# Echte Compilergenerierung

## Effiziente Implementierung einer abgeschlossenen Theorie

Jochen Demuth, Stephan Weber,
Sönke Kannapinn, Mario Kröplin

Bericht 1997 / 6

# Echte Compilergenerierung

## Effiziente Implementierung einer abgeschlossenen Theorie

Jochen Demuth, Stephan Weber,
Sönke Kannapinn, Mario Kröplin

Bericht 1997 / 6

Zur Veröffentlichung empfohlen von Prof. Dr. Bleicke Eggers

## Kurzfassung

Übersetzerbau ist heutzutage so gut verstanden, daß immer weniger einzusehen ist, warum Programmiersprachenbeschreibungen nicht gänzlich formal angegeben werden, so daß sogar Compiler automatisch generiert werden können. Die zur Beschreibung eingesetzten Attributgrammatiken können nicht befriedigen, weil sie sich jeweils auf eine Programmiersprache abstützen: Deshalb herrscht eine babylonische Verwirrung unter der Vielzahl existierender Generatorsysteme, und schlimmer noch enthalten entsprechende „Spezifikationen“ große Anteile von Programmen. Eine vielversprechende Alternative bietet der geschlossene Kalkül der Zweistufengrammatiken, welche allein auf dem Prinzip der kontextfreien Grammatiken basiert. Für die analyseorientiert eingeschränkten Erweiterten Affixgrammatiken enthält dieser Bericht die Dokumentation und vollständige Implementierung eines Systems zur Generierung praxistauglicher Compiler. Zugehörige Beispiele im Anhang sollen die Ausdruckskraft des Spezifikationsformalismus' demonstrieren.

## Abstract

Today, compiler construction is so well-understood that it is more and more unreasonable why programming language descriptions are still not given completely formally, so that even compilers could be generated automatically. The attribute grammars used for the description are unsatisfactory as they lean on a programming language each: Therefore, a Babylonian confusion among the numerous existing generator systems has emerged; and, worse still, respective “specifications” incorporate large quantities of programs. A promising alternative is offered by the closed calculus of two-level grammars, which are solely based on the principle of the context-free grammars. For the analysis-orientedly restricted Extended Affix Grammars, this report comprises the documentation and full implementation of a system generating effective compilers. Accompanying examples in the appendices shall demonstrate the expressiveness of the specification formalism.

# Vorwort

Ein Programm wird üblicherweise als Folge von Zeichen repräsentiert. Zugrunde liegt aber eine hierarchische Struktur, nach der Bestandteile (Deklarationen, Anweisungen, Ausdrücke, ...) gebildet und zusammengesetzt sind. Zur endlichen Beschreibung der Bestandteile und ihrer Zusammensetzung erweist sich eine simultan-induktive Definition als angemessen. Hinreichend einfache simultan-induktive Definitionen können zudem prägnant im Formalismus der kontextfreien Grammatiken angegeben werden. Wird dies beim Entwurf einer Programmiersprache berücksichtigt, so kann eine kontextfreie Grammatik im Report zur Vermittlung der Konstruktion von Programmen sowie der Rekonstruktion der Struktur von Programmen dienen. Daher können die kontextfreien Grammatiken als grundlegend für die gesamte Informatik angesehen werden, indem sie eine brauchbare endliche Beschreibung unendlicher Mengen ermöglichen und weiter eine Voraussetzung für das Verständnis von Programmen bilden.

Andererseits müssen neben der Struktur von Programmen auch Kontextabhängigkeiten (z.B. Bezeichneridentifikation und Typprüfung) ausgedrückt werden. Prinzipiell ist die kontextfreie Sprache gegenüber der Programmiersprache zu groß: Ein stark reduziertes Beispiel ist die Menge aller Zeichenfolgen $w\,c\,w$, die im Gegensatz zur umfassenden Menge aller Zeichenfolgen $w\,c\,w'$ nicht durch eine kontextfreie Grammatik beschrieben werden kann, wenn $w$ und $w'$ beliebige Folgen der Buchstaben $a$ und $b$ sind. Die informelle Ergänzung um Kontextabhängigkeiten, wie sie üblicherweise praktiziert wird, scheint bereits für einfache Programmiersprachen [ReiWi] zu versagen. Dagegen ist eine formale Beschreibung durch die mächtigeren kontextsensitiven Grammatiken sogar für das triviale Beispiel schwer verständlich, und zudem geht die Struktur dabei verloren. Die entscheidende Idee ist, daß die informell den Bestandteilen eines Programms zugeordneten Eigenschaften (z.B. Umgebung und Typ) formal als Parameter der Symbole der kontextfreien Grammatik aufgefaßt werden.

Bei Attributgrammatiken heißen diese Parameter Attribute, und nach dem Aufbau des Strukturbaums gemäß der kontextfreien Grammatik erfolgt die Parameterberechnung anhand von zusätzlich angegebenen Attributauswertungsregeln. So kann die Semantik einer kontextfreien Sprache definiert werden, indem der Wert eines Attributs des Startsymbols beispielsweise als Übersetzung angesehen wird, und zudem können einschränkende Kontextbedingungen formal ausgedrückt werden. Allerdings bilden die Attributgrammatiken einen sogenannten offenen Kalkül, insofern die Beschreibung der Attributauswertungsregeln einen weiteren Formalismus erfordert. Üblicherweise wird dafür eine (evtl. spezielle) Programmiersprache verwendet, damit automatisch effiziente Compiler generiert werden können, die im wesentlichen die angegebenen Routinen in der richtigen Reihenfolge ausführen. Zur Vermittlung von Programmiersprachen aber sind entsprechende Attributgrammatiken aufgrund ihres operationellen Charakters ungeeignet, zumal das Verständnis der dabei verwendeten Programmiersprache vorausgesetzt wird.

Nicht Berechnungen, sondern die Werte der Parameter in den Vordergrund zu stellen zeichnet dann auch den entgegengesetzten Ansatz aus. Damit können Kontextabhängigkeiten von vornherein berücksichtigt werden, indem Symbole der zugrundeliegenden kontextfreien Grammatik zusammen mit passenden Parameterwerten als die eigentlichen Symbole angesehen werden. Aus endlich vielen Regelmustern lassen sich nun über den eigentlichen Symbolen im allgemeinen unendlich viele kontextfreie Regeln erzeugen, die dann direkt die durch Kontextbedingungen eingeschränkte Sprache beschreiben. Das einfachste Mittel, dabei für korrespondierende Parameter die Erzeugung der Regeln zu beeinflussen, ist die Forderung der konsistenten Ersetzung der innerhalb eines Regelmusters mehrfach vorkommenden Variablen (durch gleiche Werte). Bei Zweistufengrammatiken werden auch die Parameterwerte durch kontextfreie Grammatiken beschrieben, so daß sich allein mit der konsistenten Ersetzung, minimal bezüglich der verwendeten Formalismen, ein geschlossener Kalkül ergibt. Eine Zweistufengrammatik ermöglicht also die endliche Beschreibung eines

unendlichen kontextfreien Regelsystems und auf diesem Weg eine indirekte Beschreibung einer unendlichen Sprache.

Nachfolgend soll für das Beispiel der Zeichenfolgen $w\,c\,w$ eine Zweistufengrammatik vorgestellt werden, ohne erst irgendeine Notation einzuführen. Jede kontextfreie Grammatik zur Beschreibung aller Folgen der Buchstaben $a$ und $b$ kann die Grundlage sowohl für die Parameterregeln als auch für die Regelmuster bilden. Mit den erzeugten Regeln soll dann ein Symbol, das mit einer Buchstabenfolge parametrisiert ist, gerade diese Buchstabenfolge beschreiben, d.h., die zugehörigen Regelmuster müssen eine Anhebung der beschriebenen Folge auf die Ebene der Parameter ausdrücken. Schließlich wird bei einem Regelmuster für die Zeichenfolgen $w\,c\,w'$ die konsistente Ersetzung dazu ausgenutzt, die geforderte Gleichheit der Buchstabenfolgen $w$ und $w'$ zu erreichen. Derartige Zweistufengrammatiken sind mit etwas Übung auf den ersten Blick verständlich; aber schon eine geforderte Ungleichheit der beiden Buchstabenfolgen läßt sich im reinen Kalkül nicht mehr elegant formulieren.

Dennoch genügt die konsistente Ersetzung, um kompliziertere Kontextabhängigkeiten auszudrükken, da die erzeugten Regeln neben der Struktur der beschriebenen Zeichenfolgen auch die Struktur der Parameterwerte widerspiegeln können (z.B. zum Durchsuchen einer Symboltabelle). Eine klare Trennung der beiden Ebenen wird durch das Konzept der Prädikate unterstützt: Das sind diejenigen Symbole der zugrundeliegenden kontextfreien Grammatik, die nur die leere Zeichenfolge beschreiben, so daß in den zugehörigen Regelmustern „Berechnungen“ isoliert werden können. Die Sichtweise, diese Regelmuster als Hornformeln anzusehen, eröffnet nun die Möglichkeiten der logischen Programmierung. Folglich lassen sich durch Zweistufengrammatiken alle aufzählbaren Sprachen beschreiben. Allerdings kann der Kalkül ebenso mißbraucht werden, indem eine Programmiersprache durch einen Compiler definiert wird, der dazu als logisches Programm vorliegt.

