübaums wesentlich die effiziente Arbeit mit dem Editor. Einerseits sollten Menüs möglichst wenige Auswahlmöglichkeiten bieten, um erstens das Menü übersichtlich gestalten zu können und zweitens dem Benutzer die Auswahl leicht zu machen. Andererseits stellt das „Durchhangeln“ durch diverse Menüs bis zum Erreichen eines bestimmten Auswahlpunktes eine unbequeme wie zeitintensive Arbeit dar. Da jedoch in Kapitel 3 die Forderung aufgestellt wurde, daß Menüs immer nur solche Optionen anbieten sollten, die zu einer syntaktisch korrekten Vervollständigung des Programmes führen, ist die Anzahl der verfügbaren Auswahlpunkte meistens stark eingeschränkt, so daß oft alle Möglichkeiten in einem Menü darstellbar sind.

Die beiden umfangreichsten Menüs dienen dem Aufbau von Anweisungen bzw. von Ausdrücken. Daher werden diese Menüs einige Untermenüs enthalten. Die entsprechenden Menübäume sind in Abbildung 4.1 auszugsweise dargestellt.

Gerade Pfeile bedeuten in dieser Darstellung, daß das Nichtterminal auf der linken Seite des Pfeils durch die Anwendung einer Transformation direkt in das auf der rechten Seite stehende umgewandelt werden kann. Der geschlängelte Pfeil hingegen deutet an, daß eine Transformation das linksseitige Nichtterminalsymbol durch mehrere Nichtterminalsymbole ersetzt, wobei für eines der so entstandenen Nichtterminalsymbole erneut eine Transformation angeboten wird. Beispielsweise kann `<stmt>` durch Aufruf der Transformation `BinOp:=` in eine kombinierte Zuweisung umgewandelt werden. Diese enthält dann drei Nichtterminalsymbole `<lValue>`, `<binOp>` und `<expr>`. Das für einen binären Operator stehende Nichtterminalsymbol `<binOp>` kann anschließend mit weiteren Transformationen umgewandelt werden.

4.4 Programmsichten

Der ProSet-Editor wird verschiedene Sichten anbieten. Neben der Standardsicht, die allen Benutzern der Programmerstellung dient, ist eine Sicht vorgesehen, die nur eventuelle Fehlermeldungen darstellt. Damit hat der Anwender die Möglichkeit, mit einem Blick festzustellen, ob das eingegebene Programm vom Übersetzer akzeptiert werden wird. Erscheinen diverse Fehlermeldungen, so kann er durch deren Selektion nach anschließendem Wechsel in die Standardsicht (oder bei deren gleichzeitiger Darstellung in einem anderen Fenster) die fehlerhafte Programmstelle anzeigen lassen. Eine weitere Sicht, die einen Überblick über den Aufbau eines Programms bieten soll, wird neben der Deklaration des Hauptprogramms nur die Definitionen von Prozeduren, Handler und Modulen, sowie die Benutzung von Lambda-Konstrukten darstellen.

4.5 Komplexe Transformationen

Beim Editieren eines Programmes kann die Arbeit mit einem syntaxgestützten Editor sich gelegentlich als nachteilig erweisen, wenn grundlegende strukturelle Änderungen an dem Programm vorgenommen werden sollen und der Editor keine Mechanismen zur Unterstützung solcher Änderungen anbietet. So kann die Umwandlung einer `repeat-until`-Schleife in eine `while-do` Schleife sich recht aufwendig gestalten. Während eine textuelle Änderung hier nur wenige Operationen benötigt, sind folgende Schritte bei einem syntaxgestützten Vorgehen möglich: Nachdem der Schleifenrumpf in einen Puffer kopiert worden ist, muß die gesamte Schleife gelöscht werden. Anschließend muß eine `while-do`-Schleife an derselben Stelle eingefügt werden. Danach muß die ursprüngliche Abbruchbedingung in negierter Form als `while`-Bedingung an entsprechender Stelle eingegeben werden und schließlich das Schleifeninnere aus dem Puffer wieder in die Schleife kopiert werden. Um solche mühseligen Arbeitsschritte weitgehend zu vermeiden, wird der ProSet-Editor verschiedene Transformationsmechanismen zur Verfügung stellen. Folgende Transformationen sind wünschenswert:

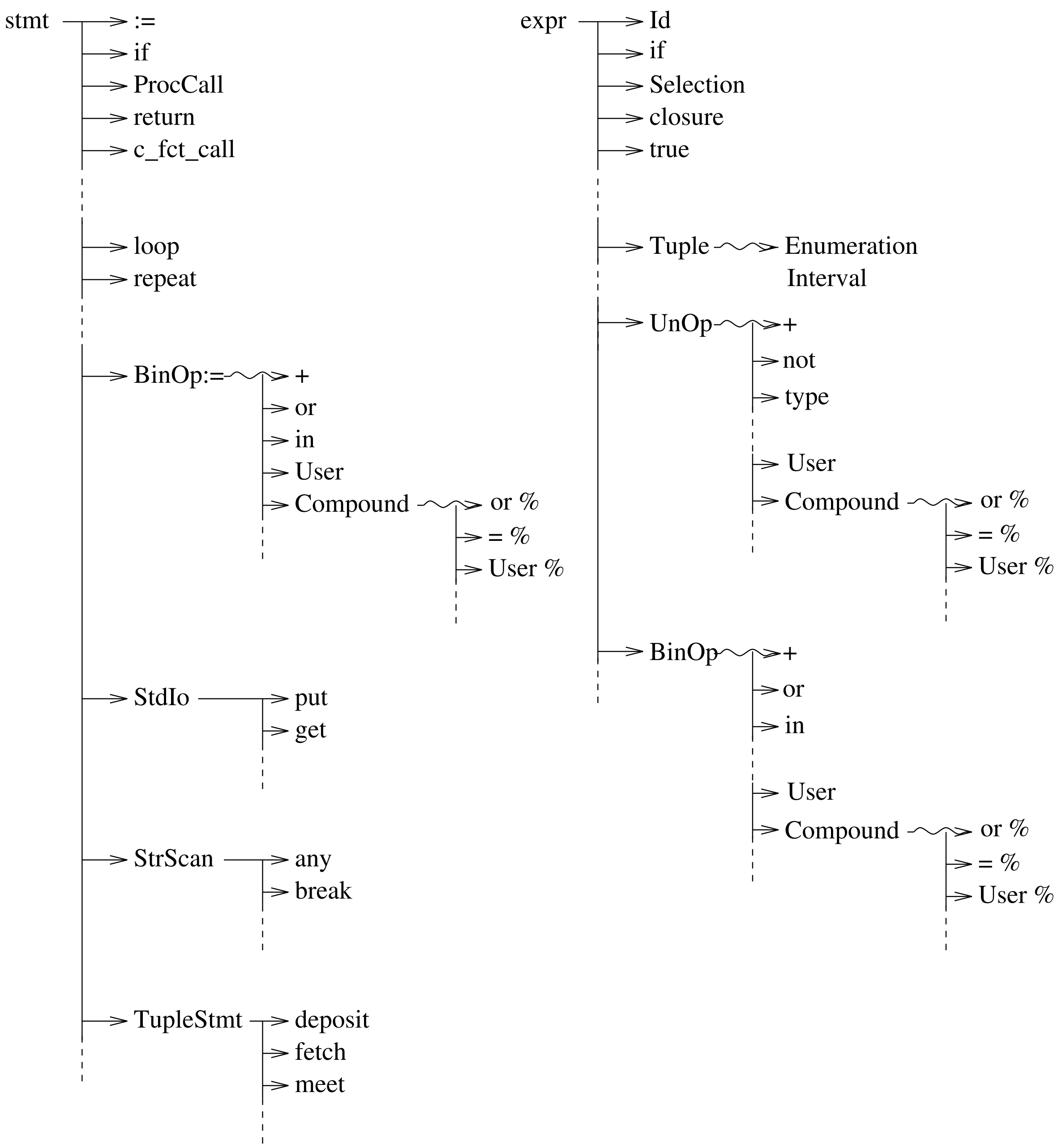

Abbildung 4.1: Menübaume für Anweisungen und Ausdrücke

repeat-until	$\rightarrow$ *while-do*
repeat-until	$\rightarrow$ *loop*
while-do	$\rightarrow$ *repeat-until*
while-do	$\rightarrow$ *loop*
for-do	$\rightarrow$ *repeat-until*
for-do	$\rightarrow$ *while-do*
for-do	$\rightarrow$ *loop*
$Proc$`(`$Expr_1$`,`$Expr_2$`)`	$\rightarrow Expr_1$ `!`$Proc$ $Expr_2$
$LValue$`:=`$LValue$ $BinOp$ $Expr$	$\rightarrow LValue$ $BinOp$`:=`$Expr$
$Expr$	$\rightarrow$ `!`$Expr$

`if` $Expr$ `then` $Stmts_1$ `else` $Stmts_2$

$\rightarrow$

`if !`$Expr$ `then` $Stmts_2$ `else` $Stmts_1$

Außerdem sollte die Umwandlung einer `case`-Anweisung in verschachtelte `if`-Anweisungen möglich sein. Da Tupel und Mengenausdrücke in ProSet sehr mächtig und nur schlecht überschaubar sind, müssen schließlich Transformationen bereitgestellt werden, die solche Ausdrücke aufschlüsseln.

4.6 Semantische Prüfungen

Wie bereits erläutert, ist die Möglichkeit der statischen Prüfung der Semantik eines ProSet-Programmes sehr eingeschränkt. Der ProSet-Editor wird jedoch all die semantischen Überprüfungen vornehmen, die auch der ProSet-Übersetzer anwendet. Hierdurch ist sichergestellt, daß der Editor dieselben Fehler erkennt wie der Übersetzer und damit ein vom Editor als fehlerfrei ausgewiesenes Programm sich auch immer fehlerfrei übersetzen läßt, sofern hierbei keine internen Übersetzer-Grenzen verletzt werden.

4.7 Werkzeuge zur Untersuchung der statischen Semantik

Der ProSet-Editor wird zwei recht leistungsfähige Werkzeuge anbieten, die dem Benutzer eine Untersuchung der statischen Semantik erlauben. Anhand dieser Werkzeuge soll der Benutzer die Antworten auf Fragen erhalten, die regelmäßig beim Programmieren insbesondere mit ProSet auftreten:

- Ist ein Bezeichner deklariert? / Wo ist ein Bezeichner deklariert?
- Wird eine deklarierte Variable benutzt?
- Wird eine lesend benutzte Variable zuvor initialisiert?
- Referenzieren zwei namentlich identische Bezeichner dasselbe Objekt?

- Was referenziert ein Bezeichner?
- Welche Bezeichner sind in einem Gültigkeitsbereich sichtbar?
- Welche Bezeichner sind nicht deklariert worden?

Diese Liste ließe sich fast beliebig verlängern. Um diese Fragen zu beantworten, wird der ProSet-Editor jederzeit Unterstützung anbieten, womit einer der größten Vorteile des syntaxgesteuerten Editors gegenüber dem konventionellen Editor zum tragen kommen wird. Nun sollen die beiden Werkzeuge kurz charakterisiert werden.

4.7.1 Suche nach Deklaration/Benutzung von Bezeichnern

Der ProSet-Editor wird Kommandos zur Verfügung stellen, die die Deklaration bzw. die nächste Benutzung eines selektierten Bezeichners, der eine Variable, Prozedur, ein Modul oder einen Handler referenziert, selektiert. Mithilfe dieses Mechanismus ist es z.B. möglich, alle Benutzungen eines derartigen Bezeichners nacheinander zu inspizieren oder zu überprüfen, ob ein gegebener Bezeichner z.B. zuvor deklariert wurde. Dieses Werkzeug stellt einen Hilfsmechanismus für das zweite Werkzeug dar.

4.7.2 Querverweisliste

Über eine zusätzliche Sicht, die im wesentlichen eine Erweiterung der schon beschriebenen Modulsicht darstellt, wird die Querverweisliste verwirklicht werden: Diese wird in die Darstellung des modularen Aufbaus des bearbeiteten Programms eine Liste aller im jeweiligen Gültigkeitsbereich sichbaren Bezeichner einfügen. Zu jedem Bezeichner wird außerdem kurz mitgeteilt werden, was er referenziert, und ob der Bezeichner innerhalb oder außerhalb des betrachteten Gültigkeitsbereichs, oder gar nicht deklariert wurde. Der Benutzer wird die Möglichkeit erhalten, einen in dieser Liste enthaltenen Bezeichner zu selektieren. In einer gleichzeitig sichtbaren Standardsicht kann er dann mittels der Suche nach Deklaration und Benutzung weitere Informationen über diesen Bezeichner sammeln.

Kapitel 5

Implementation des ProSet-Editors

Im folgenden soll nun auf die Implementation des ProSet-Editors eingegangen werden. Es wird jedoch nicht die vollständige Implementation dargelegt und erläutert werden, sondern nur deren wesentliche Aspekte grundlegend beschrieben. So werden zunächst die verschiedenen Konventionen, die vor der Implementierung getroffen wurden, erläutert. Anschließend wird die Modularisierung der Gesamtspezifikation dargestellt. Die hierauf folgenden Abschnitte orientieren sich nicht an dieser Modularisierung, sondern teilen die Spezifikation gemäß Kapitel 2.3.2 in abstrakte Syntax, konkrete Syntax, Attributierung, Unparsing und Transformationen auf.

5.1 Konventionen

Wie in [RT89b] vorgeschlagen, wurden folgende Konventionen bezüglich der Groß- und Kleinschreibung bei der Spezifikation des Editors eingehalten:

- Phyla der abstrakten Syntax werden **klein** geschrieben.
- Attributnamen werden **klein** geschrieben.
- Operatoren und Funktionen werden mit großen **AnfangsBuchstaben** geschrieben.
- Phyla von Lexemen werden **GROSS** geschrieben.
- Namen von Sichten werden **GROSS** geschrieben.

Phyla der konkreten Syntax werden nicht geschrieben, wie in [RT89b] vorgegeben, sondern wie sie aus der Eingabedatei "pst.con" für *ELI* übernommen wurden.

5.2 Modularisierung

Um die Übersichtlichkeit der Spezifikation zu verbessern, wurde diese in verschiedene Teilmodule aufgespalten. Da in SSL die Spezifikation der Attributierung und des Unparsing eng mit der der abstrakten Syntax verbunden sind, wurden diese zusammengehalten. Vielmehr fand hier eine Modularisierung auf sprachspezifischer Ebene statt. So entstanden für die Spezifikationen von abstrakter Syntax, konkreter Syntax, Unparsing und Attributierung vier Module in vier verschiedenen Dateien:

"ProSet.phm.ssl" In dieser Datei sind die Spezifikationen für den Programmrumpf, Prozedur-, Handler- und Moduldeklarationen sowie die Deklarationen von Variablen zu finden.

"ProSet.stmt.ssl" Die Spezifikation von Anweisungen und solcher syntaktischer Einheiten, die nur innerhalb von Anweisungen verwendet werden können, sind in diesem Modul zusammengefaßt.

"ProSet.expr.ssl" Ausdrücke sowie nur von diesen benötigte Einheiten sind hier abgelegt.

"ProSet.misc.ssl" Syntaktische Einheiten, die nicht eindeutig einem der oben genannten Module zuzuordnen sind, wurden in diesem Modul zusammengefaßt.

Weitere Module der Spezifikation des ProSet-Editors befinden sich in folgenden Dateien:

"ProSet.decl.ssl" In dieser Datei sind die Deklarationen von Attributen, Sichten sowie verschiedene Makros zu finden.

"ProSet.lex.ssl" Diese Datei enthält neben der Spezifikation der lexikalischen Syntax deren Zuordnung zur abstrakten Syntax, wie in Kapitel 2.3.2 beschrieben.

"ProSet.tra.ssl" Hier sind alle Transformationen spezifiziert.

"ProSet.fun.ssl" Verschiedenste SSL-Funktionen, wie sie z.B. zur Realisierung einiger komplexer Transformationen benötigt wurden, befinden sich in diesem Modul.

"ProSet.c" In dieser Datei befinden sich schließlich alle C-Funktionen. Diese dienen z.B. zur Realisierung zusätzlicher Editorkommandos.

In diversen weiteren Dateien befinden sich Hilfedateien für den ProSet-Editor. Diese beschreiben in englischer Sprache im Wesentlichen alle komplexen Transformationen sowie die verfügbaren Sichten.

5.3 Abstrakte Syntax

Die Spezifikation der abstrakten Syntax spiegelt sich bei mithilfe des **Synthesizer Generator** erstellten Editoren direkt in dem Menü wieder, das dem Benutzer die Anwendung verschiedener Produktionen auf das jeweils selektierte Inkrement anbietet. Jede angebotene Produktion muß gemäß Kapitel 3 einer syntaktisch korrekten Vervollständigung des bearbeiteten Programms dienen. Daher ist die abstrakte Syntax an die konkrete Syntax, wie sie in der Datei "pst.con" festgehalten ist, angelehnt. Wesentliche Abweichungen sind jedoch an solchen Produktionen festzustellen, die innerhalb der konkreten Syntax der Zuordnung von Präzedenzen oder der Vermeidung von Parsing-Konflikten dienen. Die abstrakte Syntax stellt daher im wesentlichen eine Vereinfachung der konkreten Syntax dar. An verschiedenen Stellen mußte die abstrakte Syntax jedoch wiederum erweitert werden. Zum Beispiel wurde die Produktion `stmt:LowStmt(comment lowstmt explAssos)` eingefügt, um eine einfache Bindung von Kommentaren und Handlerassoziationen an Anweisungen zu erreichen. Auch die abstrakte Syntax von Anweisungen und Ausdrucken wurde um verschiedene Produktionen erweitert, um eine Verlagerung verschiedener Auswahlmöglichkeiten in Untermenüs zu ermöglichen.

Die Namen für die Phyla der abstrakten Syntax erscheinen im Editor als Kontext. Daher wurden diese möglichst eindeutig und verständlich und oft wie die entsprechenden Nichtterminalsymbole der Sprachbeschreibung von ProSet [DFH+92], gewählt.

Die detaillierte Beschreibung der abstrakten Syntax ist in Anhang A zu finden.

5.4 Konkrete Syntax

Für die Spezifikation der konkreten Syntax wurde die Datei "pst.con" modifiziert. Nach der Anpassung an die Syntax des **Synthesizer Generator** mußten unter anderem alle linksrekursiven Produktionen in entsprechende rechtsrekursive Produktionen umgewandelt werden, da der **Synthesizer Generator** nur diese Form der Rekursion erlaubt. Des weiteren wurden die Attributgleichungen hinzugefügt, mittels derer der abstrakte Syntaxbaum aus dem Parsing-Baum, wie in Kapitel 2.3.2 beschrieben, entwickelt wird.

5.5 Attributierung

Die Attributierung des AST dient hauptsächlich drei verschiedenen Zwecken:

1. Überprüfung bzw. Einhaltung der Korrektheit der statischen Semantik

2. Grundlage der Semantikwerkzeuge

3. Korrekte, minimale Klammerung von Ausdrücken

```
program Test1;
begin
  P(x);
  procedure P();
  begin
    pass;
  end P;
end Test1;
```

Abbildung 5.1: Beispielprogramm 1 zur Attributierung

Die Überprüfung der Korrektheit der statischen Semantik dient der eventuellen Ausgabe geeigneter Fehlermeldungen nach dem Parsen von Programmen oder Programmteilen, während die Einhaltung der Korrektheit auf die Einschränkung des Angebots von Transformationen abzielt. Ein Beispiel für ein diesem Zweck dienliches Attribut ist das Attribut `flags`. Dieses Attribut enthält verschiedene Bits, die z.B. angeben, ob die Anweisung oder der Ausdruck, an den dieses Attribut gebunden ist, sich innerhalb einer Schleife befindet.

Ein Attribut, das den Semantikwerkzeugen dient ist z.B. `env`. Dieses Attribut, das an alle Phyla gebunden ist, die einen neuen Gültigkeitsbereich von Bezeichnern eröffnen, enthält eine Liste mit allen Bezeichnern, die innerhalb dieses Gültigkeitsbereichs sichtbar sind. Außerdem enthält diese Liste für jeden dieser Bezeichner u.a. die Information, was er referenziert.

Das Attribut `precedence` dient der korrekten und minimalen Klammerung von Ausdrücken. Jedem arithmetischen oder booleschen Ausdruck wird dazu über dieses Attribut die Präzedenz eines eventuellen übergeordneten Ausdrucks mitgeteilt, wodurch die Klammerung berechnet werden kann.

Schließlich finden weitere Attribute ihre Anwendung zum Speichern eventueller Fehlermeldungen und der Vermeidung von Mehrfachberechnungen von Attributen.

An zwei kleinen Programmbeispielen werden nun einzelne ausgewählte Attribute und deren Zusammenspiel betrachtet.

In Abbildung 5.1 ist ein fehlerhaftes Programm dargestellt. In diesem Programm wird eine parameterlose Prozedur `P` definiert. Im Anweisungsteil des Hauptprogramms wird diese Prozedur jedoch mit einem Parameter `x` aufgerufen. Dies wird der ProSet-Editor mit einer Fehlermeldung quittieren.

In Abbildung 5.2 sind ein Teil des korrespondierenden AST sowie einzelne Attribute dargestellt. Attribute sind in der Darstellung kursiv geschrieben, während die Knoten des AST normal erscheinen. Die durchgezogenen Pfeile kennzeichnen eine Vater/Sohn-Beziehung im AST. Zum Beispiel ist das Phylum `progBody` ein Sohn des Phylums `program`. Attributbeziehungen sind durch Pfeile mit gepunkteten Linien dargestellt. Das Attribut `error` ist also vom Attribut `env`

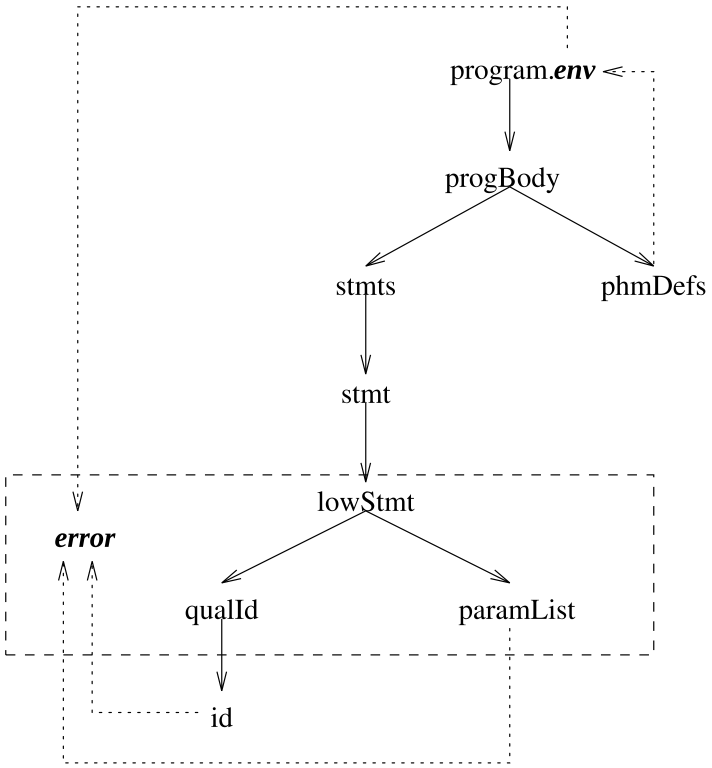

Abbildung 5.2: Attributierung zum Beispielprogramm 1

sowie von den Phyla `id` und `paramList` beeinflußt. In der Abbildung sind zwei verschiedene Attributtypen zu erkennen: Das Attribut `env` ist an ein Phylum (`program`) gebunden, während `error` an eine Produktion gebunden ist. Die in dieser Produktion auftretenden, dargestellten Phyla sind durch einen gestrichelten Kasten zusammen mit dem Attribut eingerahmt.

Der Aufbau des AST erklärt sich folgendermaßen: Der Rumpf (`progBody`) des Hauptprogramms (`program`) enthält zunächst eine Anweisungsliste (`stmts`), die aus nur einem Element (`stmt`) besteht. Diese einzelne Anweisung ist zunächst aus technischen Gründen auf ein Phylum `lowStmt` abgeleitet. Dieses Phylum enthält den Prozeduraufruf `P(x)`. Es wird also zum einen auf einen aus einem einfachen Bezeichner (`id`) bestehenden qualifizierten Bezeichner (`qualId`) abgeleitet. Zum anderen wird das Phylum `lowStmt` auf die Parameterliste (`paramList`) abgeleitet.

Das Attribut `error` wird mittels des Attributs `program.env` sowie mittels der Phyla `id` und `paramList` berechnet: Das Attribut `program.env` enthält alle im Hauptprogramm deklarierten Bezeichner für Variablen, Prozeduren, Handler und Module, im Beispiel also nur die Prozedur `P`. Neben den Bezeichnern enthält das Attribut noch jeweils verschiedene Informationen zu den einzelnen Bezeichnern: Für `P` werden neben der Information, daß es sich um eine Prozedur handelt, noch die Parameter mit abgelegt. Diese Informationen werden durch verschiedene SSL-Funktionen aus dem Phylum `phmDefs` extrahiert. Eine weitere SSL-Funktion sucht den Eintrag für `id` aus dem Attribut `env` heraus.

```
program Test2;
  visible V1;
begin
  pass;
  procedure P1();
    hidden V1;
  begin
    pass;
    procedure P2();
    begin
      V1:=3;V2:=2;
    end P2;
  end P1;
end Test2;
```

Abbildung 5.3: Beispielprogramm 2 zur Attributierung

Falls dieser gefunden wird, überprüft dieselbe SSL-Funktion, ob die im Phylum `paramList` enthaltenen Parameter korrekt sind. Ist dies der Fall, so gibt die Funktion einen Leerstring zurück, ansonsten die Fehlermeldungen `"(*<== wrong number of parameters*)"` oder `"(*<== illegal parameter*)"`. Das Ergebnis der Funktion wird schließlich dem Attribut `error` zugewiesen, und dieses durch das Unparsing hinter der textuellen Darstellung der Parameter ausgegeben.

Mittels eines zweiten kleinen Programmbeispiels kann die Funktionsweise der Querverweisliste erläutert werden. Dieses Beispiel ist in Abbildung 5.3 zu sehen.

Das Hauptprogramm enthält in diesem Beispiel eine als `visible` deklarierte Variable `V1` sowie eine Prozedur `P1`. Die Prozedur `P1` enthält wiederum eine `hidden`-Variable `V1` sowie eine Prozedur `P2`. Im Anweisungsteil dieser Prozedur werden schließlich zwei Variablen `V1` und `V1` verwendet. Gemäß der Sichtbarkeitsregeln von ProSet sind im Anweisungsteil des Hauptprogrammes nur die `visible`-Variable `V1` und die Prozedur `P1` sichtbar. Im Anweisungsteil von `P1` ist ebenfalls eine Variable `V1` sichtbar. Hier handelt es sich jedoch um die in `P1` als `hidden` deklarierte Variable. Außerdem ist in `P1` noch die Prozedur `P2` sichtbar. Bei der in `P2` verwendeten Variable `V1` handelt es sich jedoch wiederum um die `visible`-Deklaration aus dem Hauptprogramm, da die in `P1` deklarierte Variable `V1` hier nicht sichtbar ist. Die Variable `V2` schließlich wurde nicht explizit deklariert: Hier wird die Querverweisliste eine besondere Warnung aufweisen.

Ein Auszug aus dem zum korrespondierenden AST ist in Abbildung 5.4 dargestellt.