Zwar bieten Zweistufengrammatiken für die Definition von Programmiersprachen eine hervorragende Fundierung, dafür ist aber der Übergang zu brauchbaren Implementierungen beliebig unbestimmt. So ist ein naives Verschränken der zweistufigen Aufzählung von Regeln und von Zeichenfolgen hoffnungslos ineffizient, und zudem kann die Termination aus theoretischen Gründen unmöglich garantiert werden. Andererseits stehen aus der langjährigen Entwicklung des Übersetzerbaus bewährte systematische Verfahren sowohl für die klassische Syntaxanalyse als auch für die Attributauswertung zur Verfügung. Um davon profitieren zu können, ist das einfachste Mittel eine Transformation von Zweistufengrammatiken in entsprechende Attributgrammatiken. Dazu lassen sich im Formalismus der Erweiterten Affixgrammatiken analyseorientiert eingeschränkte Zweistufengrammatiken angeben, wobei zur Vereinfachung der Transformation für jeden Parameter explizit festgelegt wird, ob er als Eingabe- oder Ausgabeparameter zu behandeln ist. Die Einschränkung, daß auf den Parameterpositionen in Regelmustern nur Satzformen erlaubt sind, soll zusätzlich die zu generierenden Attributauswertungsregeln vereinfachen, indem dadurch die Syntaxanalyse der Parameterwerte in die Transformation vorgezogen werden kann.

Eine Transformation von Erweiterten Affixgrammatiken in Attributgrammatiken ist konzeptionell nicht weiter schwierig, wenn die zwangsläufig dabei auftretenden problematischen Fälle einfach ausgegrenzt werden. Statt der logischen Programmierung werden so nur funktionale Abhängigkeiten zwischen Eingabe- und Ausgabeparametern unterstützt, da das erforderliche Backtracking über einzelne Attributauswertungsregeln hinaus nicht ins Konzept paßt. Weiterhin stellt eine Attributgrammatik für die ursprüngliche Erweiterte Affixgrammatik noch eine, womöglich unverträgliche Sprachbeschreibung dar. Um nun bei der Compilergenerierung auf der Basis Erweiterter Affixgrammatiken von der Transformation und überhaupt von Attributgrammatiken abstrahieren zu können, müssen also gewisse Wohlgeformtheitsbedingungen eingehalten werden. Mächtige hinreichende Bedingungen dafür, daß die beschriebene Sprache unter der Transformation erhalten bleibt, hängen aber mit unentscheidbaren Problemen (insbesondere Eindeutigkeit und Termination) zusammen und können daher automatisch nicht direkt überprüft werden.

Auf der Grundlage, Wohlgeformtheitsbedingungen erstmals im obigen Sinne aufzufassen, wurde die Entwicklung des Eta-Compilergenerators an der Technischen Universität Berlin entscheidend

begünstigt durch die Wahl sehr mächtiger Bedingungen samt der Konsequenz, daß deren Einhaltung in der Verantwortung des Benutzers liegen soll. Weiterhin wurde die Transformation durch die Einführung einer Zwischensprache vereinfacht, wofür eine leicht verständliche Normalform Erweiterter Affixgrammatiken definiert wurde. Allerdings genügt es nicht, nur die Transformation zu implementieren, da weder allgemein anerkannte Generatorsysteme für Attributgrammatiken verfügbar sind, noch auf diese Art Fehler in Terminologie der Erweiterten Affixgrammatiken gemeldet werden können. Mit der vollständigen Implementierung eines Kernsystems aber konnte anhand verschiedener Spezifikationen für Untermengen von Programmiersprachen demonstriert werden, daß die geforderten Einschränkungen oft nur kuriose Formulierungen verbieten und daß die Kennzeichnung des Informationsflusses durch Richtungen der Parameter überaus hilfreich ist. In einer anschließenden Wachstumsphase wurde dann durch Erweiterung des modular strukturierten Prototyps um die stärker einschränkenden Verfahren zur Generierung effizienterer Compiler entsprechendes Know-how gesammelt. Die Hinterlassenschaft war schließlich ein zu großes System auf aussterbender Hardware.

Eine Portierung ausgewählter Komponenten ist jedoch in dieser Situation unangebracht, da wesentliche Effizienzeinbußen gerade darauf zurückzuführen sind, daß bereits die Schnittstellen des Eta-Generatorsystems auf den zu simplen normierten Erweiterten Affixgrammatiken beruhen. Daneben wurde erkannt, wie Defizite im Ausdruckskomfort behoben werden können, indem z.B. die oben angeführte Ungleichheit durch eine besondere Notation formuliert wird, die als Abkürzung verstanden werden kann und so die Reinheit des Kalküls bewahrt. Eine erforderliche Neukonzeption führte im Rahmen der Gruppendiplomarbeit zweier der Autoren (J. Demuth und S. Weber) zu einem Kernsystem zur Generierung praxistauglicher Compiler, das im vorliegenden Bericht vorgestellt wird. Wir hoffen nun auf diesem Weg, den vielversprechenden Ansatz, der an der Technischen Universität Berlin aufgrund von Personalknappheit und Desinteresse keine Zukunft hat, einer breiteren Öffentlichkeit zugänglich zu machen.

Beeinflußt von der Arbeit mit dem Oberon-System von Prof. Wirth, schließen wir uns seiner Kritik an den Dimensionen üblicher Softwaresysteme an. So folgen wir seinem Beispiel und drucken die Quelltexte unserer Implementierung kommentiert und vollständig im Original mit ab. Wir halten dies für unerläßlich, um eine Einschätzung der Kompliziertheit eines Systems und damit seiner Angemessenheit zu ermöglichen; auch eine Diskussion unterschiedlicher Programmiertechniken kann erst aufgrund solcher Veröffentlichungen stattfinden. Dieses Vorgehen wirkt sich fraglos auch stark auf die Form der Implementierung aus. Zum einen ist eine gut lesbare Programmiersprache erforderlich, zum anderen müssen die verwendeten Algorithmen knapp und geradlinig umgesetzt werden. Als Folge können die üblicherweise präsentierten „abstrakten Algorithmen“ entfallen, die ohnehin die bestehenden Optimierungsmöglichkeiten größtenteils verdecken würden. Aus den genannten Gründen betrachten wir die Quelltexte als wesentlichen Aspekt der Veröffentlichung.

Die Dokumentation beginnt mit einer kurzen Kritik des herkömmlichen Compilerbaus und einer Einführung in den der Implementierung zugrundeliegenden Kalkül. In Kapitel 2 wird die Neukonzeption unseres Compilergenerators *Epsilon* im Vergleich zu Eta vorgestellt. Das Kapitel 3 leitet die Beschreibung der Implementierung mit einem Überblick ein, in den nachfolgenden Kapiteln 4, 5, 6 und 7 werden schließlich die einzelnen Module im Detail behandelt, gegliedert nach den Hauptaufgaben Einlesen einer Spezifikation, Generierung von Scanner und Parser sowie zweier alternativer Evaluatoren. Kapitel 8 bietet einen Ausblick.

Im Anhang wird die Spezifikationssprache von Epsilon und ein reduziertes Beispiel vorgestellt; zusätzliche größere Beispiele sollen die Ausdrucksmöglichkeiten verdeutlichen; anschließend wird die Bedienung des Generators erklärt. Eine Demonstration der Güte der generierten Fehlerbehandlung beschließt den Anhang.

Berlin, im März1997

Jochen Demuth, Stephan Weber, Sönke Kannapinn, Mario Kröplin

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Die Entwicklung des Compilerbaus in den letzten Jahrzehnten führte zu einer Standard-Modularisierung von Übersetzern und zu Methoden zur Realisierung dieser Module. Teilweise werden sogar „Tools“ zur Generierung einzelner Module verwendet, diese sind aber uneinheitlich und in komplizierteren Gebieten (Kontextbedingungen, Codeerzeugung) ohne intime Detailkenntnisse nur schwer verständlich. Letztlich ist die Automatisierung des „Übersetzerbaus von Hand“ zumindest fragwürdig.

Ein systematischerer Ansatz für eine Generierung von Übersetzern ist die formale Beschreibung einer Programmiersprache mit Hilfe *eines* Kalküls. Als Beispiel hierfür seien die van-Wijngaarden-Grammatiken genannt, eine Form zweistufiger Grammatiken. Eine solche Grammatik wurde erstmals zur Definition der Kontextbedingungen der Sprache Algol 68 eingesetzt; dieser Kalkül eignet sich jedoch nicht für die Generierung entsprechender Compiler. Im Gegensatz dazu sind die bekannten Attributgrammatiken implementierungsnäher; sie stellen einen offenen Kalkül dar, indem sie die Verwendung einer weiteren Spezifikations- bzw. Programmiersprache erfordern. Dadurch sind sie zum einen schwieriger zu verstehen, zum anderen führte diese Offenheit zu einer Vielzahl unterschiedlicher Realisierungen.

Die von Watt vorgeschlagenen analyseorientierten *Erweiterten Affixgrammatiken* vereinen die Vorteile beider Grammatikklassen [Watt]; sie bieten als geschlossener Kalkül die Möglichkeit der einheitlichen Beschreibung von Programmiersprachen mit ihren Übersetzungen und bilden die Grundlage unseres Compilergenerators.

Die prinzipielle Tauglichkeit dieses Kalküls für die automatische Erstellung von Übersetzern wurde bereits durch den 1984 entstandenen Compilergenerator Eta demonstriert [Schröer], der seither mit vielen Erweiterungen im Rahmen von Lehrveranstaltungen eingesetzt wurde. Die Wahl der Programmiersprachen zur Implementierung auf einem Großrechner sowie die schiere Größe des Systems machten eine Wartung oder Portierung jedoch nahezu unmöglich. So wurde nur ein Kernsystem auf moderne UNIX-Rechner übertragen; außerdem blockiert die aus heutiger Sicht veraltete Konzeption wesentliche Optimierungen.

Mit *Epsilon* sollte nun ein *kleines* Experimentalsystem mit größerer Flexibilität und Beherrschbarkeit von Grund auf neu konzipiert und implementiert werden, das die Generierung *effizienter* Compiler aus einer einheitlichen, formalen Spezifikation ermöglicht.

## 1.2 Erweiterte Affixgrammatiken

Im folgenden wird vorausgesetzt, daß der Leser mit dem Prinzip zweistufiger Grammatiken bereits vertraut ist, da die später benötigten Begriffe hier nur informell eingeführt werden. Im Vordergrund steht die textuelle Benennung wesentlicher Inhalte zur Klärung der begrifflichen Ebene sowie die Einführung eines Prozedurmodells als operationelle Semantik, die für das Verständnis der Implementierung ausreichend und insbesondere hilfreich ist. Eine formale Definition von EAGen findet sich beispielsweise in [Schröer].

Eine *Erweiterte Affixgrammatik*, kurz *EAG* genannt, ist eine zweistufige Grammatik: Sie besteht aus einer kontextfreien Grammatik, deren Nichtterminale um Parameter angereichert sind (der *Hyper-Grammatik*), und einer Menge von kontextfreien Regeln (mit eigenen Terminalen und Nichtterminalen) zur Beschreibung der Parameterwerte (der *Meta-Grammatik*).

Für jeden Parameter eines Hyper-Nichtterminals ist eine Richtung (Eingabe, Ausgabe) und durch Angabe eines Meta-Nichtterminals (des *Wertebereichssymbols*) ein Typ festgelegt; die Folge der Richtungen und Wertebereichssymbole aller Parameter eines Hyper-Nichtterminals wird als dessen *Signatur* bezeichnet. In den Hyper-Regeln steht auf jeder Parameterposition eine *Affixform* zum jeweiligen Wertebereichssymbol. Dies ist eine Satzform zu diesem Meta-Nichtterminal, in der jedes darin vorkommende Nichtterminal mit einer ihm eventuell nachgestellten Zahl zu einer sogenannten *Variablen* zusammengefaßt wird.

Streicht man aus der Hyper-Grammatik die Parametrisierung heraus, so ergibt sich eine gewöhnliche kontextfreie Grammatik, die *Skelettgrammatik*; diese ist für EAGen üblicherweise jedoch mehrdeutig, da bestimmte Berechnungen mit Hyper-Nichtterminalen (den *Prädikaten*) formuliert werden, die nur nach leer ableitbar sind (und dies im allgemeinen eben mehrdeutig), die aber folglich nichts zum kontextfreien Anteil der Quellsprache beitragen. Streicht man auch die Prädikate, so erhält man die *Grundgrammatik* der EAG; ist diese eindeutig, so kann nach klassischen Methoden der Syntaxanalyse ein Parser erstellt werden, der für korrekte Eingaben den zugehörigen *Ableitungsbaum* liefert.

Die große Ausdrucksmächtigkeit von EAGen resultiert aber aus der Parametrisierung der Hyper-Grammatik. Eine Implementierung kann dadurch vorgenommen werden, daß jedes Hyper-Nichtterminal durch eine *Prozedur* realisiert wird. Die Parameter der Nichtterminale werden in Prozedurparameter umgesetzt, die Richtung und der Typ werden dabei entsprechend übertragen. Jede Affixform beschreibt die syntaktische Struktur eines Parameters, die entsprechend der Richtung überprüft wird (*Analyse*) oder aufgebaut wird (*Synthese*). Gleiche Variablen in Affixformen einer Hyper-Regel stehen für gleiche Werte (*konsistente Ersetzung*). Dies muß in der Implementierung gegebenenfalls durch *Vergleiche* überprüft werden. Für die strukturellen Überprüfungen wird gefordert, daß die Meta-Grammatik zu einem Wertebereichssymbol, zu dem Analysen oder Vergleiche durchgeführt werden, eindeutig ist.

Die Berechnung der zur Grundgrammatik gehörenden Parameter erfolgt gemäß den angegebenen Richtungen anhand des Ableitungsbaums durch Analysen und Synthesen. Für Prädikate werden die Parameterwerte analog berechnet, jedoch wird dabei die Ableitung durch Backtracking bestimmt. Kontextfehler zeigen sich im Scheitern einer Analyse oder eines Vergleichs bzw. eines Prädikats.

Die *Sprache* einer EAG ist die Menge aller Wörter, die in der Sprache der Grundgrammatik enthalten sind, und für die die Parameterberechnung erfolgreich durchgeführt werden kann. Das Startsymbol jeder EAG besitzt keine Eingabe- und genau einen Ausgabeparameter. Am Ende der Berechnungen kann der Wert des Ausgabeparameters als *Übersetzung* der Eingabe verwendet werden; diese fällt also eher nebenbei ab und wird nicht gesondert unterstützt.

Löst man das vorgestellte Prozedurmodell auf, so können zur Parameterberechnung alle von Attributgrammatiken her bekannten Auswertungsverfahren zur Anwendung kommen. Eine EAG

kann dazu als Attributgrammatik betrachtet werden, deren Attribute entsprechend den Wertebereichssymbolen typisiert sind, und deren Berechnungsvorschriften durch die Strukturbäume der Affixformen sowie die Prädikate gegeben sind.

# Kapitel 2

# Revision und Neukonzeption

## 2.1 Bisherige Implementierungen

Bereits 1984 entstand an der TU-Berlin der Compilergenerator Eta [Schröer], der auf eine langjährige Benutzung im Rahmen von Lehrveranstaltungen zurückblicken kann.

Die von Eta generierten Compiler bestehen grundsätzlich aus drei hintereinander ablaufenden Phasen. In der ersten Phase wird ein Quelltext durch einen festen Scanner in eine Tokenfolge überführt. Ein generierter Parser überprüft diese Tokenfolge in der zweiten Phase auf syntaktische Korrektheit bezüglich der kontextfreien Grundgrammatik und erzeugt die Linksableitung des Quelltexts. Diese wird in der dritten Phase von einem Evaluator verarbeitet, der Kontextfehler erkennt und im fehlerfreien Fall die Übersetzung des Quellprogramms erstellt und ausgibt. Die Datenübergabe zwischen den Phasen erfolgt über Dateien.

Die Generierung eines Compilers erfolgt immer in vier Phasen. In der ersten Phase wird die in der Eingabesprache COLA spezifizierte EAG nach Überprüfung auf Korrektheit in eine normierte EAG überführt, anhand derer der Compiler generiert wird. In dieser kommen nichttriviale Affixformen höchstens in den durch die Normierung entstandenen sogenannten primitiven Prädikaten vor. In der zweiten Phase wird ein Parser erzeugt. In der dritten und vierten Phase wird ein Evaluator generiert. Dazu wird in der dritten Phase Programmcode für die als Prädikate angegebenen Nichtterminale sowie für die primitiven Prädikate erzeugt. In der vierten Phase erfolgt die Generierung des Evaluators unter Verwendung des bereits generierten Prädikatcodes.

## 2.2 Erhöhung des Ausdruckskomforts

Unter Eta werden EAGen in der Spezifikationssprache COLA beschrieben. Für die Neuimplementierung wurden Erweiterungen an dieser Sprache vorgenommen mit dem Ziel, den Ausdruckskomfort zu erhöhen. Diese Spracherweiterungen stellen Abkürzungsmechanismen dar, bieten aber auch Hinweise für Optimierungen in der Implementierung. Insbesondere wird der Kalkül der EAGen nicht verlassen.

Um in COLA analog zur konsistenten Ersetzung die Ungleichheit von Variablenwerten fordern zu können, ist die Ausformulierung von unequal-Prädikaten notwendig. Die Vermeidung der umfangreichen naiven Formulierung führt dabei zu einer komplizierten Spezifikation. Zusätzlich wird in der Implementierung ein Großteil der Ressourcen verbraucht, was zu ineffizienten Compilern führt. Unter der Voraussetzung, daß die Meta-Grammatik eindeutig ist, läßt sich ein wohlgeformtes unequal-Prädikat aber auch automatisch erzeugen. Die Implementierung dieses Prädikats

erfolgt jedoch günstiger als Negation des primitiven equal-Prädikats. Als neues Ausdrucksmittel stellen wir in Epsilon daher das Ungleichheitszeichen „#“ zur Verfügung. Wird es dem Bezeichner einer Variablen vorangestellt, so bezeichnet dieser Ausdruck eine neue Variable, die mit der ursprünglichen durch das unequal-Prädikat verbunden ist. Auf definierenden Positionen müssen daher verschiedene Vorkommen der neuen Variablen gleiche Werte erhalten. Zur Vereinfachung und mit Blick auf die praktische Relevanz darf nicht mehr als ein unequal-Operator auf einen Bezeichner angewendet werden. Komplemente von Affixformen sowie die allgemeine Negation werden nicht unterstützt.