Die Semantik der verwendeten Darstellung entspricht weitgehend der im vorigen Beispiel verwendeten. Zusätzlich enthält diese Abbildung jedoch Pfeile mit gestrichelten Linien. Hierdurch wird eine indirekte Nachkommenschaft von Phyla innerhalb des AST dargestellt, da verschiedene Phyla aus Gründen der

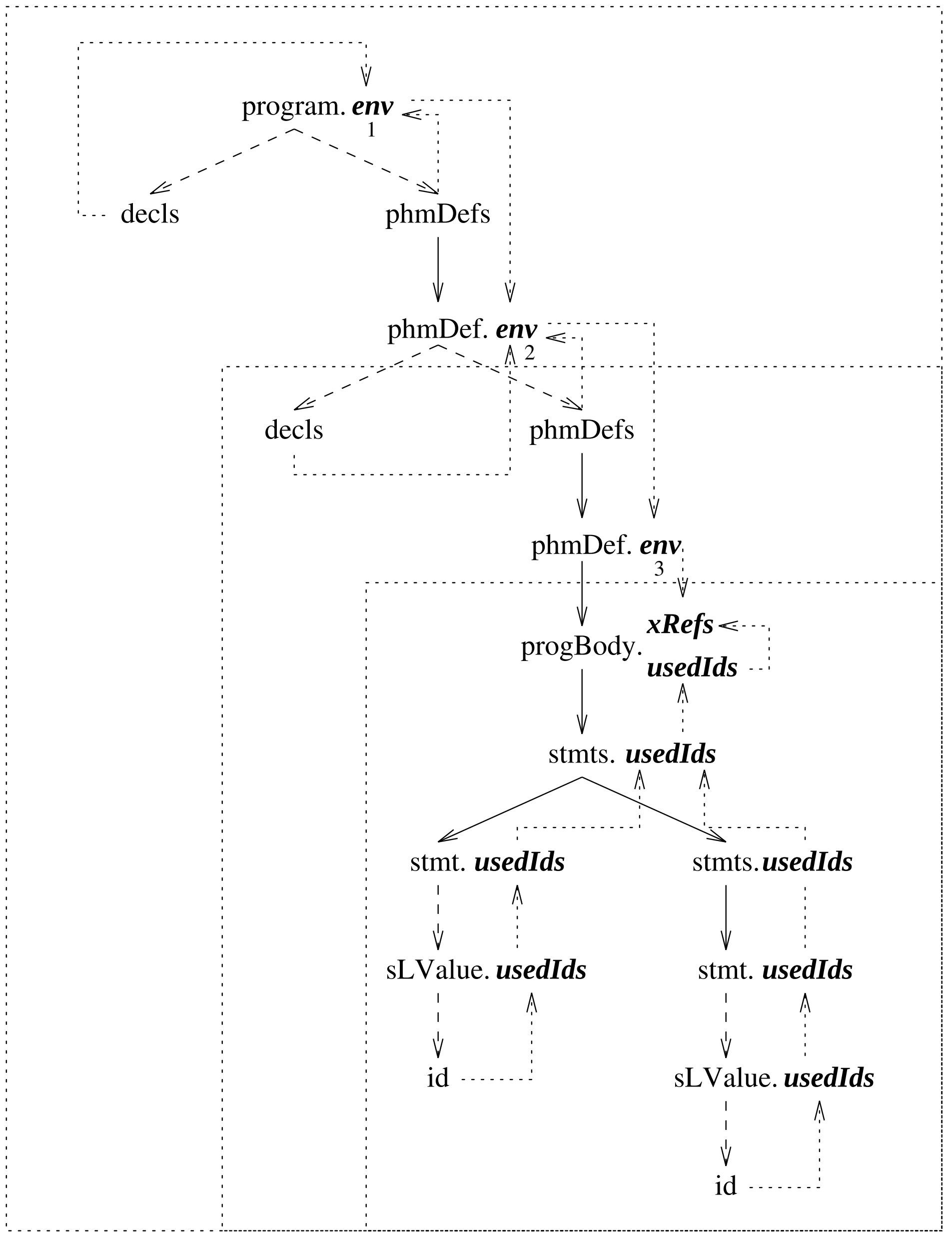

Abbildung 5.4: Attributierung zum Beispielprogramm 2

Übersichtlichkeit nicht dargestellt sind. Außerdem sind die in der Darstellung enthaltenen Attribute `env` durchnumeriert. Auch in dieser Abbildung wurden nur die zur Erklärung der Attributierung wichtigen Teile des AST dargestellt.

Die gepunkteten Kästen umfassen jeweils die verschiedenen Ebenen des Programmbeispiels: Der äußere Kasten umschließt das Hauptprogramm: Dieses enthält neben der in `decls` enthaltenen Deklaration von `V1` die Prozedur `P1` (`phmDef`). Die durch den mittleren Kasten eingeschlossene Prozedur `P1` enthält ebenfalls eine Variablendeklaration und eine Prozedurdefinition. Die Prozedur `P2` wird durch den inneren Kasten umschlossen: Sie enthält in ihrem Anweisungsteil (`progBody`) zwei Anweisungen: Beide enthalten als Blätter des AST die Variablenbezeichner (jeweils `id`).

Die Funktionsweise der Querverweisliste kann nun am Beispiel des Attributs `progBody.xRef` erläutert werden: Dieses Attribut enthält alle zum Aufbau der Querverweisliste der Prozedur `P2` notwendigen Informationen.

Wie in Abbildung 5.4 sichtbar, wird dieses Attribut u.a. mithilfe des ebenfalls an das Phylum `progBody` gebundenen Attributs `usedIds` ermittelt. Dieses Attribut `usedIds` enthält lediglich eine Liste aller im Anweisungsteil auftauchender Bezeichner, die im selben Namensraum liegen wie Variablenbezeichner. Beispielsweise sind Bezeichner für Sprungmarken hier nicht enthalten. Das Attribut `usedIds` ist an viele weitere Phyla des AST gebunden. Wie aus der Zeichnung ersichtlich, werden die als Blätter des AST auftauchenden Bezeichner jeweils in eine Liste aufgenommen und im AST nach oben durchgereicht. Hat ein Knoten des AST mehrere Söhne, die das Attribut `usedIds` besitzen, so wird das Attribut dieses Knotens durch Aneinanderreihung und Elimination doppelter Elemente der einzelnen Listen der Söhne erzeugt. Ein Beispiel für ein solches Phylum ist das im inneren Kasten oben stehende Phylum `stmts`. Dieses enthält im Beispiel eine Liste mit den Bezeichnern `V1` und `V2`.

In die Berechnung des Attributs `progBody.xRefs` geht ebenfalls das Attribut phmDef.env (Nr.3) ein. Dieses ist wiederum indirekt vom Attribut `program.env` (Nr.1) abhängig. Das Attribut `program.env` (Nr.1) enthält einerseits die Information, daß `V1` eine als `visible` deklarierte Variable ist. Dies wird durch verschiedene SSL-Funktionen aus dem Phylum `Decls` ermittelt. Andererseits enthalten ist der Bezeichner `P1`, sowie die Information, daß es sich hier um eine Prozedur handelt.
`program.env` (Nr.1) enthält also:

> `V1`: `visible`-Variable,
> `P1`: Prozedur

Über eine SSL-Funktion werden alle als `hidden` deklarierten Variablen aus `program.env` (Nr.1) herausgefiltert und zur Berechnung von `program.env` (Nr.2) verwendet. Da `V1` im Hauptprogramm als `visible` deklariert ist, bleibt `V1` jedoch erhalten. In `P2` sind eine weitere Variable `V1` sowie die Prozedur `P2` definiert. Diese werden an die gefilterte, von `program.env` (Nr.1) übernommene Liste angehängt. Wäre die in `P2` deklarierte Variable `V1` keine `hidden`-Variable, so würde die im Hauptprogramm deklarierte Variable `V1` aus der Liste entfernt worden, da diese in keinem untergeordneten Block mehr sichtbar würde. Da `V1` in `P2` jedoch als `hidden` deklariert wurde, wird `V1` aus `P1` in untergeordneten Blöcken wieder sichtbar sein. `program.env` (Nr.2) enthält also:

> `V1`: `visible`-Variable,
> `V1`: `hidden`-Variable,
> `P1`: Prozedur,
> `P2`: Prozedur

`program.env` (Nr.3) geht aus `program.env` (Nr.2) hervor, indem aus letzterem alle `hidden`-Variablen entfernt wurden. `program.env` (Nr.3) enthält also:

> `V1`: `visible`-Variable,
> `P1`: Prozedur,
> `P2`: Prozedur

Eine weitere SSl-Funktion berechnet schließlich aus den Attributen `program.env` (Nr.3) und `progBody.usedIds` die Querverweisliste: `program.env` (Nr.3) enthält alle in `P2` sichtbaren, explizit deklarierten Bezeichner, während alle in `progBody.usedIds`, jedoch nicht in `program.env` (Nr.3) enthaltenen Bezeichner zwar in `P2` verwendet, jedoch nicht explizit deklariert sind.
`progBody.xRefs` enthält somit folgende Bezeichner:

> `V1`: `visible`-Variable,
> `V2`: nicht explizit deklariert,
> `P1`: Prozedur,
> `P2`: Prozedur

Kapitel 6

Spezifische Elemente des ProSet-Editors

An dieser Stelle wird nun eine Erläuterung derjenigen Elemente des ProSet-Editors erfolgen, die nicht zwangsläufig durch die Benutzung des **Synthesizer Generator** als generierendes System vorgegeben sind. Wie bereits mehrfach erwähnt, sind dies im wesentlichen die angebotenen Sichten, die einfachen wie die komplexen Transformationen, die verschiedenen Werkzeuge, die den Programmierer bei der semantischen Analyse unterstützen werden sowie die Schnittstellen, die eine Ein- oder Anbindung weiterer Werkzeuge an den Editor ermöglichen sollen.

6.1 Programmsichten

Der ProSet-Editor bietet insgesamt vier verschiedene Sichten. Diese Sichten unterscheiden sich jeweils bezüglich verschieder Kriterien:

1. Vollständigkeit der Darstellung hinsichtlich des zugrundeliegenden AST
2. Art der Darstellung
3. Granularität der Selektion
4. Möglichkeit der textuellen Eingabe

Die vier angebotenen Sichten sollen nun jeweils nach einer allgemeinen Erläuterung hinsichtlich dieser vier Kriterien charakterisiert werden.

6.1.1 BASEVIEW

Diese Sicht ist diejenige, die standardmäßig vom ProSet-Editor zur Darstellung eines Programms benutzt wird. Sie sollte vom Benutzer zur Erstellung bzw. Modifikation eines Programms verwendet werden.

In dieser Sicht wird der gesamte, das Programm repräsentierende AST dargestellt.

Alle nicht-expandierten, jedoch für eine syntaktische Korrektheit notwendigen Inkremente werden durch Nennung der entsprechenden Nichtterminalsymbole in spitzen Klammern dargestellt. Optionale Elemente werden nur bei ihrer Selektion durch den Benutzer sichtbar. Diese werden durch eckige statt durch spitze Klammern dargestellt. Fehlermeldungen und Warnungen werden in Form von ProSet-Kommentaren innerhalb des Programmtextes angezeigt. Ein syntaktisch und semantisch korrektes Programm, das mittels dieser Sicht an den ProSet-Übersetzer übergeben wird, wird von diesem fehlerfrei übersetzt werden. Andererseits wird jedes Programm, welches vom Übersetzer nicht akzeptiert wird auch vom ProSet-Editor innerhalb dieser Sicht mit entsprechenden Fehlermeldungen oder der Darstellung nicht-expandierter, notwendiger Inkremente versehen werden.

Die Sicht bietet eine feingranulare Selektion. Das heißt der Benutzer kann fast jede syntaktische Struktur einzeln selektieren, um entweder eine Transformation hierauf anzuwenden oder um eine textuelle Eingabe zu machen.

Die textuelle Eingabe ist für die meisten Inkremente vorgesehen. Lediglich die Kopf- und die Fußzeilen von Hauptprogramm, Prozeduren, Handlern, Modulen sowie den verschiedenen Schleifen können nicht textuell eingegeben werden.

In Abbildung 6.1 ist die Sicht **BASEVIEW** für ein kurzes Beispielprogramm dargestellt. In den folgenden Abschnitten wird die jeweils beschriebene Sicht für dasselbe Programm dargestellt werden. Da die Sicht **XREFVIEW** unter der Version 5.0 des **Synthesizer Generator** fertiggestellt wurde, zeigen alle Darstellungen den ProSet-Editor, der mit der Version 5.0 generiert wurde. Die textuelle Darstellung des Programms ist jedoch in allen Fällen mit der eines unter V4.2 erstellten Editors identisch.

6.1.2 MODULEVIEW

Diese Ansicht dient der Darstellung einer Übersicht über die Struktur eines bearbeiteten Programms.

Es werden bei Benutzung dieser Sicht nur die Kopf- und Fußzeilen von Hauptprogramm, Prozeduren, Handlern und Modulen sowie die von Lambda-Definitionen angezeigt. Die Darstellung letzterer mag auf den ersten Blick verwundern, ist jedoch aufgrund der semantischen Ähnlichkeit zur Definition von Prozeduren gerechtfertigt.

Die sichtbaren Elemente werden ebenso wie in der **BASEVIEW** dargestellt. Die einzelnen Deklarationen werden jedoch ihrer Verschachtelung entsprechend eingerückt.

Jede Deklaration kann einzeln, jedoch nur vollständig selektiert werden. Dadurch wird es möglich, in einem gleichzeitig geöffneten Fenster, das dasselbe Programm in der **BASEVIEW** anzeigt, die entsprechende Programmstelle anzuzeigen.