Die kontextfreie Struktur von Programmiersprachen wird in Sprachreporten üblicherweise in EBNF-Notation angegeben. Um diese als Grundlage für eine Spezifikation übernehmen zu können, werden in Epsilon die EBNF-Operatoren für Alternativen, Optionen und Wiederholungen eingeführt. Ihre Bedeutung wird durch die Angabe einer Transformation in kontextfreie Regeln festgelegt. Dabei erweist sich die direkte Umsetzung jeder Gruppierung in ein neues Nichtterminal als günstig.

Auf Hyper-Regeln wird eine solche Transformation übertragen, indem die Parametrisierung der sogenannten anonymen Nichtterminale zusätzlich beschrieben wird. Dies ist Gegenstand der Tabellen 2.1 und 2.2. In diesen werden die Parameter der linken Seite durch die $\pi_i$, die Parameter der Nichtterminale der rechten Seite durch die $\overline{\pi}_i$ angedeutet.

Kommt ein EBNF-Operator im nichttrivialen Kontext vor, so wird ein anonymes Nichtterminal eingeführt, wie aus Tabelle 2.1 ersichtlich ist. Tritt in der ursprünglichen Regel ein EBNF-Operator ohne Kontext auf, so sprechen wir von einem benannten EBNF-Operator. Dieser wird dann gemäß Tabelle 2.2 transformiert.

| Regel mit EBNF-Operator | Transformierte Regel |
|---|---|
| $A \to \alpha\ \overline{\pi}_0(\pi_1\beta_1 \mid \ldots \mid \pi_n\beta_n)\ \gamma$ | $A \to \alpha\ B\overline{\pi}_0\ \gamma,$<br>$B\pi_1 \to \beta_1, \ldots, B\pi_n \to \beta_n$ |
| $A \to \alpha\ \overline{\pi}_0[\pi_1\beta_1 \mid \ldots \mid \pi_n\beta_n]\pi_{n+1}\ \gamma$ | $B\pi_1 \to \beta_1, \ldots, B\pi_n \to \beta_n,$<br>$B\pi_{n+1} \to \epsilon$<br>$A \to \alpha\ B\overline{\pi}_0\ \gamma,$ |
| $A \to \alpha\ \overline{\pi}_0\{\pi_1\beta_1\overline{\pi}_1 \mid \ldots \mid \pi_n\beta_n\overline{\pi}_n\}\pi_{n+1}\ \gamma$ | $B\pi_1 \to \beta_1 B\overline{\pi}_1, \ldots, B\pi_n \to \beta_n B\overline{\pi}_n,$<br>$B\pi_{n+1} \to \epsilon$ |

Tabelle 2.1: Transformation der EBNF-Operatoren mit Kontext

| Regel mit EBNF-Operator | Transformierte Regel |
|---|---|
| $A \to \pi_1\alpha_1 \mid \ldots \mid \pi_n\alpha_n =$<br>$A \to (\pi_1\alpha_1 \mid \ldots \mid \pi_n\alpha_n)$ | $A\pi_1 \to \alpha_1, \ldots, A\pi_n \to \alpha_n$ |
| $A \to [\pi_1\alpha_1 \mid \ldots \mid \pi_n\alpha_n]\pi_{n+1}$ | $A\pi_1 \to \alpha_1, \ldots, A\pi_n \to \alpha_n,$<br>$A\pi_{n+1} \to \epsilon$ |
| $A \to \{\pi_1\alpha_1\overline{\pi}_1 \mid \ldots \mid \pi_n\alpha_n\overline{\pi}_n\}\pi_{n+1}$ | $A\pi_1 \to \alpha_1 A\overline{\pi}_1, \ldots, A\pi_n \to \alpha_n A\overline{\pi}_n,$<br>$A\pi_{n+1} \to \epsilon$ |

Tabelle 2.2: Transformation der EBNF-Operatoren ohne Kontext

Im Gegensatz zu COLA lassen sich nun aber für anonyme Nichtterminale die Signaturen, d.h. die Richtungen und Wertebereichssymbole der Parameter, nicht mehr einem Bezeichner zuordnen, weshalb diese Eigenschaften zusammen mit der Parametrisierung innerhalb der Regeln angegeben werden. Jedoch werden solche formalen Parameterlisten nur dann verwendet, wenn sie

in den transformierten Regeln zu linken Seiten gehören. Somit werden diese Eigenschaften im allgemeinen mehrfach angegeben. Diese Festlegung wird aus Konsistenzgründen auf alle Nichtterminale ausgeweitet. Ein besonderer Abschnitt der Spezifikation, in dem die Signaturen der Hyper-Nichtterminale aufgeführt werden, ist nun nicht mehr erforderlich. Analog zu Programmiersprachen ohne FORWARD-Deklarationen ist dann allerdings die Überprüfung der Affixformen nicht bereits beim Einlesen der Spezifikation möglich.

Weiterhin können anonyme Nichtterminale, die als Prädikate besonders zu behandeln sind, nicht in einem Abschnitt der Spezifikation aufgeführt werden. Zudem ist überhaupt keine explizite Auszeichnung der Prädikate ihrer umstrittenen Rolle angemessen. Eine Bestimmung der Prädikate kann durch den Generator erfolgen, wobei verschiedene Implementierungen denkbar sind.

In Meta-Regeln ist die Verwendung der EBNF-Operatoren nicht sinnvoll, da ein induktiver Auf- und Abbau der Affixe mit anonymen Meta-Nichtterminalen nicht formuliert werden kann. Somit erfolgt in Meta-Regeln eine Beschränkung auf den Operator „|“ zur Zusammenfassung von Alternativen.

Die Trennung von Meta- und Hyper-Regeln wird ebenfalls aufgehoben. Zur Unterscheidung von Meta- und Hyper-Regeln werden unterschiedliche Zeichen zur Trennung der linken und rechten Regelseiten verwendet.

## 2.3 Neukonzeption des Compilergenerators

Die strikte Trennung der drei Phasen in den von Eta generierten Compilern soll aufgehoben werden. Verbleibende Datenübergaben vom Parser zum Evaluator sollen effizient im Speicher erfolgen. Eine Verschränkung der lexikalischen und der syntaktischen Analyse wird in Epsilon dadurch realisiert, daß die Zerteilung des Quelltextes nicht in einem Zuge, sondern tokenweise erfolgt.

Unter der Voraussetzung, daß die gemäß Abschnitt 2.2 transformierten Regeln der Hyper-Grammatik linksdefinierend sind, können auch Parser und Evaluator verschränkt werden. In diesem Fall lassen sich dann also echte Ein-Pass-Compiler nach dem Vorbild effizienter Compiler generieren, die auf rekursiven Abstiegsparsern basieren. Der Code zur Berechnung der Parameterwerte wird dazu in die Prozeduren des Parsers eingefügt.

Eine separate Evaluation wird anhand eines Ableitungsbaums durchgeführt, der vom Parser im Speicher aufgebaut wird. Die Ausgabe einer Linksableitung ist nicht mehr erforderlich.

Die Berechnung der Parameterwerte wird nicht mehr wie bei Eta durch primitive Prädikate vorgenommen, sondern erfolgt direkt in den für die Nichtterminale generierten Prozeduren. Dadurch lassen sich viele Prozeduraufrufe einsparen und weitergehende Optimierungen durchführen.

Implementiert wird die lexikalische Analyse ähnlich wie unter Eta durch einen festen, parametrisierten Scanner. Um Ein-Pass-Compiler generieren zu können, erfolgt die syntaktische Analyse nach dem LL(1)-Verfahren. Zur effizienten Umsetzung von Wiederholungen in Schleifen muß dabei die LEAG-Bedingung eingeschränkt werden. Als Beispiel eines mächtigeren Auswertungsverfahrens wird ein separater single sweep-Evaluator angeboten.

Für die Generierung ist die Normierung einer EAG und die Einführung von primitiven Prädikaten eher hinderlich, da dabei die Struktur der Affixformen verlorengeht. Die Datenübergabe zwischen den Phasen im Generator erfolgt ebenfalls effizient, indem eine Internalisierung der Spezifikation permanent im Speicher vorliegt.

# Kapitel 3

# Allgemeines zur Implementierung

## 3.1 Die Implementierungssprache

Für die Implementierung des Epsilon-Compilergenerators sollte eine einfache imperative Programmiersprache gewählt werden, um die wesentlichen Algorithmen einerseits effizient und andererseits gut lesbar formulieren zu können. Derzeit erscheint uns sie Sprache *Oberon*, die auch im Hinblick auf die Veröffentlichung von umfangreichem Quellcode als Nachfolger von Pascal und Modula angepriesen wird [ReiWi], als am besten geeignet.

Ein weiterer Grund für die Wahl von Oberon ist das zugehörige gleichnamige Betriebssystem, das eine einfache Fensteroberfläche besitzt, deren Verwendung in Epsilon nahezu keinen Programmieraufwand erfordert. Das Oberon-System wurde mittlerweile auf viele Plattformen portiert und wird von der ETH Zürich kostenlos zur Verfügung gestellt [WiGu] [Reiser].

Auch langfristig scheint Oberon eine vernünftige Wahl zu sein, da der geringe Sprachumfang und die strenge Typprüfung eine eventuell später nötige Übertragung von Epsilon in eine andere imperative Sprache als verhältnismäßig einfach erscheinen lassen. Als Generatorzielsprache wurde ebenfalls Oberon gewählt, um das Oberon-System als einheitliche Arbeits- und Testumgebung verwenden zu können.

Im folgenden werden noch zwei Module vorgestellt, die grundlegende Basisfunktionen bereitstellen. Eines dient Epsilon als Schnittstelle zum Betriebssystem, im anderen sind beliebig große Mengen natürlicher Zahlen als abstrakter Datentyp implementiert.

### 3.1.1 Die Schnittstelle zum Betriebssystem

Das Modul `eIO` dient als Schnittstelle zum Betriebssystem, um für Portierungen auf andere Plattformen Änderungen auf dieses Modul zu beschränken. Konzeptionell ist es auf die (text-orientierte) Fensteroberfläche des Oberon-Systems ausgerichtet, jedoch sind dessen Eigenheiten so weit isoliert, daß beispielsweise auch eine Version für UNIX ohne Fensteroberfläche realisiert werden kann.

Das Modul bietet die von Epsilon und den generierten Compilern benötigten Funktionen zum sequentiellen Lesen und Schreiben von Texten und binären Dateien sowie zum Zugriff auf Parameter von Kommandos.

```
DEFINITION eIO;

  CONST eol = 0DX;

  TYPE
    TextIn = POINTER TO RECORD END;
    Position = RECORD END;
    TextOut = POINTER TO RECORD END;
    File = POINTER TO RECORD END;

  VAR
    Msg : TextOut;
    UndefPos : Position;

  PROCEDURE OpenIn(VAR In : TextIn; Name : ARRAY OF CHAR;
                   VAR Error : BOOLEAN);
  PROCEDURE CloseIn(VAR In : TextIn);
  PROCEDURE Read(In : TextIn; VAR c : CHAR);
  PROCEDURE Pos(In : TextIn; VAR Pos : Position);
  PROCEDURE PrevPos(In : TextIn; VAR Pos : Position);

  PROCEDURE CreateOut(VAR Out : TextOut; Name : ARRAY OF CHAR);
  PROCEDURE CreateModOut(VAR Out : TextOut; Name : ARRAY OF CHAR);
  PROCEDURE CloseOut(VAR Out : TextOut);
  PROCEDURE Write(Out : TextOut; c : CHAR);
  PROCEDURE WriteInt(Out : TextOut; i : LONGINT);
  PROCEDURE WriteIntF(Out : TextOut; i : LONGINT;
                      Len : INTEGER);
  PROCEDURE WriteString(Out : TextOut; Str : ARRAY OF CHAR);
  PROCEDURE WriteText(Out : TextOut; Str : ARRAY OF CHAR);
  PROCEDURE WriteLn(Out : TextOut);
  PROCEDURE WritePos(Out : TextOut; Pos : Position);
  PROCEDURE Show(Out : TextOut);
  PROCEDURE Update(Out : TextOut);
  PROCEDURE Compile(Out : TextOut; VAR Error : BOOLEAN);

  PROCEDURE OpenFile(VAR F : File; Name : ARRAY OF CHAR;
                     VAR Error : BOOLEAN);
  PROCEDURE CreateFile(VAR F : File; Name : ARRAY OF CHAR);
  PROCEDURE CloseFile(VAR F : File);
  PROCEDURE GetLInt(F : File; VAR i : LONGINT);
  PROCEDURE GetSet(F : File; VAR s : SET);
  PROCEDURE PutLInt(F : File; i : LONGINT);
  PROCEDURE PutSet(F : File; s : SET);

  PROCEDURE InputName(VAR Name : ARRAY OF CHAR);
  PROCEDURE IsOption(c1 : CHAR) : BOOLEAN;
  PROCEDURE IsLongOption(c1, c2 : CHAR) : BOOLEAN;
  PROCEDURE NumOption(VAR Num : LONGINT);
  PROCEDURE StringOption(VAR Str : ARRAY OF CHAR);

  PROCEDURE TimeStamp() : LONGINT;

END eIO.
```

Für die Eingabe stehen Prozeduren zum Öffnen und Schließen eines Textes, zum sequentiellen Lesen einzelner Zeichen und zur Positionsbestimmung zur Verfügung. Die Kapselung von Positionen erfordert zusätzlich eine Prozedur zum Zugriff auf die unmittelbar vorhergehende Position sowie eine (konstante) Variable `UndefPos`, die als Initialisierung verwendet werden kann.

Für die Ausgabe gibt es Prozeduren zum Anlegen und Schließen von Texten; die Prozedur `CreateModOut` fügt dem Textnamen noch eine zum verwendeten Compiler passende Endung hinzu. Ferner können Zeichen, Zahlen (eventuell formatiert), Zeichenketten, Zeilenenden und Positionen geschrieben werden. Um das Schreiben von Sonderzeichen zu erleichtern, werden in der Prozedur `WriteText` während der Ausgabe ähnlich wie in der Programmiersprache C die Sequenzen `\t`, `\n`, `\'` und `\\` in einen Tabulator, ein Zeilenende, ein doppeltes Anführungszeichen bzw. den Backslash umgesetzt. Um einen Text anzuzeigen, ist `Show` aufzurufen; Anfügungen an einen sichtbaren Text werden spätestens nach Aufruf von `Update` angezeigt. Der exportierte Text `Msg` wird standardmäßig angezeigt und kann für Meldungen verwendet werden. Ist ein Text ein Oberon-Modul, so kann dies mit `Compile` übersetzt werden; dabei werden mittels der unten beschriebenen Prozedur `StringOption` eventuell angegebene Compileroptionen ermittelt und gegebenfalls dem Compiler übergeben.

Ein Zugriff auf binäre Dateien wird durch Prozeduren zum Öffnen, Anlegen und Schließen von Dateien sowie Prozeduren zum Lesen und Schreiben für die Datentypen `LONGINT` und `SET` ermöglicht.

Die Datentypen `TextIn`, `TextOut` und `File` sind als Zeiger konzipiert, so daß Variablen dieser Typen durch einfache Zuweisungen kopiert werden können und anschließend Original und Kopien gemischt in den entsprechenden Prozeduraufrufen verwendet werden können.

Mittels der Prozeduren `InputName`, `IsOption`, `IsLongOption`, `NumOption` und `StringOption` können Kommandoparameter abgefragt werden. Zwischenraum (Leerzeichen, Tabulatoren, etc.) trennt diese voneinander und ist daher innerhalb von Parametern nicht erlaubt. Parameter, die mit einem Bindestrich oder einem Backslash beginnen, geben Optionen an. Hinter diesem ersten Zeichen darf eine Buchstabenfolge, eine Ganzzahl oder eine in doppelte Anführungszeichen eingeschlossene Zeichenkette stehen. `IsOption` liefert `TRUE`, wenn in einer Buchstabenfolge der angegebene Kleinbuchstabe ohne nachfolgenden Großbuchstaben auftritt, `IsLongOption` liefert `TRUE`, wenn der angegebene Kleinbuchstabe gefolgt vom angegebenen Großbuchstaben auftritt. `NumOption` liefert die erste als Option angegebene Zahl; ist keine angegeben, wird der Wert Null zurückgegeben. `StringOption` liefert die erste als Option angegebene Zeichenkette ohne die Anführungszeichen; ist keine angegeben, wird eine leere Zeichenfolge zurückgegeben. Der erste Parameter, der keine Optionen angibt, wird als Eingabename interpretiert. Hinter dem Eingabenamen stehende Parameter werden ignoriert.

Die Prozedur `TimeStamp` gibt idealerweise bei jedem Aufruf einen neuen Wert zurück, der zur Überprüfung der Zusammengehörigkeit von erstellten Compilern und Steuerdateien verwendet wird.

Auf den Abdruck dieses Moduls wird verzichtet, da Implementierungen systemspezifisch sind.

### 3.1.2 Mengen

Das Modul `eSets` stellt Mengen von natürlichen Zahlen mit üblichen Operationen darauf als abstrakten Datentyp zur Verfügung.

```
DEFINITION eSets;

  TYPE
    OpenSet = POINTER TO ARRAY OF SET;

  PROCEDURE New(VAR s0 : OpenSet; MaxElem : INTEGER);

  PROCEDURE Empty(VAR s0 : OpenSet);
  PROCEDURE Incl(VAR s0 : OpenSet; n : INTEGER);
  PROCEDURE Excl(VAR s0 : OpenSet; n : INTEGER);
  PROCEDURE Assign(VAR s0 : OpenSet; s1 : OpenSet);
  PROCEDURE Union(VAR s0 : OpenSet; s1, s2 : OpenSet);
  PROCEDURE Intersection(VAR s0 : OpenSet; s1, s2 : OpenSet);
  PROCEDURE Difference(VAR s0 : OpenSet; s1, s2 : OpenSet);
  PROCEDURE SymmetricDifference(VAR s0 : OpenSet; s1, s2 : OpenSet);
  PROCEDURE Complement(VAR s0 : OpenSet; s1 : OpenSet);

  PROCEDURE IsEmpty(s1 : OpenSet) : BOOLEAN;
  PROCEDURE Equal(s1, s2 : OpenSet) : BOOLEAN;
  PROCEDURE Disjoint(s1, s2 : OpenSet) : BOOLEAN;
  PROCEDURE Included(s1, s2 : OpenSet) : BOOLEAN;
  PROCEDURE In(s1 : OpenSet; n : INTEGER) : BOOLEAN;

  PROCEDURE nSetsUsed(s1 : OpenSet) : INTEGER;
  PROCEDURE ConvertToSET(s1 : OpenSet; Index : INTEGER) : SET;

END eSets.
```

Mit der Prozedur `New` wird eine neue, anfangs leere Menge dynamisch angelegt. In diese Menge dürfen Zahlen im Bereich `[0 ... MaxElem]` eingetragen werden. Den Prozeduren, die mehrere Mengen als Parameter erwarten, dürfen nur Mengen gleicher Größe übergeben werden. `Empty` löscht alle Elemente aus einer Menge, mit `Assign` wird die zweite Menge in die erste Menge kopiert. Auch bei den weiteren Prozeduren, die die Operationen auf dem Basistyp `SET` auf beliebig große Mengen fortsetzen, stellt der erste Parameter das Ziel der Mengenoperation dar. Neben den üblichen Mengenoperationen stehen einige Funktionsprozeduren zur Abfrage von Eigenschaften zur Verfügung.