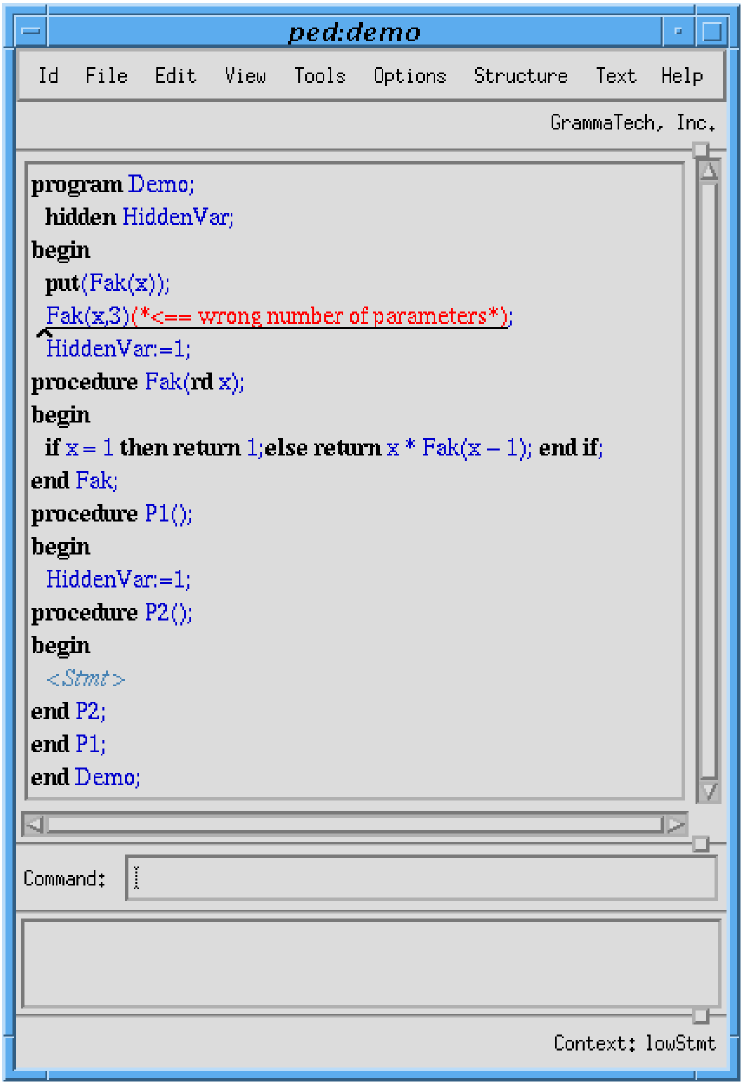

Abbildung 6.1: Die Sicht **BASEVIEW**

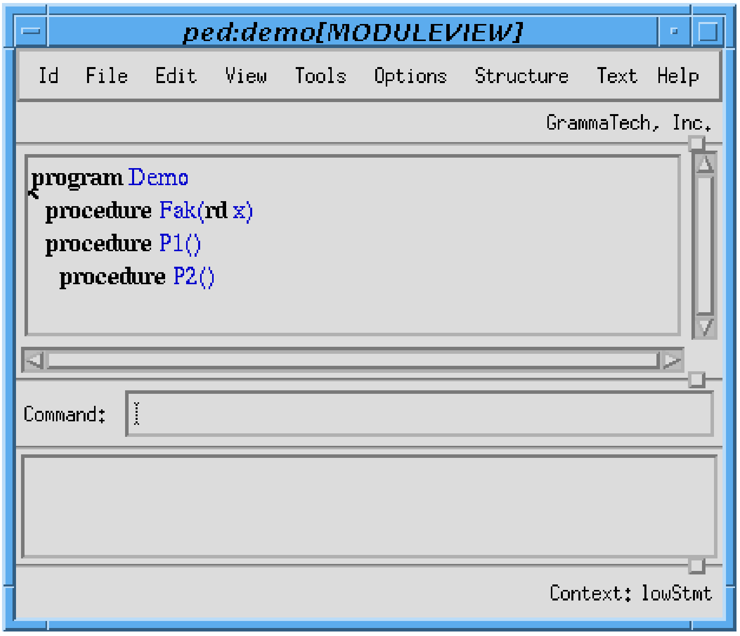

Abbildung 6.2: Die Sicht **MODULEVIEW**

Da diese Sicht nicht der Eingabe von Programmen dient, ist für kein Inkrement eine textuelle Eingabe erlaubt.

In Abbildung 6.2 ist die Darstellung des schon gezeigten Beispielprogramms in der **MODULVIEW** zu sehen.

6.1.3 XREFVIEW

Diese Sicht realisiert in Zusammenarbeit mit dem Werkzeug zum Auffinden von Deklarationen und Benutzungen von Bezeichnern einer Querverweisliste. Sie stellt eine Erweiterung der zuvor charakterisierten **MODULEVIEW** um die Darstellung einer Liste sichtbarer Bezeichner für jeden Gültigkeitsbereich dar. Für jeden explizit deklarierten Bezeichner wird außerdem mitgeteilt, ob er Prozedur, Handler, Modul, Variable oder Konstante referenziert. Bei Variablen oder Konstanten referenzierenden Bezeichnern wird zusätzlich die Art der Deklaration (Sichtbarkeit, Persistenz) wiedergegeben. Jeder Bezeicher, der außerhalb des betrachteten Gültigkeitsbereichs deklariert wurde, wird mit dem Wort `"inherited"` markiert. Nicht explizit deklarierte Bezeichner werden in den Gültigkeitsbereichen ihrer Anwendung mit dem Word `"implicit"` gekennzeichnet.

In dieser Sicht kann jeder Bezeichner einzeln selektiert werden. Daher ist die Anwendung der Werkzeuge zum Auffinden von Deklarationen und Benutzungen möglich.

Weitere Informationen bezüglich dieser Sicht sind im Kapitel 6.3 zu finden.

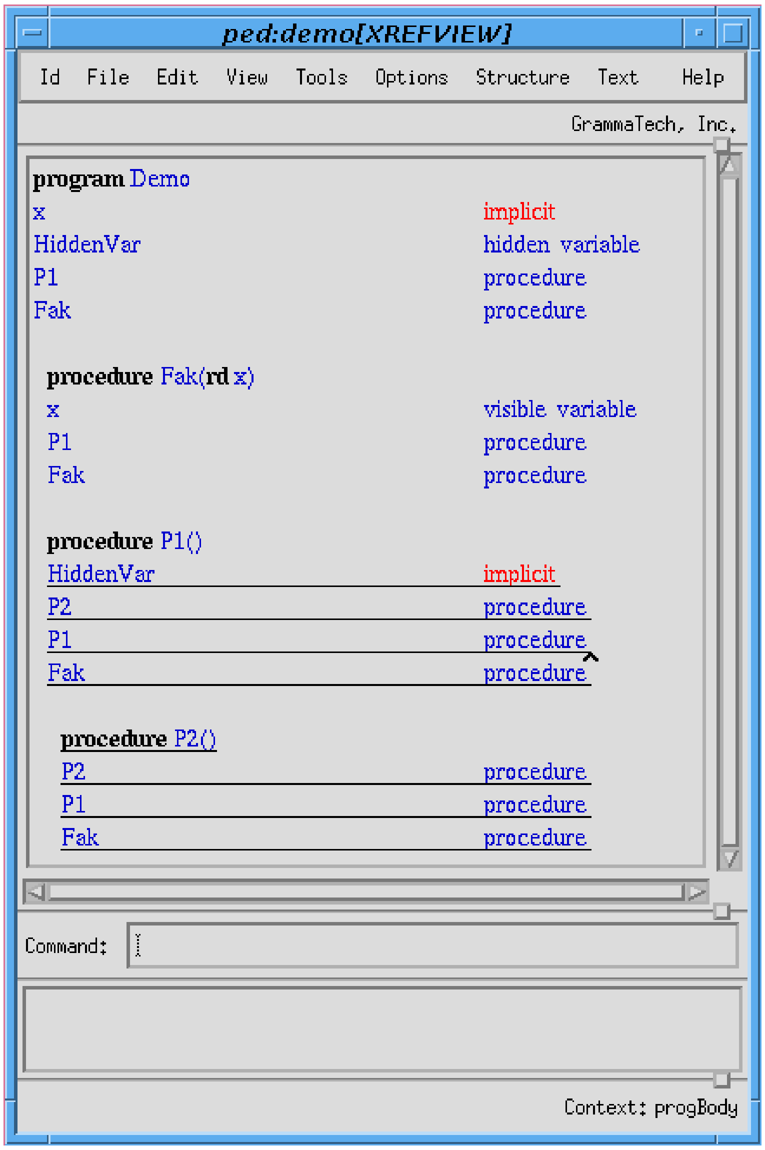

Abbildung 6.3: Die Sicht **XREFVIEW**

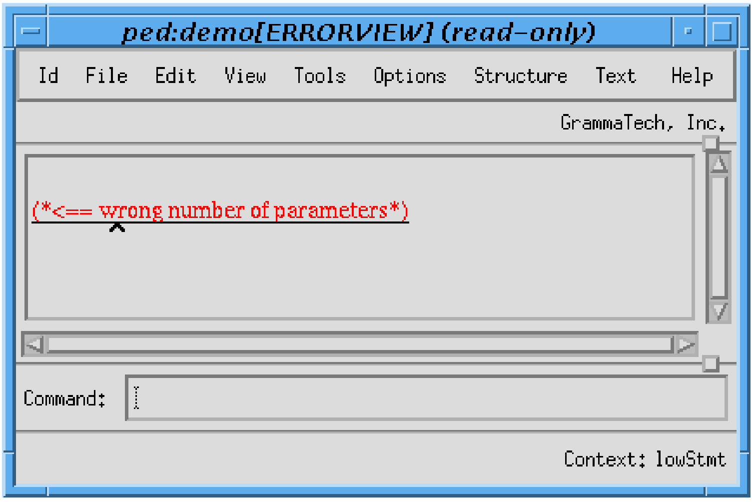

Abbildung 6.4: Die Sicht **ERRORVIEW**

In Abbildung 6.2 ist die Darstellung in der **XREFVIEW** zu sehen.

6.1.4 ERRORVIEW

Diese letzte Sicht dient der ausschließlichen Anzeige von Fehlermeldungen.

Es werden hier innerhalb des gesamten AST alle Meldungen zeilenweise angezeigt, wobei jede Meldung einzeln selektiert werden kann, um in der entsprechenden **BASEVIEW** die fehlerhafte Stelle des Programms aufzusuchen.

Eingaben oder Modifikationen sind in dieser Sicht nicht möglich. Daher enthält ein Fenster, wie in Abbildung 6.4 zu sehen, keinen Bereich, in dem eventuelle Knöpfe für Transformationen angeordnet sind.

6.2 Komplexe Transformationen

Nun sollen die verschiedenen komplexen Transformationen, die der ProSet-Editor zur Verfügung stellt, erläutert werden. Zu jeder Transformation wird zunächst der Name genannt. Namen von Transformationen zur Umwandlung von Kontrollstrukturen beginnen mit einem `"-"` oder einem `"="`. Letzteres deutet an, daß die Transformation die Semantik des Programms verändern kann. Für jede Transformation wird unter „Selektion“ das Phylum genannt, welches selektiert sein muß, um die Transformation zu aktivieren. Oft muß dieses Phylum weitere Bedingungen erfüllen. Diese werden jeweils unter „Bedingung“ aufgeführt. Neben verschiedenen Bemerkungen sind für einzelne Transformationen schließlich Beispiele angegeben.

- Transformationen zur Umwandlung von Programmschleifen
  - Umwandlung einer `repeat`-Schleife in eine `while`-Schleife

    Name: `-while`

    Selektion: `loopStmt`

    Bedingung: Das selektierte `loopStmt` muß eine `repeat`-Anweisung sein.

    Bemerkung: Die Abbruchbedingung wird negiert. Daher ist der einzige resultierende semantische Unterschied, daß die entstehende `while`-Schleife nicht unbedingt durchlaufen wird, während eine `repeat`-Schleife mindestens einmal bearbeitet wird.
  - Umwandlung einer `while`-Schleife in eine `repeat`-Schleife

    Name: `-repeat`

    Selektion: `loopStmt`

    Bedingung: Das selektierte `loopStmt` muß eine `while`-Anweisung sein.

    Bemerkung: Die Abbruchbedingung wird negiert. Daher ist der einzige resultierende semantische Unterschied, daß die entstehende `repeat`-Schleife mindestens einmal bearbeitet wird, während eine `repeat`-Schleife nicht unbedingt durchlaufen wird.
  - Umwandlung einer `repeat`-Schleife in eine `loop`-Schleife

    Name: `=loop`

    Selektion: `loopStmt`

    Bedingung: Das selektierte `loopStmt` muß eine `repeat`-Anweisung sein.

    Bemerkung: Eine `if`-Anweisung wird als letzte Anweisung in den Schleifenrumpf eingefügt. Diese enthält das Abbruchkriterium und eine `quit`-Anweisung.
    Diese Transformation verändert nicht die Semantik.
  - Umwandlung einer `while`-Schleife in eine `loop`-Schleife

    Name: `=loop`

    Selektion: `loopStmt`

    Bedingung: Das selektierte `loopStmt` muß eine `while`-Anweisung sein.

    Bemerkung: Eine `if`-Anweisung wird als erste Anweisung in den Schleifenrumpf eingefügt. Diese enthält das negierte Abbruchkriterium und eine `quit`-Anweisung.
    Diese Transformation verändert nicht die Semantik.
  - Umwandlung einer `loop`-Schleife in eine `repeat`-Schleife

    Name: `=repeat`

    Selektion: `loopStmt`

    Bedingung: Das selektierte `loopStmt` muß eine `loop`-Anweisung sein. Die letzte Anweisung innerhalb des Schleifenrumpfs muß eine `if`-Anweisung ohne `elseif`- oder `else`-Teil sein.

Bemerkung: Die obengenannte `if`-Anweisung wird entfernt. Der in ihr enthaltene Ausdruck wird Abbruchkriterium.
Diese Transformation verändert nicht die Semantik.

– Umwandlung einer `loop`-Schleife in eine `while`-Schleife

  Name: `=while`

  Selektion: `loopStmt`

  Bedingung: Das selektierte `loopStmt` muß eine `loop`-Anweisung sein. Die erste Anweisung innerhalb des Schleifenrumpfs muß eine `if`-Anweisung ohne `elseif`- oder `else`-Teil sein.

  Bemerkung: Obengenannte `if`-Anweisung wird entfernt. Der in ihr enthaltene Ausdruck wird negiertes Abbruchkriterium.
  Diese Transformation verändert nicht die Semantik.

- Transformationen für `if`- und `case`-Anweisungen/Ausdrücke

  – Vertauschung von `then`- und `else`-Teil einer `if`-Anweisung oder eines `if`-Ausdrucks

    Name: `then=else`

    Selektion: `stmt` bzw. `priSel`

    Bemerkung: Die Bedingung wird negiert und vereinfacht. Falls kein `else`-Zweig vorhanden ist, enthält der entstehende `then`-Zweig die `pass`-Anweisung bzw. `om`. Ein eventueller `elseif`-Zweig wird verarbeitet, wie im Beispiel gezeigt.

    Beispiele:

```
if e1 then s1();
elseif e2 then s2();
else s3();
end if;
```

→

```
if not e1 then
  if e2 then s2();
  else s3();
  end if;
else s1();
end if;
```

`if e1 then e2 end if;` → `if not e1 then om else e2;`

  – Umwandlung einer `case`-Anweisung bzw. eines `case`-Ausdrucks in eine `if`-Anweisung bzw. einen `if`-Ausdruck

    Name: `=if`

    Selektion: `stmt` bzw. `priSel`

    Bedingung: Eine `case`-Anweisung bzw. ein `case`-Ausdruck muß selektiert sein.