Die Prozeduren `nSetsUsed` und `ConvertToSET` dienen der Umwandlung einer Menge des Typs `OpenSet` in eine Folge von Mengen des Basistyps `SET`. Die Prozedur `nSetsUsed` gibt die Anzahl der benötigten Mengen zurück, die Prozedur `ConvertToSET` liefert die im zweiten Parameter angegebene Menge zurück. Die erste Menge (mit Index Null) enthält dabei die Elemente des Bereichs `[0 ... MAX(SET)]`, die zweite die Elemente aus `[MAX(SET) + 1 ... 2 * MAX(SET) + 1]` usw.

## 3.2 Überblick über Epsilon

Der Epsilon-Compilergenerator ist ein modulares System, dessen Komponenten in Abbildung 3.1 in einer Modulhierarchie dargestellt sind. Die Pfeile darin geben die wesentlichen Importbeziehungen an. Da das Modul EAG von allen anderen Modulen importiert wird, sind diese Beziehungen nur angedeutet.

Eine Compilerspezifikation wird vom `Analyser` unter Benutzung eines `Scanners` für die anschließende Generierung eingelesen und in entsprechenden Datenstrukturen des Moduls `EAG` abgelegt.

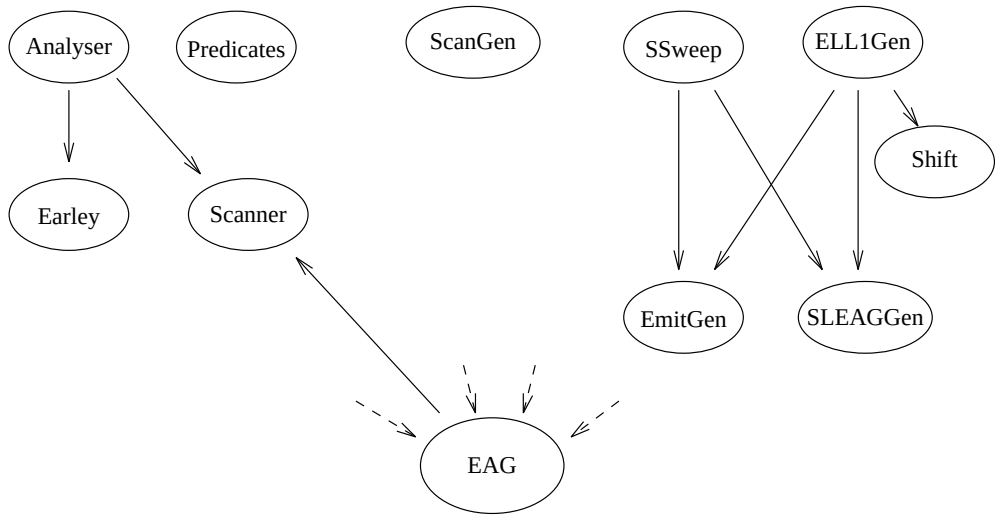



Abbildung 3.1: Modulhierarchie von Epsilon

Dabei wird ein erweiterter `Earley`-Parser verwendet, um für die auf Parameterpositionen angegebenen Affixformen Ableitungsbäume zu erstellen. `Predicates` exportiert ein Kommando zur Bestimmung der Prädikate der Spezifikation.

Als erster Schritt der Generierung kann nun das Modul `ScanGen` zur Erstellung eines Scanners verwendet werden. Das Modul `ELL1Gen` ist die Implementierung eines ELL(1)-Parsergenerators, mit dem anschließend echte Ein-Pass-Compiler erzeugt werden können. Dazu werden vom Modul `SLEAGGen` exportierte Prozeduren verwendet, um den entsprechenden Evaluationscode in den Parsercode einzubetten. Das Modul `EmitGen` dient der Erzeugung von Ausgabeprozeduren für den so generierten Compiler.

Unter Benutzung des Moduls `Shift` kann der Parsergenerator auch zur Erstellung eines Parsers verwendet werden, der nur einen statischen Ableitungsbaum aufbaut. Ein solcher Parser bildet zusammen mit einem vom Modul `SSweep` generierten Evaluator einen Compiler, der die Parameterberechnung nach dem single sweep-Verfahren vornimmt.

Um Namenskonflikte im Oberon-System zu vermeiden, wurde in der Implementierung allen Modulnamen der Buchstabe „e“ vorangestellt.

## 3.3 Allgemein verwendete Programmiertechniken

In diesem Abschnitt werden spezielle Programmiertechniken und Konventionen vorgestellt, die allen Modulen gemeinsam sind.

Eine wesentliche und bis auf wenige Ausnahmen durchgehaltene Programmiertechnik ist der Verzicht auf die dynamische Allokation einzelner benötigter Datenobjekte; im allgemeinen werden Objekte eines Typs stattdessen in einem großen Feld zusammengefaßt, was wesentlich speicher- und laufzeiteffizienter ist. Außerdem kann so jedes Objekt durch seinen Index bezeichnet werden, der sich zur einfachen Handhabung in Mengen eignet. Eine Datentyperweiterung (insbesondere in anderen Modulen) ist nun einfach und wieder speichereffizient durch Anlegen von Parallelfeldern möglich. Für jedes Feld *`A`* existiert eine Konstante `firstA`, die auf den ersten verwendeten Eintrag zeigt, und eine Variable `NextA`, die auf den ersten noch freien Eintrag verweist.

Erkauft werden die oben genannten Vorteile allerdings mit dem Verlust der Typsicherheit.

Um Größenbeschränkungen zu vermeiden, wird jedes Feld dynamisch alloziert und Zugriffe erfolgen über eine globale Zeigervariable. Vor Eintragungen erfolgt stets ein Test auf Überlauf des Feldes, der durch Aufruf einer Prozedur `Expand` behandelt wird, in der ein entsprechendes Feld größerer Länge alloziert, die bisherigen Einträge umkopiert und der Verweis auf das neue Feld in die globale Zeigervariable eingetragen wird. In jedem Modul gibt es nur eine solche `Expand`-Prozedur, die eine Vergrößerung für alle verwendeten Felder durchführen kann.

In Verweisen auf in solchen Feldern gespeicherte Objekte wird die Konstante `nil` analog zum Zeigerwert `NIL` verwendet. Diese Konstante ist in der Regel als der Wert Null definiert; dies erlaubt, über das Vorzeichen Verweise auf zwei verschiedene Felder zu unterscheiden. Aus diesem Grund bleibt in den meisten Feldern der erste Eintrag ungenutzt, da ein Verweis auf diesen Eintrag als `nil` interpretiert werden würde.

# Kapitel 4

# Die Internalisierung

## 4.1 Die interne Darstellung der EAG

Das Modul `EAG` stellt dem gesamten Compilergenerator eine interne Darstellung der Spezifikation zur Verfügung. Dazu enthält dieses Modul offenliegende Datenstrukturen für den effizienten Zugriff auf Komponenten der Meta- und der Hyper-Grammatik. Desweiteren gibt es dazugehörige Konstruktorprozeduren zum Aufbau der internen Darstellung sowie einfache Ausgabeprozeduren für Meldungen in Terminologie der Spezifikation.

In Anlehnung an die Transformation in Abschnitt 2.2 werden für die interne Darstellung von Hyper-Regeln zwar anonyme Nichtterminale eingeführt, aber die EBNF-Konstrukte bleiben erhalten. Eine solche Zwischenform kann dann sowohl als EBNF-Grammatik als auch als transformierte Grammatik angesehen und behandelt werden.

Die Konstruktorprozeduren sind nach einem einheitlichen Schema aufgebaut und benannt. Eine Funktionsprozedur `Find`*`A`* liefert den Index eines entsprechenden Eintrags in einem Feld *`A`*. Dazu wird das Feld linear durchsucht; mit dem gesuchten Wert als Sentinel im nächsten freien Eintrag vereinfacht sich die Abbruchbedingung, und ebendieser Eintrag wird für neue Werte vervollständigt. Eine Prozedur `App`*`A`* erweitert ein Feld *`A`* in jedem Fall um einen neuen Eintrag. Ansonsten werden Komponenten von Datenstrukturen durch Prozeduren `New`... angelegt und eventuell eingebunden.

Als Beispiel zur Erläuterung der internen Darstellung dient die folgende einfache Spezifikation:

```
N = 'i' N | .
S <+ N: N>:
  <N> { <+ 'i' N: N> 'a' <N> } <+ : N>
  <N> { <- 'i' N: N> 'b' <N> } <- : N>.
```

### 4.1.1 Meta-Grammatik

Die Meta-Terminale, -Nichtterminale sowie -Regeln werden in den Feldern `MTerm`, `MNont` bzw. `MAlt` und `MembBuf` dargestellt (siehe Abbildung 4.1).

```
VAR
  MNont: POINTER TO ARRAY OF RECORD
    Id, MRule: INTEGER;
    IsToken: BOOLEAN
  END;

  MTerm: POINTER TO ARRAY OF RECORD
    Id: INTEGER
  END;

  MAlt: POINTER TO ARRAY OF RECORD
    Left, Right, Arity, Next: INTEGER
  END;
  MaxMArity: INTEGER;

  MembBuf: POINTER TO ARRAY OF INTEGER;

PROCEDURE AppMemb (Val: INTEGER);
PROCEDURE FindMNont (Id: INTEGER): INTEGER;
PROCEDURE FindMTerm (Id: INTEGER): INTEGER;
PROCEDURE NewMAlt (Sym, Right: INTEGER): INTEGER;
```

Dabei verweisen die Komponenten `Id` auf die textuellen Repräsentationen im Modul `Scanner`. Die Komponente `IsToken` besagt, daß das Nichtterminal in der Spezifikation mit dem Zeichen „`*`“ für die Ausgabe als Token markiert wurde. Die Komponente `MRule` verweist auf die (textuell) erste Alternative eines Nichtterminals; die jeweils nächste Alternative ist über die Komponente `Next` zu erreichen. Die rechte Seite einer Alternative wird durch einen Bereich im Feld `MembBuf` beschrieben, der mit dem Wert `nil` abgeschlossen ist. Meta-Terminale werden durch negative Zahlen, Meta-Nichtterminale durch positive Zahlen gekennzeichnet; der Betrag verweist dann in das Feld `MTerm` bzw. `MNont`. Für den Earley-Parser wird der Eintrag hinter dem Bereich verwendet, um auf die zugehörige Alternative (in `MAlt`) zu verweisen.

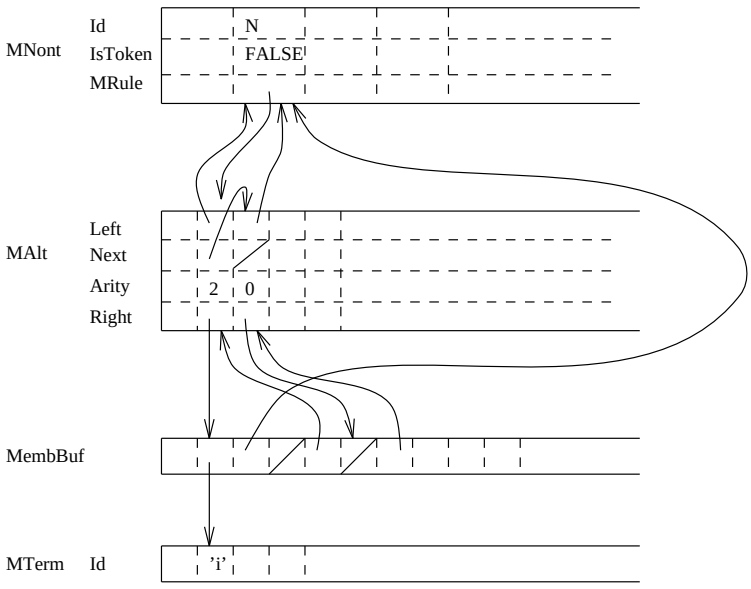



Abbildung 4.1: Interne Darstellung der Meta-Grammatik.

### 4.1.2 Hyper-Terminale und -Nichtterminale

```
TYPE
  Rule = POINTER TO RuleDesc;
  RuleDesc = RECORD Sub: Alt END;
  Grp = POINTER TO RECORD (RuleDesc) END;
  Opt = POINTER TO RECORD (RuleDesc)
    EmptyAltPos: eIO.Position;
    Scope: ScopeDesc;
    Formal: ParamsDesc
  END;
  Rep = POINTER TO RECORD (RuleDesc)
    EmptyAltPos: eIO.Position;
    Scope: ScopeDesc;
    Formal: ParamsDesc
  END;

VAR
  HNont: POINTER TO ARRAY OF RECORD
    Id, NamedId, Sig: INTEGER;
    Def: POINTER TO RuleDesc;
    IsToken: BOOLEAN
  END;
  StartSym: INTEGER;

  HTerm: POINTER TO ARRAY OF RECORD
    Id: INTEGER
  END;

  DomBuf: POINTER TO ARRAY OF INTEGER;

PROCEDURE FindHNont (Id: INTEGER): INTEGER;
PROCEDURE FindHTerm (Id: INTEGER): INTEGER;
PROCEDURE NewAnonymNont (Id: INTEGER): INTEGER;
PROCEDURE AppDom (Dir: CHAR; Dom: INTEGER);
PROCEDURE SigOK (Sym: INTEGER): BOOLEAN;
PROCEDURE WellMatched (Sig1, Sig2: INTEGER): BOOLEAN;
PROCEDURE NewGrp (Sym: INTEGER; Sub: Alt);
PROCEDURE NewOpt (Sym: INTEGER; Sub: Alt; Formal: ParamsDesc;
          Pos: eIO.Position);
PROCEDURE NewRep (Sym: INTEGER; Sub: Alt; Formal: ParamsDesc;
          Pos: eIO.Position);
```

Hyper-Nichtterminale werden durch einen eindeutigen Eintrag in dem Feld `HNont` repräsentiert. Ein positiver Eintrag in der Komponente `Id` kennzeichnet ein benanntes Nichtterminal und verweist auf dessen textuelle Repräsentation in dem Modul `Scanner`. Ein negativer Eintrag beschreibt eindeutig ein anonymes Nichtterminal. Die Komponente `NamedId` verweist bei anonymen Nichterminalen auf den Namen des zugehörigen benannten Nichtterminals, sonst sind die Einträge in `Id` und `NamedId` identisch. Die Komponente `IsToken` besagt, daß das Nichtterminal in der Spezifikation mit dem Zeichen „`*`“ als Token markiert wurde. In der Komponente `Def` wird auf die Definition eines Nichtterminals verwiesen. Die Komponente `Sig` verweist auf die in dem Feld `DomBuf` dargestellte Signatur eines Nichtterminals.

Die Komponente `Sub` des Datentyps `Rule` verweist auf die Hyper-Regeln eines Nichterminals. Jedes Nichtterminal (also auch ein benanntes) kann einen EBNF-Operator darstellen, was durch die Erweiterungen `Grp`, `Opt` und `Rep` dieses Datentyps ausgedrückt wird. Ein Optional bzw. eine Wiederholung besitzen eine leere Alternative, für die die Komponente `Formal` die formale Parameterliste aufnimmt.

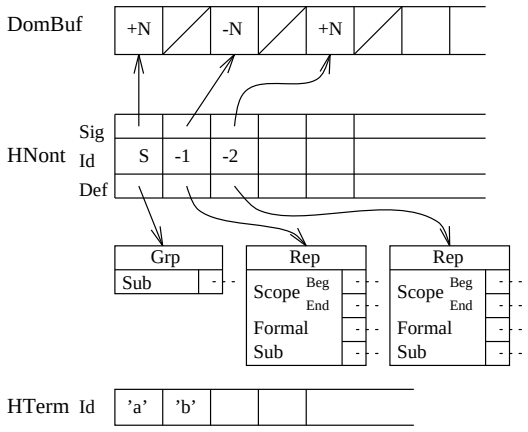



Abbildung 4.2: Erläuterung der Zeigerstruktur von Hyper-Nichtterminalen

Die Signatur eines Hyper-Nichtterminals wird durch eine Folge von Einträgen in das Feld `DomBuf` beschrieben. Ein negativer Eintrag kennzeichnet einen Eingabeparameter, ein positiver einen Ausgabeparameter. Der absolute Wert verweist in beiden Fällen auf das Feld `MNont` und kennzeichnet dadurch das Wertebereichssymbol eines Parameters. Der Eintrag `nil` beschließt eine Signatur. Die Funktionsprozeduren `SigOk` und `WellMatched` stellen die Konsistenz von Signaturen sicher.

Ein Hyper-Terminal wird durch einen eindeutigen Eintrag in dem Feld `HTerm` repräsentiert. Die Komponente `Id` verweist auch hier auf die textuelle Repräsentation.

Die Variable `StartSym` verweist auf das Startsymbol der Hyper-Grammatik.

### 4.1.3 Hyper-Regeln

```
CONST
  firstHAlt = 0; firstHFactor = 0;

TYPE
  Alt = POINTER TO RECORD
    Ind, Up: INTEGER; Next: Alt;
    Sub, Last: Factor;
    Scope: ScopeDesc;
    Formal, Actual: ParamsDesc;
    Pos: IO.Position
  END;

  Factor = POINTER TO FactorDesc;
  FactorDesc = RECORD
    Ind: INTEGER;
    Prev, Next: Factor
  END;
```

```
Nont = POINTER TO RECORD (FactorDesc)
  Sym: INTEGER;
  Actual: ParamsDesc;
  Pos: eIO.Position
END;
Term = POINTER TO RECORD (FactorDesc)
  Sym: INTEGER;
  Pos: eIO.Position
END;

VAR
  NextHAlt: INTEGER; NextHFactor: INTEGER; NONont: INTEGER;
  All, Reach, Prod, Null, Pred: eSets.OpenSet;

PROCEDURE NewAlt (VAR A: Alt; Sym: INTEGER; Formal,
            Actual: ParamsDesc; Sub,Last: Factor; Pos: eIO.Position);
PROCEDURE NewNont (VAR F: Factor; Sym: INTEGER; Actual: ParamsDesc;
            Pos: eIO.Position);
PROCEDURE NewTerm (VAR F: Factor; Sym: INTEGER; Pos: eIO.Position);
```

Die transformierten EBNF-Regeln werden samt Parametern in den Datenstrukturen `Alt` und `Factor` dargestellt.