Bemerkung: Funktion wie im Beispiel gezeigt.

Beispiel:

```
case e0
  when e1,e2 => s1();
  when e3    => s2();s3();
  else s4();
end case
```

→

```
if e0 in [e1,e2] then s1();
elseif e0=e3 then s2();s3();
else s4();
end if;
```

- Umwandlung einer `if`-Anweisung bzw. eines `if`-Ausdrucks in eine `case`-Anweisung bzw. einen `case`-Ausdruck

  Name: `=case`

  Selektion: `stmt` bzw. `priSel`

  Bedingung: Eine `if`-Anweisung bzw. ein `if`-Ausdruck muß selektiert sein. Die Bedingung sowie alle in `elseif`-Zweigen enthaltene Bedingungen müssen entweder die Prüfung auf Gleichheit mittels `=` oder eine Anwendung des `in`-Operators darstellen. Sie müssen außerdem denselben Ausdruck als linken Operator besitzen.

  Bemerkung: Dies ist die Umkehrung der Transformation `=if`.

• Transformationen für Ausdrücke

- Negation von Ausdrücken

  Name: `negate`

  Selektion: `expr`

  Bemerkung: Der selektierte Ausdruck wird mittels des `not`-Operators negiert. Der resultierende Ausdruck wird soweit möglich vereinfacht. (Siehe Transformation `simplify`.)

- Vereinfachung von Ausdrücken

  Name: `simplify`

  Selektion: `expr`

  Bemerkung: Der selektierte Ausdruck wird falls möglich vereinfacht.

  Beispiele: `not a<b` → `a>=b`
  `not a notin b` → `a in b`
  `not (not a and b)` → `a or not b`

- Gesetze von De'Morgan

  Name: `DeMorgan`

  Selektion: `expr`

Bemerkung: Auf den selektierten Ausdruck werden die Gesetze von De'Morgan angewendet.

– Vertauschung von Operatoren

Name: `commutate`

Selektion: `expr`

Bedingung: Der selektierte Ausdruck muß aus der Anwendung eines beliebigen binären Operators bestehen.

Bemerkung: Die beiden Operanden des Operators werden vertauscht.

– Transformation gemäß Assoziativitätsgesetz

Name: `associate`

Selektion: `expr`

Bedingung: Der selektierte Ausdruck muß aus der Anwendung eines beliebigen binären Operators bestehen, wobei einer der beiden Operanden ebenfalls aus der Anwendung eines beliebigen binären Operators bestehen muß.

Bemerkung: Der Ausdruck wird dem Assoziativitätsgesetz folgend umgeformt. Da die beteiligten binären Operatoren beliebig sind, wird die Gültigkeit des Assoziativitätsgesetzes nicht überprüft.

– Transformation gemäß Distributivitätsgesetz

Name: `distribute`

Selektion: `expr`

Bedingung: Der selektierte Ausdruck muß aus der Anwendung eines beliebigen binären Operators bestehen, wobei einer der beiden Operanden ebenfalls aus der Anwendung eines beliebigen binären Operators bestehen muß.

Bemerkung: Der Ausdruck wird dem Distributivitätsgesetz folgend umgeformt. Da die beteiligten binären Operatoren beliebig sein können, wird die Gültigkeit des Distributivitätsgesetzes nicht überprüft.

– Transformation gemäß Distributivitätsgesetz (Umkehrung)

Name: `undistribute`

Selektion: `expr`

Bedingung: Der selektierte Ausdruck muß aus der Anwendung eines beliebigen binären Operators bestehen, wobei beide Operanden aus der Anwendung desselben beliebigen binären Operators bestehen müssen.

Bemerkung: Diese Transformation stellt die Umkehrung der vorigen dar.

- Transformationen für Zuweisungen

  – Umwandlung in die kombinierte Zuweisung

Name: `combine`

Selektion: `stmt`

Bedingung: Die selektierte Anweisung muß eine Zuweisung sein, auf deren linker Seite ein einfacher l-Value steht, der gleichzeitig auf der rechten Seite der Zuweisung Operand einer binären Operation ist.

Bemerkung: Die binäre Operation kann benutzerdefiniert sein.

Beispiele: `a := a+1` → `a +:= 1`

```
a(1):=a(1) !User if a=b then 1 else 0 end if
```
→
```
a(1) !User:=if a=b then 1 else 0 end if
```

– Dekombination einer kombinierten Zuweisung

Name: `decombine`

Selektion: `stmt`

Bedingung: Die selektierte Anweisung muß eine kombinierte Zuweisung sein.

Bemerkung: Dies ist die Umkehrtransformation von `combine`.

In Kapitel 3 waren verschiedene Transformationen gefordert, die leider nicht im Rahmen dieser Arbeit realisiert werden konnten. Dies waren neben den Transformationen zur Umwandlung von `for`- und `whilefound`-Schleifen in andere Schleifentypen und umgekehrt solche, die der Aufschlüsselung komplexer Mengenausdrücke dienen.

Eine `for`- bzw. `whilefound`-Anweisung läßt sich jedoch nicht durch eine einzelne `loop`-, `repeat`- oder `while`-Schleife ersetzen, da hier mindestens eine weitere Anweisung zum Initialisieren einer Hilfsvariable nötig ist. Eine einzelne Anweisung muß also bei einer solchen Transformation durch eine Folge von Anweisungen ersetzt werden. Mit dem **Synthesizer Generator** können jedoch nur solche Tranformationen realisiert werden, deren Ergebnis denselben Typ besitzt, wie die Selektion, auf die die Transformation angewendet wird. Eine Lösung für dieses Problem bietet erst der **Synthesizer Generator** Version 5.0. Diese wird in Kapitel 7 skizziert.

6.3 Werkzeuge zur semantischen Analyse

Der ProSet-Editor besitzt zwei Werkzeuge zur semantischen Analyse eines Programms: Die Querverweisliste und die Suche nach Deklaration und Benutzungen von Bezeichnern. Auf beide Werkzeuge wurde bereits in den Kapiteln 3 und in diesem Kapitel in der Beschreibung der Sicht **XREFVIEW** eingegangen. Die Benutzung der Querverweisliste ist weitgehend in letzterem beschrieben. Eine konkrete Beschreibung der Bedienung des Werkzeugs zur Suche nach Deklaration und Benutzungen von Bezeichnern folgt nun.

Dieses Werkzeug ist über ein Menü namens *Id* zu bedienen. Dieses ist ein Untermenü des Hauptmenüs des ProSet-Editors und befindet sich am linken Fensterrand. Die beiden enthaltenen Auswahlpunkte heißen *search-declaration* und *search-occurence*. Diese beiden Auswahlpunkte sind vom Benutzer nur dann zu selektieren, wenn die strukturelle Selektion auf einen Bezeichner verweist, der eine Variable, eine Prozedur, ein Modul oder einen Handler referenziert. Versucht der Benutzer die Funktionalität einer dieser Auswahlpunkte über die textuelle Eingabe von Befehlen zu erreichen, so erhält er die Fehlermeldung „Selection must be an ID!“.

Wählt der Benutzer den Auswahlpunkt *search-declaration*, so wird die Deklaration des aktuell selektierten Bezeichners vom ProSet-Editor dargestellt und selektiert. Kann eine Deklaration nicht gefunden werden, so wird die textuell erste Benutzung selektiert und es erscheint die Meldung „Declaration not found. First occurence selected.“.

Wird die Funktion *search-occurence* aufgerufen, zeigt der ProSet-Editor die nächste Benutzung des selektierten Bezeichners an und selektiert diese. Nach einem Aufruf von *search-declaration* können so durch mehrfaches Aufrufen dieses Befehls sukzessiv alle Benutzungen eines Bezeichners inspiziert werden. Wird keine weiter Benutzung des selektierten Bezeichners gefunden, meldet der ProSet-Editor „No further occurence found.“.

6.4 Anbindung fremder Werkzeuge

Aus dem ProSet-Editor heraus können zwei Fremdwerkzeuge kontextsensitiv aufgerufen werden: Der *P–File-Editor* sowie das Werkzeug *P–File-Tool*. Die Kommunikation und Koordination beider Werkzeuge mit dem **Synthesizer Generator** geschieht über das Nachrichtensystem *ToolTalk* [Mic93]. Beide Werkzeuge sollen hier nicht näher erläutert werden, da ihre Funktionsweise für diese Arbeit unerheblich ist. Die Werkzeuge werden über die Auswahlpunkte *P–File-Editor* bzw. *P–File-Tool* aufgerufen. Wie die Werkzeuge zur semantischen Analyse befinden sich diese Auswahlpunkte in dem Menü em Id. Beide Werkzeuge können nur dann aufgerufen werden, wenn sich die strukturelle Selektion auf einem zu deklarierenden Bezeichner in einer `persistent`-Deklaration befindet. Außerdem muß der Ausdruck am Ende der Deklaration aus einer String-Konstante bestehen, da diese Konstante den Namen des zu bearbeitenden *P–Files* bestimmt. Wird versucht eines der Werkzeuge über die textuelle Kommandoeingabe aufzurufen, ohne daß obige Bedingungen erfüllt sind, so quittiert der ProSet-Editor dies mit der Meldung „Selection must be an ID in persistent declaration!“ bzw. „Unknown P–File!“. Können die Werkzeuge nicht aufgerufen werden, weil die Kommunikation mit ihnen über *ToolTalk* fehlschlägt, erscheint die Meldung „No ToolTalk service available!“.

Bei korrektem Aufruf eines Werkzeugs wird diesem der selektierte Bezeichner sowie der Name des zu bearbeitenden *P–Files* übermittelt.

Kapitel 7

Abschluß

In diesem Kapitel soll nun zunächst die Version 5.0 des **Synthesizer Generator** kurz vorgestellt werden. Diese Version bietet entscheidende Neuerungen gegenüber der Version 4.2. Es konnte jedoch weitgehend nur die ältere Version benutzt werden, da die Version 5.0 leider erst gegen Beendigung dieser Arbeit zur Verfügung stand.

Anschließend wird ein Ausblick auf mögliche Anschlußarbeiten gegeben, die von der neuen Version des **Synthesizer Generator** wesentlich profitieren werden. Der Schwerpunkt dieses Ausblicks wird auf der Integration bzw. der Anbindung anderer Werkzeuge an den ProSet-Editor liegen.

Ein bewertender Rückblick auf die Erfahrungen mit dem **Synthesizer Generator** sowie auf die praktische Ausarbeitung wird dieses Kapitel beschließen.

7.1 Synthesizer Generator Version 5.0

Die Version 5.0 des **Synthesizer Generator** wurde gegenüber der Version 4.2 erheblich verbessert bzw. modifiziert. Die wesentlichen Neuerungen dieser Version werden nun kurz skizziert.

Neugestaltete Benutzungsoberfläche: Ein mit der Version 5.0 generierter Editor enthält eine verbesserte Benutzungsoberfläche. Diese unterscheidet sich jedoch nur optisch von der alten Oberfläche, deren Funktionalität vollständig übernommen wurde. Unter anderem Abbildung 6.1 in Kapitel 6.1 zeigt die neue Oberfläche. Zum Vergleich sei auf Abbildung 2.8 im Kapitel 2.3.3 verwiesen.

Anbindung eines HTML-Browsers als Hilfesystem: Die Version 4.2 des **Synthesizer Generator** sieht ein kontextsensitives, hypertextfähiges Hilfesystem vor. Hier muß für jede Hilfeseite eine eigene Datei erstellt werden, die neben dem eigentlichen Text noch verschiedene Steuersequenzen enthält, die z.B. Darstellung des Hilfetextes beeinflußen oder Links zu anderen Hilfeseiten enthalten. Die Darstellung dieser Seiten ist Aufgabe des Editors selbst.

Die neue Version des **Synthesizer Generator** verwendet demgegenüber ein externes Programm als Hilfesystem. So wird beim Aufruf der Hilfe vom Editor ein zuvor vom Benutzer zu spezifizierender HTML-Browser gestartet. Die verschiedenen Hilfeseiten müssen also im HTML-Format vorliegen. Dieses Format ist relativ verbreitet, so daß hierfür verschiedene WYSIWYG-Editoren bzw. Editor-Erweiterungen und auch Konvertierungsprogramme existieren. Dadurch wird die Erstellung der Hilfeseiten für den Editor erheblich vereinfacht.

SNOW **als Skriptsprache:** Die wohl wesentlichste Neuerung ist die Einführung der Sprache SNOW als Skriptsprache für die generierten Editoren. SNOW ist eine Untermenge der Sprache STK. Diese ist wiederum eine Erweiterung der Lisp-verwandten Programmiersprache Scheme.

Die Version 4.2 des **Synthesizer Generator** verwendete eine kleine Untermenge von SSL genannt Dynamic SSL als Skriptsprache. Die Einsatzmöglichkeiten hierfür sind jedoch sehr gering. Demgegenüber bieten SNOW-Skripte folgende Möglichkeiten:

- Konfiguration und Erweiterung von Editoren: Auf einfache Weise können existierende Editor-Kommandos modifiziert oder auch neue Kommandos einem Editor hinzugefügt werden. Diese Möglichkeit bestand wohl auch in der Vorgängerversion. Hier mußte dies jedoch auf einem sehr niedrigen Niveau in C geschehen. Da SNOW-Skripte vom Editor interpretiert werden und in externen Dateien vorliegen, kann nun auch der Benutzer des Editors diesen konfigurieren und erweitern.
- Anbindung von externen Werkzeugen: Externe Werkzeuge können mit dem Editor kommunizieren, indem sie SNOW-Skripte als Nachrichten an den Editor senden. Dieser führt die empfangenen Skripte dann aus. Da verschiedene SNOW-Befehle existieren, die einen (evtl. modifizierenden) Zugriff auf den AST ermöglichen, können die externen Werkzeuge so auf den AST zugreifen.
- Servermodus: Ein Editor kann im sogenannten Servermodus gestartet werden. In diesem Modus wird die Oberfläche des Editors selbst nicht dargestellt. Externe Werkzeuge (auch ein externer Editor) können trotzdem mit dem Editor kommunizieren und auf den AST zugreifen.

7.2 Ausblick

Mit der Implementation des syntaxgesteuerten Editors für PROSETist ein zentrales Element für eine inkrementorientierte Programmierentwicklungsumgebung geschaffen worden. Um den Programmierer während des gesamten Programmierzyklus zu unterstützen, ist es sinnvoll, ihm weitere Werkzeuge wie Debugger, Interpreter oder ein Versionskontrollsystem zur Verfügung zu stellen. Aufgabe wird es nun sein, neue Werkzeuge zu erstellen und neue oder auch bestehende Werkzeuge so an den Editor anzubinden, daß diese dem Benutzer möglichst nicht als Einzelwerkzeuge sondern als Teil der Entwicklungsumgebung erscheinen. Hierfür ist es nicht nur erforderlich, diesen Werkzeugen die interne Darstellung des Programmtextes zugänglich zu machen. Es ist auch sinnvoll, die Werkzeuge soweit möglich über die Oberfläche des Editors zu steuern. Somit bedarf die Integration weiterer Werkzeuge der Betrachtung folgender Aspekte:

- Datenintegration

  Datenintegration wird vom **Synthesizer Generator** nur rudimentär durch die Möglichkeit des Zugriffs auf den AST unterstützt.

  Schon in der Version 4.2 bot der **Synthesizer Generator** eine Vielzahl von C-Funktionen, mittels derer auf die interne Darstellung des Programmes zugegriffen werden konnte. Da diese Funktionen jedoch nicht dokumentiert waren, erscheinen erst die mit der Version 5.0 über die SNOW-Schnittstelle gebotenen Möglichkeiten als ausreichend, um einen komfortablen Zugriff zu gewährleisten. Da die SNOW-Schnittstelle auch die programmgesteuerte Durchführung von Transformationen ermöglicht, kann der AST nun auch modifiziert werden.

- Kontrollintegration

  In beiden Versionen 4.2 und 5.0 des **Synthesizer Generator** kann über *ToolTalk* eine eingeschränkte Kontrollintegration erreicht werden. Wie bereits erwähnt, bietet die Version 5.0 zusätzlich die Möglichkeit, SNOW-Skripte zu empfangen und auszuführen. Es können daher zwei grundsätzliche Konzepte der Realisierung und Anbindung neuer Werkzeuge unterschieden werden:

  – Interne Werkzeuge

    Es besteht die Möglichkeit, neue Werkzeuge direkt in den PROSET-Editor zu integrieren. Solche Werkzeuge könnten in C oder in SNOW geschrieben werden. Sinnvoll erscheint es, laufzeitkritische oder betriebssystemnahe Teile eines Werkzeugs in C zu implementieren, während solche Teile, die die obengenannten Aspekte betreffen, wesentlich einfacher in SNOW zu realisieren sein werden.

  – Externe Werkzeuge

    Weitere Werkzeuge können auch als eigene Prozesse laufen und über Inter-Prozeß-Kommunikation mit dem Editor zusammenarbeiten. Dies ist eine Möglichkeit, um auch schon bestehende Werkzeuge an den Editor anzubinden, wie für den *P–File-Editor* und das *P–File-Tool* geschehen. Eine Bedienung aus der Oberfläche des Editors heraus ist jedoch nur dann möglich, wenn solche Werkzeuge einerseits Kommandos über Inter-Prozeß-Kommunikation entgegennehmen und andererseits Ausgaben ebenfalls hierüber tätigen. Viel Spielraum ist bei der Realisierung der Editor-seitigen Schnittstelle gegeben: Empfangene Nachrichten können als SNOW-Skripte empfangen und ausgeführt werden. Dies ist eine (gemäß Dokumentation) einfach zu realisierende Möglichkeit. Denkbar wäre jedoch auch eine Schnittstelle auf einem sprachspezifischen Niveau: So könnte die Schnittstelle auf Anforderung komplexe Informationen, wie eine Liste aller sichtbaren Bezeichner o.ä. bereitstellen.

- Präsentationsintegration

  Die Version 4.2 des **Synthesizer Generator** ermöglicht das Erstellen komfortabler Eingabemasken über die C-Schnittstelle. Zukünftige Versionen des **Synthesizer Generator** versprechen jedoch auch hier eine wesentlich leichtere Handhabung, da sie STk, eine Implementation von Tk enthalten werden. Einfache menügesteuerte Eingaben sind auch mit den Versionen 4.2 bzw. 5.0 schon einfach möglich, indem neue Menüs einem Editor hinzugefügt werden oder aber bestehende Menüs modifiziert werden können.

  Über zusätzliche Sichten, die nicht die interne Repräsentation des Programmtextes, sondern beliebige abstrakte Syntaxbäume visualisieren, sind in SSL Ausgaben von Fremdwerkzeugen in der Oberfläche des ProSet-Editors einfach zu realisieren. Die Querverweisliste des ProSet-Editors ist hierfür ein Beispiel. Über die Snow-Schnittstelle ist ein einfaches Bewegen der strukturellen Selektion sowie die Ausgabe einzelner Meldungen möglich.

Zusammenfassend ist zu bemerken, daß die Version 5.0 des **Synthesizer Generator** die komfortable Integration verschiedenster Werkzeuge auf hohem Niveau erlauben wird.

7.3 Rückblick

Abschließend soll ein Rückblick auf die Realisierung dieser Arbeit gegeben werden. Hierfür soll zunächst die Eignung des **Synthesizer Generator** für diese Arbeit bewertet werden. Anschließend wird das Endergebnis dieser Arbeit, der ProSet-Editor, bewertet werden.

7.3.1 Eignung des Synthesizer Generator

Der **Synthesizer Generator** ist ein sehr leistungsfähiges und ausgereiftes Werkzeug, das eine komfortable Erstellung syntaxgesteuerter Editoren auf hohem Niveau ermöglicht. Dennoch sollen nun einige wenige Schwachpunkte genannt werden:

- Schwächen der Spezifikationssprache SSL

  SSL ist eine sehr komfortable und homogene Spezifikationssprache, die für ihr Einsatzgebiet sehr geeignet erscheint. Geringfügige Mängel sind jedoch trotzdem zu erkennen:

  - Beschränkter Zugriff auf Attribute

    Es ist leider nicht möglich, von einer Produktion aus auf Attribute zuzugreifen, die an Nachfahren der rechtsseitigen Phyla dieser Produktion gebunden sind. Solche Attribute müssen über alle zwischenliegenden Phyla weitergereicht werden. Hinderlich war dies für diese Arbeit vor allem bei der Realisierung des Attributs `usedIds`, das alle verwendeten Bezeichner im entsprechenden Teilbaum aufnimmt. Dieses Attribut mußte leider an über 40 Phyla gebunden werden.

  - Kein Zugriff auf Phyla in Richtung Wurzel des AST

    Während es über den Mechanismus des Pattern-Matching einfach möglich ist, Informationen über beliebige Nachfahren im AST zu erlangen, ist es nicht direkt möglich zu entscheiden, ob sich z.B. eine gegebene Anweisung in einem Prozedurrumpf oder im Rumpf des Hauptprogramms befindet. Solche Informationen sind jedoch z.B. nötig, um zu entscheiden, ob eine Transformation (in diesem Falle z.B. `<stmt>` → `return`) zulässig ist. Solche Informationen sind leider nur über entsprechende Attribute zu übermitteln. Ein Beispiel für ein solches Attribut ist in dieser Arbeit das Attribut `flags`.

  - Transformationen beziehen sich nur auf ein Phylum (Version 4.2)

    Wie bereits in Kapitel 6.2 erläutert, besteht nicht die Möglichkeit, beispielsweise eine einzelne Anweisung durch eine Folge von Anweisungen über eine Transformation zu ersetzen. Dies schränkt den Transformationsmechanismus erheblich ein. In der Version 5.0 ist es jedoch möglich, über SNOW-Skripte Folgen verschiedener Transformationen automatisch aufzurufen. Ein solches Skript kann auch über Betätigung eines Knopfes erfolgen, der sich an derselben Stelle befindet wie die Knöpfe für die Transformationen. Damit ist dieser Vorgang für den Benutzer transparent.

- Schwächen in der Anbindung von Werkzeugen (Version 4.2)

  Wie auch im letzten Abschnitt erläutert, sind erst in der Version 5.0 des **Synthesizer Generator** komfortable Möglichkeiten gegeben, den Editor um Werkzeuge zu erweitern. Die Werkzeuge zur semantischen Analyse wurden im Rahmen dieser Arbeit über die leider undokumentierte, sehr komplexe C-Schnittstelle des **Synthesizer Generator** realisiert. Dies stellte sich als sehr arbeits- und zeitintensiv heraus.

7.3.2 Beurteilung des ProSet-Editors

Es konnten nicht alle in der Aufgabenstellung formulierten Forderungen erfüllt werden: So konnten die dort erwähnten Transformationen aufgrund der oben erwähnten Schwäche des **Synthesizer Generator** nicht alle realisiert werden. Ebenfalls nicht realisiert wurden verschiedene Sichten für unterschiedliche Benutzergruppen: Dies hatte sich im Fortgang dieser Arbeit wie bereits erwähnt als unzweckmäßig herausgestellt. Der Editor an sich erscheint relativ stabil.

Die gegebene Komfortabilität wird jedoch mit einer hohen Komplexität der Bedienung erkauft. Erst im Zusammenspiel mit weiteren Werkzeugen wird der Editor seine Vorzüge voll herausstellen können. Die realisierten Werkzeuge für die semantische Analyse deuten dies jedoch schon jetzt an.

Anhang A

Spezifikation der abstrakten Syntax

Nun soll die dem ProSet-Editor zugrundeliegende abstrakte Syntax spezifiziert werden. Als Notation soll hierbei SSL verwendet werden.

- Produktionen für Programm-, Prozedur-, Handler und Moduldefinition.
  Enthalten in Datei "ProSet.phm.ssl".

```
/* Programm */

program: Program(id progBody ERRSTRING);
/* "id" ist Programmname im Header.
   Falls dieser vom id im Trailer nach dem Parsen
   abweicht, wird Id im Trailer angepasst und
   ERRSTRING erhaelt Fehlermeldung. */

progBody: ProgBody(comment decls stmts phmDefs);

/* Variablendeklarationen */

decls: NilDecls()
      |DeclsPair(decl decls);

decl: Decl(declKey constKey varDecls explAssos)
     |PersDecl(declKey constKey idList expr explAssos);

constKey: NilConstKey()
         |ConstKey();

declKey: NilDeclKey()
        |VisibleDeclKey()
        |HiddenDeclKey();

varDecls: NilVarDecls()
```

```
        |VarDeclsPair(varDecl varDecls);

varDecl: VarDecl(id varAss);

varAss: NilVarAss()
       |VarAssExpr(expr);

/* Prozedur-, Handler- und Moduldefinitionen */

phmDefs: NilPHMDefs()
        |PHMDefsPair(phmDef phmDefs);

phmDef: NilPHMDef()
       |ProcDef(id params progBody ERRSTRING)
       |HandlerDef(id params implAsso progBody ERRSTRING)
       |ModuleDef(id importList exportList progBody ERRSTRING) ;

/* Sonstiges */

implAsso: NilImplAsso()
         |NilImplAsso2()
         |ImplAsso(idList)
         |ImplOthers();

importList: NilImportList()
           |NilImportList2()
           |ImportList(idList);

exportList: NilExportList()
           |NilExportList2()
           |ExportList(idList);
```

- Produktionen für Anweisungen.
  Enthalten in Datei "ProSet.stmt.ssl".

```
/* Statements */

stmts: NilStmts()
      |StmtsPair(stmt stmts);

stmt: NilStmt()
     |LowStmt(comment lowStmt explAssos);
/* Um Kommentare und explizite Handlerassoziationen
   an alle Anweisungen zu binden, wird Nichtterminalsymbol
   "lowStmt" eingefuegt. */


lowStmt: AssStmt(lValue expr)
        |From(lValue sLValue)
        |FromE(lValue sLValue)
        |FromB(lValue sLValue)
        |CombAssStmt(lValue binOp expr)
        |IfStmt(expr stmts elifStmts elseStmts)
        |CaseStmt(expr stmtCases elseStmts)
        |LoopStmt(label loopStmt ERRSTRING)
         /* Fehlerhafe Label werden nach dem Parsen korrigiert
```

```
               und eine Warnung wird in "ERRSTRING" ausgegeben. */
            |ProcCallStmt(qualId lSelectors paramList)
            |LambdaCallStmt(lambda paramList)
            |SelfStmt(paramList)
            |Return(expr)
            |RCommit(expr)
            |Resume(expr)
            |Quit(optId)
            |Pass()
            |Stop(expr)
            |ForkStmt(expr)
            |SignalRetVal(lValue id paramList)
            |Signal(qualId lSelectors paramList)
            |NotifyRetVal(lValue id paramList)
            |Notify(qualId lSelectors paramList)
            |Escape(id paramList)
            |CCallStmt(id typeList)
             /* Anweisungen aus StdIO, zum String Scanning und fuer
                die Tupelraumkommunikation werden in jeweils
                eigenen Untermenues untergebracht. *
            |StdIO(stdIO)
            |StrScanStmt(strScan)
            |TupleStmt(tupleStmt);

/* StdIO */

stdIO: NilStdIO()
      |Put(paramList)
      |EPut(paramList)
      |FPut(paramList)
      |PutF(paramList)
      |EPutF(paramList)
      |FPutF(paramList)
      |Get(paramList)
      |FGet(paramList);

/* Anweisungen fuer Tupelraumkommunikation */

tupleStmt: NilTupleStmt()
          |Deposit(expr expr block)
          |Fetch(templates expr elseStmts)
          |Meet(templates expr elseStmts);

block: NilBlock()
      |NilBlock2()
       /* Eine Nullproduktione fuer Completing Term
          und eine fuer Placeholder Term. */
      |Block();


templates: NilTemplates()
          |TemplatePair(template templates);

template: Template(fields restriction tempStmts);

fields: NilFields()
       |FieldsPair(field fields);
```

```
field: NilField()
      |ExprField(expr)
      |FormalField(optSLValue into);

optSLValue: NilOptSLValue()
           |NilOptSLValue2()
            /* Eine Nullproduktione fuer Completing Term
               und eine fuer Placeholder Term. */
           |OptSLValue(sLValue);
/* Optionaler einfacher L-Value. Unterscheidet
   sich vom SLValue im Unparsing. */

into: NilInto()
     |NilInto2()
     |Into(expr);

tempStmts: NilTempStmts()
          |NilTempStmts2()
          |TempStmts(stmts);

/* Label */

label: NilLabel()
      |NilLabel2()
      |Label(id);

/* Sonstiges fuer if-, case- und Schleifenanweisungen */

elifStmts: NilElifStmts()
          |ElifStmtsPair(elifStmt elifStmts);

elifStmt: ElifStmt(expr stmts);

elseStmts: NilElseStmts()
          |NilElseStmts2()
          |ElseStmts(stmts);

stmtCases: NilStmtCases()
          |StmtCasesPair(stmtCase stmtCases);

stmtCase: StmtCase(exprs stmts);

loopStmt: NilLoopStmt()
         |Loop(stmts)
         |While(expr stmts)
         |Repeat(stmts expr)
         |For(iterator restriction stmts)
         |WhileFound(iterator restriction stmts);
```

- Produktionen für Ausdrücke.
  Enthalten in Datei "ProSet.expr.ssl".

```
/* Konstanten */
```

```
const: NilConst()
      |IntConst(INT)
      |FloatConst(REALLIT)
      |BoolConst(BOOLEANLIT)
      |StringConst(STRING)
      |EmptySet()
      |EmptyTuple()
      |Om();

/* Typen */

type: NilType()
     |AtomType()
     |BoolType()
     |IntType()
     |RealType()
     |StrType()
     |TupleType()
     |SetType()
     |FunctionType()
     |ModuleType()
     |InstanceType();

/* Binaere Operanden */

binOp: NilBinOp()
      |Or()
      |And()
      |Eq()
      |NEq()
      |Lo()
      |LoEq()
      |Gr()
      |GrEq()
      |In()
      |NotIn()
      |SubSet()
      |With()
      |Less()
      |LessF()
      |NPow()
      |Sum()
      |Diff()
      |Prod()
      |Div()
      |Mod()
      |Pot()
      |Maxi()
      |Mini()
      |UserBinOp(qualId)

       /* Binaere Compound Operanden */

      |NilComp()
      |OrComp()
      |AndComp()
      |EqComp()
```

```
       |NEqComp()
       |LoComp()
       |LoEqComp()
       |GrComp()
       |GrEqComp()
       |InComp()
       |NotInComp()
       |SubSetComp()
       |WithComp()
       |LessComp()
       |LessFComp()
       |SumComp()
       |DiffComp()
       |ProdComp()
       |DivComp()
       |ModComp()
       |PotComp()
       |MaxComp()
       |MinComp()
       |UserBinOpComp(qualId);

/* Unaere Operanden */

unOp: NilUnOp()
      |UPlus()
      |UMinus()
      |Hash()
      |Fork()
      |Not()
      |Pow()
      |Arb()
      |Random()
      |Domain()
      |Range()
      |Type()
      |IsMap()
      |IsSMap()
      |UserUnOp(qualId)

       /* Unaere Compound Operanden */

      |NilUComp()
      |UOrComp()
      |UAndComp()
      |UEqComp()
      |UNEqComp()
      |ULoComp()
      |ULoEqComp()
      |UGrComp()
      |UGrEqComp()
      |UInComp()
      |UNotInComp()
      |USubSetComp()
      |UWithComp()
      |ULessComp()
      |ULessFComp()
      |USumComp()
```

```
     |UDiffComp()
     |UProdComp()
     |UDivComp()
     |UModComp()
     |UPotComp()
     |UMaxComp()
     |UMinComp()
     |UserUnOpComp(qualId);

/* "Primary Selektors" */

priSel: NilPriSel()
       |HandlerPriSel(priSel handAsso)
       |QualIdPriSel(qualId)
       |SelectionPriSel(priSel lSelectors)
       |ProcCallPriSel(priSel paramList)
       |LambdaCallPriSel(lambda paramList)
       |SelfPriSel(paramList)
       |Closure(closure)
       |IfPriSel(expr expr elifExprs elseExpr)
       |CasePriSel(expr exprCases elseExpr)
       |TupleFormer(former)
       |SetFormer(former)
       |Dollar()
       |Argv()
       |CCallPriSel(id typeList id)
       |ExprPriSel(expr);

/* Ausdruecke */

expr: NilExpr()
     |PriSelExpr(priSel)
     |ConstExpr(const)
     |TypeExpr(type)
     |NilStrScanExpr()
     |StrScanExpr(strScan)
     |Quantifier(quantifier)
     |BinOpExpr(expr binOp expr)
     |UnOpExpr(unOp expr)
     |NewAtExpr()
     |Instantiate(expr instExportList instImportList);

/* Import-, Exportliste */

instExportList: NilInstExportList()
               |NilInstExportList2()
               |InstExportList(exprs);

instImportList: NilInstImportList()
               |NilInstImportList2()
               |InstImportList(idList);

/* Sonstiges fuer if- und case-Ausdruecke */

elifExprs: NilElifExprs()
          |ElifExprsPair(elifExpr elifExprs);
```

```
elifExpr: ElifExpr(expr expr);

elseExpr: NilElseExpr()
         |NilElseExprs2()
         |ElseExpr(expr);

exprCases: NilExprCases()
          |ExprCasesPair(exprCase exprCases);

exprCase: ExprCase(exprs expr);

/* Ausdruckslisten */

exprs: NilExprs()
      |ExprPair(expr exprs);

optExprs: NilOptExprs()
         |NilOptExprs2()
         |Exprs(exprs);

/* closure */

closure: NilClosure()
        |SelfClosure()
        |LambdaClosure(lambda)
        |QualIdClosure(qualId);
```

- Produktionen, die sowohl von Anweisungen als auch von Ausdrücken verwendet werden.
  Enthalten in Datei "ProSet.expr.ssl".

```
/* Kommentare */

comment: NilComment()
        |NilComment2()
        |Comment(COMMENT);

/* Ids, Id-Listen */

id: NilId()
   |exported Id(STR);

optId: NilOptId()
      |NilOptId2()
      |OptId(STR);

qualId: NilQualId()
       |SimpleQualId(id)
       |QualId(id optId);

listId: NilListId()
       |ListId(STR);

idList: NilIdList()
```

```
       |IdListPair(listId idList);

/* Iteratoren, L-Values */

iterator: NilIterator()
         |IteratorPair(simpleIt iterator);

simpleIt: NilSimpleIt()
         |InIt(itLValue expr)
         |MapIt(itLValue expr mapSel);

mapSel: NilMapSel()
       |OneMapSel(itLValue)
       |MulMapSel(itLValue);

lValue: NilLValue()
       |SLValue(sLValue)
       |exported MultipleLValue(lVComponents);

sLValue: exported QualIdSLValue(qualId lSelectors);

lVComponents: NilLVComponents()
             |LVComponents(lVComponent lVComponents);

lVComponent: NilLVComponent()
            |LValueLVComponent(lValue)
            |DummyLVComponent();

itLValue: NilItLValue()
         |ItSLValue(itSLValue)
         |MultipleItLValue(itLVComponents);

itSLValue: IdItSLValue(id lSelectors);

itLVComponents: NilItLVComponents()
               |ItLVComponents(itLVComponent itLVComponents);

itLVComponent: NilItLVComponent()
              |LValueItLVComponent(itLValue)
              |DummyItLVComponent();

restriction: NilRestriction()
            |NilRestriction2()
            |Restriction(expr);

/* Lambda-Definition */

lambda: Lambda(params progBody);

/* Quantifizierer */

quantifier: NilQuantifier()
           |exported ExistsQuantifier(iterator restriction)
           |exported ForAllQuantifier(iterator restriction);


/* L-Selektoren */
```

```
lSelectors: NilLSelectors()
           |LSelectorsPair(lSelector lSelectors);

lSelector: NilLSelector()
          |LSelec1(exprs)
          |LSelec2(expr)
          |LSelec3(expr expr)
          |LSelec4(exprs);

/* Tupel- und Mengen-Former */

former: NilFormer()
       |Enumeration(optExprs)
       |Interval(expr expr)
       |StepInterval(expr expr expr)
       |Description(expr iterator restriction);

/* String Scanning */

strScan: NilStrScan()
        |Any(sLValue expr)
        |RAny(sLValue expr)
        |Break(sLValue expr)
        |RBreak(sLValue expr)
        |Len(sLValue expr)
        |RLen(sLValue expr)
        |LPad(sLValue expr)
        |RPad(sLValue expr)
        |Match(sLValue expr)
        |RMatch(sLValue expr)
        |NotAny(sLValue expr)
        |RNotAny(sLValue expr)
        |Span(sLValue expr)
        |RSpan(sLValue expr);

/* Handler-Assoziationen */

explAsso: ExplAsso(handAsso);

explAssos: NilExplAssos()
          |ExplAssoPair(explAsso explAssos);

handAsso: HandAsso(idList id)
         |OthersAsso(id);

/* Parameter, Parameterlisten */

paramList: ParamList(optExprs);

paramLists: NilParamLists()
           |ParamListPair(paramList paramLists);

params: NilParams()
       |ParamsPair(param params);

param: Param(paramMode id);
```

```
paramMode: NilMode()
          |RDMode()
          |WRMode()
          |RWMode();

/* Typlisten fuer C-Calls */

typeList: NilTypeList()
         |TypeListPair(idPair typeList);

idPair: IdPair(id id);
```

Anhang B

Attributierung

Im folgenden wird die Attributierung des AST erläutert. Hierzu werden zu jedem Attribut folgende Angaben gemacht:

Name: Hier wird der Name des Attributs genannt.

Attributtyp: Dieser Eintrag beschreibt die Klassifizierung des Attributs in *inherited*, *synthesized* oder *local*.

Werttyp: Der Typ der Werte, die das Attribut speichert, wird hier festgehalten.

Phyla/Produktionen: Alle Phyla bzw. Produktionen, an die das Attribut gebunden ist, werden hier aufgezählt.

Funktion: Hier wird kurz die Funktion des Attributs erläutert.

Abhängigkeiten: Alle Attribute, die dieses Attribut direkt beeinflussen, werden hier genannt.

- **Name:** `env`

  Attributtyp: synthesized

  Werttyp: `environment`

  Phyla/Produktionen:
  `former, lambda, loopStmt, phmDef, program, quantifier`

  Funktion:
  Enthält die Bezeichner der im entsprechenden Sichtbarkeitsbereich deklarierten Variablen, Prozeduren, Handler, Module

  Abhängigkeiten:

former.env:
`former.env, lambda.env, loopStmt.env, phmDef.env, program.env, quantifier.env`

lambda.env:
`former.env, lambda.env, loopStmt.env, phmDef.env, program.env, quantifier.env`

loopStmt.env:
`former.env, lambda.env, loopStmt.env, phmDef.env, program.env, quantifier.env`

phmDef.env:
`phmDef.env, program.env`

quantifier.env:
`former.env, iterator.locEnv, lambda.env, loopStmt.env, phmDef.env, program.env, quantifier.env`

- **Name:** `locEnv`

  Attributtyp: synthesized

  Werttyp: `environment`

  Phyla/Produktionen:
  `explAsso, explAssos, exportList, handAsso, implAsso, importList, instImportList, iterator, itLValue, itLVComponent, itLVComponents, itSLValue, simpleIt`

  Funktion:
  Enthält die Bezeichner der im entsprechenden Sichtbarkeitsbereich deklarierten Variablen, Prozeduren, Handler, Module

  Abhängigkeiten:

  explAsso.locEnv:
  `handAsso.locEnv`

  explAssos.locEnv:
  `explAsso.locEnv, explAssos.locEnv`

  iterator.locEnv:
  `iterator.locEnv, simpleIt.locEnv`

  itLValue.locEnv:
  `itLVComponents.locEnv, itSLValue.locEnv`

  itLVComponents.locEnv:
  `itLVComponent.locEnv, itLVComponents.locEnv`

  simpleIt.locEnv:
  `itLValue.locEnv`

- **Name:** `usedIds`

  Attributtyp: synthesized

  Werttyp: `environment`

  Phyla/Produktionen:

  `closure, elifExpr, elifExprs, elifStmt, elifStmts, elseExpr, elseStmts, expr, exprCase, exprCases, exprs, former, iterator, itLValue, itLVComponent, itLVComponents, itSLValue, loopStmt, lowStmt, lSelector, lSelectors, lValue, lVComponent, lVComponents, mapSel, optExprs, paramList, priSel, progBody, quantifier, restriction, simpleIt, sLValue, stdIO, stmt, stmtCase, stmtCases, stmts, strScan, template, templates, tempStmts, tupleStmt`

  Funktion:

  Enthält alle im entsprechenden Sichtbarkeitsbereich verwendeten Bezeichner

  Abhängigkeiten:

  elifExpr.usedIds:
  `exprs.usedIds`

  elifExprs.usedIds:
  `elifExpr.usedIds, elifExprs.usedIds`

  elifStmt.usedIds:
  `expr.usedIds, stmts.usedIds`

  elifStmts.usedIds:
  `elifStmt.usedIds, elifStmts.usedIds`

  elseExpr.usedIds:
  `exprs.usedIds`

  elseStmts.usedIds:
  `stmts.usedIds`

  expr.usedIds:
  `priSel.usedIds, strScan.usedIds`

  exprCase.usedIds:
  `expr.usedIds, exprs.usedIds`

  exprCases.usedIds:
  `exprCase.usedIds, exprCases.usedIds`

  exprs.usedIds:
  `expr.usedIds, exprs.usedIds`

  former.usedIds:
  `expr.usedIds, iterator.usedIds, optExprs.usedIds, restriction.usedIds`

  iterator.usedIds:
  `iterator.usedIds, simpleIt.usedIds`

  itLValue.usedIds:
  `itLVComponents.usedIds, itSLValue.usedIds`

lVComponent.usedIds:
`lValue.usedIds`

itLVComponents.usedIds:
`itLVComponent.usedIds, itLVComponents.usedIds`

itSLValue.usedIds:
`lSelectors.usedIds`

loopStmt.usedIds:
`expr.usedIds, iterator.usedIds, stmts.usedIds, restriction.usedIds`

lowStmt.usedIds:
`elifStmts.usedIds, elseStmts.usedIds, expr.usedIds, loopStmt.usedIds, lSelectors.usedIds, lValue.usedIds, paramList.usedIds, sLValue.usedIds, stdIo.usedIds, stmts.usedIds, stmtCases.usedIds, strScan.usedIds, tupleStmt.usedIds`

lSelector.usedIds:
`expr.usedIds, exprs.usedIds`

lSelectors.usedIds:
`lSelector.usedIds, lSelectors.usedIds`

lValue.usedIds:
`lVComponents.usedIds, sLValue.usedIds`

lVComponent.usedIds:
`lValue.usedIds`

lVComponents.usedIds:
`lVComponent.usedIds, lVComponents.usedIds`

mapSel.usedIds:
`itLValue.usedIds`

optExprs.usedIds:
`exprs.usedIds`

paramList.usedIds:
`optExprs.usedIds`

priSel.usedIds:
`closure.usedIds, elifExprs.usedIds, elseExpr.usedIds, exprCases.usedIds, former.usedIds, lSelectors.usedIds, paramList.usedIds, priSel.usedIds`

progBody.usedIds:
`stmts.usedIds`

quantifier.usedIds:
`iterator.usedIds, restriction.usedIds`

restriction.usedIds:
`expr.usedIds`

simpleIt.usedIds:
`expr.usedIds, itLValue.usedIds, mapSel.usedIds`

sLValue.usedIds:
`lSelectors.usedIds`

stdIO.usedIds:
`paramList.usedIds`

stmt.usedIds:
`lowStmt.usedIds`

stmtCase.usedIds:
`exprs.usedIds, stmts.usedIds`

stmtCases.usedIds:
`stmtCase.usedIds, stmtCases.usedIds`

stmts.usedIds:
`stmt.usedIds, stmts.usedIds`

strScan.usedIds:
`expr.usedIds, sLValue.usedIds`

template.usedIds:
`restriction.usedIds, tempStmts.usedIds`

templates.usedIds:
`template.usedIds, templates.usedIds`

tempStmts.usedIds:
`stmts.usedIds`

tupleStmt.usedIds:
`elseStmts.usedIds, expr.usedIds, templates.usedIds`

- **Name:** `xRefs`

  Attributtyp: local

  Werttyp: `xReferences`

  Phyla/Produktionen:
  `ProgBody`

  Funktion:
  Enthält alle im entsprechenden Sichtbarkeitsbereich sichtbaren Bezeichner, sowie verschiedene Flags für diese.

  Abhängigkeiten:

  ProgBody:
  `lambda.env, phmDef.env, progBody.usedIds, program.env`

- **Name:** `name`

  Attributtyp: synthesized

  Werttyp: `id`

  Phyla/Produktionen:
  `program, phmDef`

  Funktion:
  Enthält eine Programm-, Prozedur, Handler oder Modulnamen.

Abhängigkeiten:

–

- **Name:** `precedence`

  Attributtyp: inherited

  Werttyp: `INT`

  Phyla/Produktionen:
  `expr, priSel`

  Funktion:
  Enthält die Präzedenz des Ausdrucks oder des "Primary Selectors".

  Abhängigkeiten:

  expr:
  `binOp.assoc, binOp.precedence, expr.precedence, priSel.precedence`

  priSel:
  `expr.precedence, priSel.precedence`

- **Name:** `precedence`

  Attributtyp: synthesized

  Werttyp: `INT`

  Phyla/Produktionen:
  `binOp`

  Funktion:
  Enthält die Präzedenz des binären Operators.

  Abhängigkeiten:

  –

- **Name:** `assoc`

  Attributtyp: synthesized

  Werttyp: `BOOL`

  Phyla/Produktionen:
  `binOp`

  Funktion:
  Enthält die Information, ob die für den binären Operator das Assoziativgesetz gilt.

  Abhängigkeiten:

  –

- **Name:** `enabled`

  Attributtyp: inherited

Werttyp: `BOOL`

Phyla/Produktionen:
`into`

Funktion:
Enthält die Information, ob die Produktion Into im Kontext erlaubt ist.

Abhängigkeiten:

into.enabled:
`tupleStmt.intoEnabled`

- **Name:** `intoEnabled`

  Attributtyp: synthesized

  Werttyp: `BOOL`

  Phyla/Produktionen:
  `tupleStmt`

  Funktion:
  Enthält die Information, ob die Produktion Into im Kontext erlaubt ist.

  Abhängigkeiten:
  –

- **Name:** `flags`

  Attributtyp: inherited

  Werttyp: `INT`

  Phyla/Produktionen:
  `closure, declKey, expr, param, params, priSel, progBody, restriction, stmt, stmts`

  Funktion:
  Enthält verschiedene Flags.

Name	Wert	Phylum befindet sich in
ppProcedure	1	Prozedur
ppModule	2	Modul
ppHandler	4	Handler
ppPHM	7	Prozedur, Modul oder Handler
ppInUnLabLoop	8	Schleife ohne Label
ppInLabLoop	16	Schleife mit Label
ppInLoop	24	Schleife
ppInLambda	32	Lambda-Definition
ppInPriSel	64	”Primary Selector”

  Abhängigkeiten:

  closure.flags:
  `priSel.flags`

declKey.flags:
`progBody.flags`

expr.flags:
`expr.flags, priSel.flags, stmt.flags`

param.flags:
`params.flags`

params.flags:
`params.flags`

priSel.flags:
`expr.flags, priSel.flags`

stmt.flags:
`stmt.flags, stmts.flags`

stmts.flags:
`progBody.flags, stmt.flags`

- **Name:** `cr`

  Attributtyp: synthesized

  Werttyp: `STR`

  Phyla/Produktionen:
  `progBody`

  Funktion:
  Wird für Unparsing verwendet.

  Abhängigkeiten:
  –

- **Name:** `labelEnv`

  Attributtyp: local

  Werttyp: `labels`

  Phyla/Produktionen:
  `ProgBody, LoopStmt`

  Funktion:
  Enthält alle im entsprechenden Sichtbarkeitsbereich verwendeten Label.

  Abhängigkeiten:

  LoopStmt.labelEnv:
  `LoopStmt.labelEnv, ProgBody.labelEnv`

- **Name:** `oneKey`

  Attributtyp: local

  Werttyp: `BOOL`

  Phyla/Produktionen:
  `Decl, PersDecl`

  Funktion:
  Enthält die Information, ob mindestens eines der Schlüsselwörter

`"constant"`, `"visible"` oder `"hidden"` bei der Variablendeklaration angegeben ist.

Abhängigkeiten:

–

- **Name:** `necStr`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `NilDeclKey`

  Funktion:
  Falls kein Deklarationschlüssel angegeben, enthält diese Attribut das Schlüsselwort `"hidden"`.

  Abhängigkeiten:

  NilDeclKey.necStr:
  `Decl.oneKey, PersDecl.oneKey`

- **Name:** `optStr`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `NilDeclKey`

  Funktion:
  Enthält `"[Declaration-Key]"`, falls die Deklaration mindestens einen Schlüssel enthält.

  Abhängigkeiten:

  NilDeclKey.optStr:
  `NilDeclKey.necStr`

- **Name:** `const`

  Attributtyp: local

  Werttyp: `BOOL`

  Phyla/Produktionen:
  `Decl, PersDecl`

  Funktion:
  Enthält die Information, ob eine Konstante deklariert wird.

  Abhängigkeiten:

  –

- **Name:** `error`

  Attributtyp: local

Werttyp: `STR`

Phyla/Produktionen:
`Continue, Dollar, Escape, HandAsso, HandlerDef, IdItSLValue, Into, LambdaCallPriSel, LambdaCallStmt, LambdaClosure, ListId, ModuleDef, Notify, NotifyRetVal, OthersAsso, Param, PersDecl, ProCallStmt, ProcCallPriSel, ProcDef, QualIdClosure, QualIdSLValue, Quit, RCommit, Resume, Return, SelfClosure, SelfPriSel, SelfStmt, Signal, SignalRetVal, UserBinOp, UserBinOpComp, UserUnOp, UserUnOpComp, VarDecl`

Funktion:
Enthält eine eventuelle Fehlermeldung oder Warnung.

Abhängigkeiten:
–

- **Name:** `error2`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `HandlerDef, ModuleDef, SelfClosure, SelfPriSel, SelfStmt`

  Funktion:
  Enthält eine eventuelle zweite Fehlermeldung oder Warnung.

  Abhängigkeiten:
  –

- **Name:** `warning`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `ModuleDef, Param`

  Funktion:
  Enthält eine eventuelle Warnung.

  Abhängigkeiten:
  –

- **Name:** `nec`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `NilVarAss`

  Funktion:
  Enthält `"Assignment"`, falls eine Konstante deklariert wird.

Abhängigkeiten:

NilVarAss.nec:
`Decl.const`

- **Name:** `labelId`

  Attributtyp: local

  Werttyp: `id`

  Phyla/Produktionen:
  `LoopStmt`

  Funktion:
  Enthält NilId() bei Schleifen ohne Label, ansonsten den Id des Labels.

  Abhängigkeiten:
  –

- **Name:** `labelStr`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `For, Loop, Repeat, While, WhileFound`

  Funktion:
  Entält bei Schleifen ohne Label eines der Schlüsselworter `"Loop "`, `"While"` usw., ansonsten den String des Ids des Labels.

  Abhängigkeiten:

  For.labelStr:
  `LoopStmt.labelId`

  Loop.labelStr:
  `LoopStmt.labelId`

  Repeat.labelStr:
  `LoopStmt.labelId`

  While.labelStr:
  `LoopStmt.labelId`

  WhileFound.labelStr:
  `LoopStmt.labelId`

- **Name:** `phOpt`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `NilExpr`

  Funktion:
  Falls der Ausdruck optional ist, enthält dieses Attribut `"[Expr]"`.

Abhängigkeiten:

NilExpr.phOpt:
`expr.opt`

- **Name:** `phNec`

  Attributtyp: local

  Werttyp: `STR`

  Phyla/Produktionen:
  `NilExpr`

  Funktion:
  Falls der Ausdruck nicht optional ist, enthält dieses Attribut `"<Expr>"`.

  Abhängigkeiten:

  NilExpr.phOpt:
  `expr.opt`

- **Name:** `lp`

  Attributtyp: local

  Werttyp: `unparsingCommand`

  Phyla/Produktionen:
  `BinOpExpr`

  Funktion:
  Dient der minimalen Klammerung von Ausdrücken.

  Abhängigkeiten:

  BinOpExpr.lp:
  `binOp.precedence, expr.precedence`

- **Name:** `rp`

  Attributtyp: local

  Werttyp: `unparsingCommand`

  Phyla/Produktionen:
  `BinOpExpr`

  Funktion:
  Dient der minimalen Klammerung von Ausdrücken.

  Abhängigkeiten:

  BinOpExpr.lp:
  `binOp.precedence, expr.precedence`

Literaturverzeichnis

[BC88] P. Borras und D. Clement. CENTAUR: the System. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988.

[BGdV92] R. A. Ballance, S. L. Graham und M. L. Van de Vanter. The Pan Language-Based Editing System. *ACM Transactions on Software Engineering and Methodology*, 1(1), 1992.

[Bud92] F. Buddrus. Generierung von Syntaxgesteuerten Werkzeugen auf der Basis eines Objektorientierten Datenbanksystems. Diplomarbeit, Universität Dortmund, Lehrstuhl Informatik 10, 1992.

[Dar87] S. A. Dart. *Software Development Environments.* Software Engineering Inst., Carnegie Mellon University, Pittsburgh, Pa., 1987.

[DFH+92] E.-E. Doberkat, W. Franke, W. Hasselbring, C. Pahl, H.-G. Sobottka und B. Sucrow. ProSet - Prototyping with Sets - Language Definition. Technischer Bericht 02-92, Universität Essen, 1992.

[DGGKL84] V. Donzeau-Gouge, G.Huet, G. Kahn und B. Lang. Programming Environments based on Structured Editors: The MENTOR Experience. In E. Sandewall D. R. Barstow, H. W. Shrobe, Hrsg., *Programming Environments*, Kapitel 7, Seiten 128–140. McGraw-Hill, New-York, 1984.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang und B. Melese. Document Structure and Modularity in MENTOR. *Proc. of the 1st SIGPlan/SIGSoft Symposium on Practical Software Development Environments*, 22(5), 1984.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer und A. Schurr. Building Integrated Software Environments Part I: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[EW89] G. Engels und W.Schäfer. *Programmentwicklungsumgebungen - Konzepte und Realisierung.* Leitfäden der angewandten Informatik. Teubner, Stuttgart, 1989.

[HN86] N. Habermann und D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Joh75] S. C. Johnson. *Yacc - Yet Another Compiler-Compiler.* AT&T Bell Laboratories, 1975.

[Kah87] G. Kahn. Natural Semantics. *Technical Report 601, INRIA*, 1987.

[Kas80] U. Kastens. Ordered Attributed Grammars. *ACTA Informatica*, 13(3):229–256, 1980.

[Kas90] U. Kastens. *Übersetzerbau.* Handbuch der Informatik. Oldenbourg, München, Wien, 1990.

[KLLMM94] J. L. Knudsen, M. Löfgren, O. Lehrmann-Madsen und B. Magnusson. *Object-Oriented Environments - The Mjølner Approach.* The Object-Oriented Series. Prentice-Hall, New York [u.a.], 1994.

[Knu68] D. E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Les75] M. E. Lesk. *Lex - a Lexical Analyzer Generator.* AT&T Bell Laboratories, 1975.

[Mic93] Sun Microsystems. *ToolTalk 1.1.1 User's Guide.* Sun Microsystems, 1991 - 1993.

[Rei85] S. P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering*, SE-11(13), 1985.

[RT89a] T. W. Reps und T. Teitelbaum. *The Synthesizer Generator - a System for Constructing Language Based Editors.* Texts and Monographs in Computer Science. Springer, New York [u.a.], 1989.

[RT89b] T. W. Reps und T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer, New York [u.a.], 1989.

[SBA92] M. A. Schoonover, J. S. Bowie und W. R. Arnold. *GNU Emacs. Unix Text Editing and Programming.* Addison Wesley, New York [u.a.], 1992.

[SL93] B. Staudt-Lerner. Contrasting Approaches of Two Environment Generators: The Synthesizer Generator and Pan. 1993.

[WHK88] W. M. Waite, V. P. Heuring und U. Kastens. Configuration Control in Compiler Construction. *International Workshop on Software Version and Configuration Control '88*, 1988.