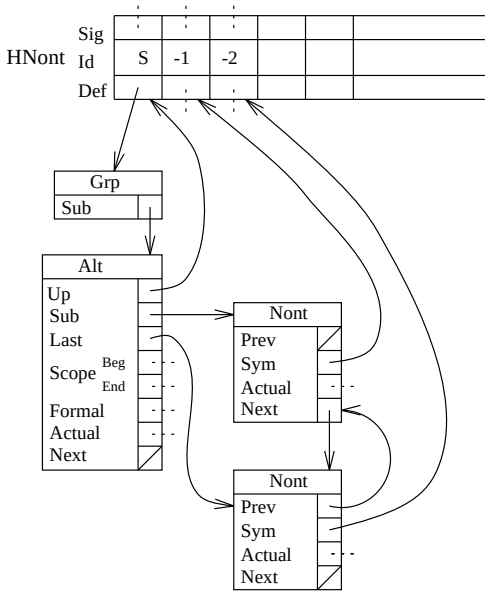



Abbildung 4.3: Erläuterung der Zeigerstrukturen einer Hyper-Regel

Ein Hyper-Nichtterminal tritt auf der linken Seite eines oder mehrerer Hyper-Regeln (oder auch -Alternativen) auf, die durch den Typ `Alt` dargestellt werden. Darin gibt die Komponente `Up` dieses Nichtterminal der linken Seite an. Ferner besitzt eine Alternative formale Parameter, die durch die Komponente `Formal` dargestellt werden. Zu den Alternativen einer Wiederholung existieren zusätzlich Rekursionsparameter, die durch die Komponente `Actual` dargestellt werden. Sind zu einem Nichtterminal mehrere Alternativen vorhanden, so sind diese durch die Komponente `Next` miteinander verkettet. Die Komponenten `Sub` und `Last` verweisen auf den ersten und letzten Faktor der Alternative.

Eine Hyper-Alternative besteht aus einer (evtl. leeren) Folge von Faktoren, die durch den Typ `Factor` dargestellt werden. Jeder Faktor ist entweder ein Terminal oder ein (evtl. anonymes) Nichtterminal. Dies wird durch die Erweiterungen `Nont` und `Term` dieses Datentyps beschrieben. Deren Komponente `Sym` verweist entsprechend auf einen Eintrag in `HNont` bzw. `HTerm`. Ein Nichtterminal besitzt aktuelle Parameter. Die Faktoren einer Regel sind über die Komponenten `Next` und `Prev` doppelt verkettet.

Die Komponenten `Ind` der Alternativen und Faktoren beinhalten jeweils einen eindeutigen Index. Die Variablen `NextHAlt` und `NextHFactor` geben dabei den jeweils nächsten zu vergebenden Indexwert an. Die Variable `NONont` beinhaltet die Anzahl der Knoten vom Typ `Nont`.

Zu einer EAG werden die Mengen der erreichbaren, produktiven und leerableitbaren Hyper-Nichtterminale sowie die Menge der Prädikate abgespeichert. Da in dem Feld `HNont` aus technischen Gründen (s. Seite 24) Lücken entstehen können, wird durch `All` die Menge der definierten Einträge beschrieben.

### 4.1.4 Parameter

```
TYPE
  ParamsDesc = RECORD
    Params: INTEGER;
    Pos: eIO.Position
  END;

  ScopeDesc = RECORD
    Beg, End: INTEGER
  END;

VAR
  ParamBuf:  POINTER TO ARRAY OF RECORD
    Affixform: INTEGER;
    Pos: IO.Position;
    isDef: BOOLEAN
  END;

  NodeBuf: POINTER TO ARRAY OF INTEGER;

  Var: POINTER TO ARRAY OF RECORD
    Sym, Num, Neg: INTEGER;
    Pos: eIO.Position;
    Def: BOOLEAN
  END;

PROCEDURE AppParam (Affixform: INTEGER; Pos: eIO.Position);
PROCEDURE FindVar (Sym, Num: INTEGER; Pos: eIO.Position;
            Def: BOOLEAN): INTEGER;
```

Eine Parameterliste wird durch den Typ `ParamDesc` beschrieben. Die Komponente `Pos` dieses Records bezeichnet dabei die Position im Quelltext, auf der die öffnende Klammer einer Parameterliste steht. Die Komponente `Params` verweist auf eine Folge von Parametern, die in dem Feld `ParamBuf` repräsentiert ist. Ist diese Folge leer, so hat die Komponente `Params` den Wert `empty`.

Ein Eintrag in `ParamBuf` verweist auf die Baumdarstellung einer Affixform. Dieser Ableitungsbaum besteht aus Knoten, die für die Anwendung einer Meta-Regel stehen, und aus Variablen.

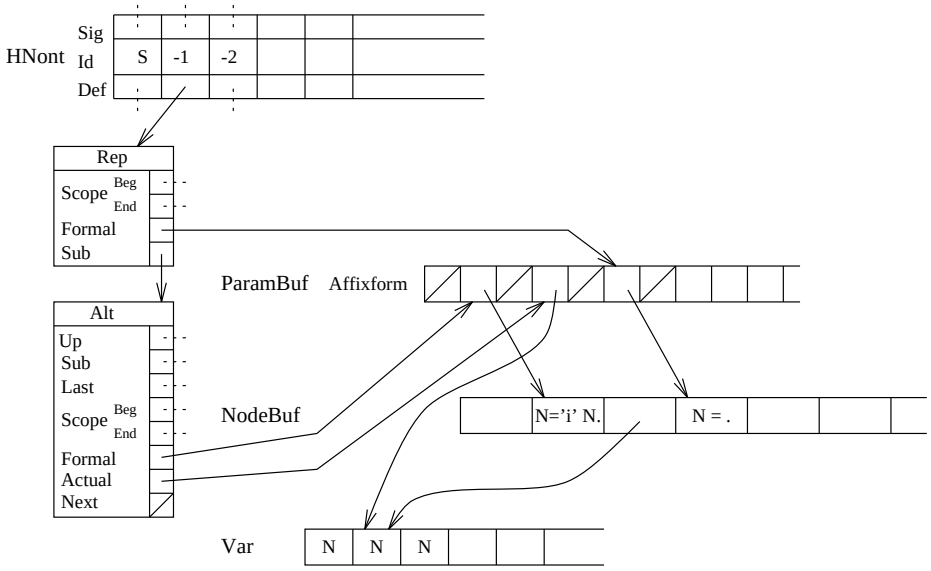



Abbildung 4.4: Erläuterung der Zeigerstruktur der Parameter

Die Knoten eines Ableitungsbaums werden im Feld `NodeBuf`, Variablen im Feld `Var`, repräsentiert. Besitzt die Komponente `Affixform` einen negativen Wert, so besteht die Affixform aus einer Variablen; der absolute Wert verweist auf einen Eintrag im Feld `Var`. Besitzt die Komponente `Affixform` einen positiven Wert, so verweist dieser auf einen Eintrag in `NodeBuf`. Die Komponente `isDef` gibt an, ob sich die Affixform auf definierender oder applizierender Affixposition befindet.

Knoten eines Ableitungsbaums werden durch aufeinanderfolgende Einträge im Feld `NodeBuf` dargestellt. `NodeBuf[i]` bezeichnet dabei die angewendete Meta-Alternative, `NodeBuf[i+j]` den j-ten Unterbaum. Ist `NodeBuf[i+j]` ein negativer Wert, so bezeichnet er eine Variable.

Eine Variable wird durch einen Eintrag in dem Feld `Var` beschieben. Ein Verweis auf diesen Eintrag kennzeichnet eindeutig eine Variable eines Gültigkeitsbereichs. Ein Gültigkeitsbereich umfaßt die Parameter einer transformierten Hyper-Regel. Die Variablen eines Gültigkeitsbereichs bilden zusammenhängende Einträge. Die Komponente `Sym` verweist auf ein Meta-Nichtterminal in dem Feld `MNont`. Die Komponente `Num` beinhaltet eine Variablennummer. Ein negativer Eintrag kennzeichnet eine Variable mit „`#`“-Operator. Die Komponente `Neg` verweist auf die jeweils negierte Form der Variable, falls diese in dem Gültigkeitsbereich vorkommt. Sonst besitzt dieser Eintrag der Wert `nil`.

Der Datentyp `Scope` kennzeichnet die Variablen eines Gültigkeitsbereichs. Sie werden in dem Feld `Var` in den Einträgen von `Beg` bis `End - 1` dargestellt.

### 4.1.5 Sonstiges

```
CONST
  BaseNameLen = 18;
VAR
  BaseName: ARRAY 18 OF CHAR;
```

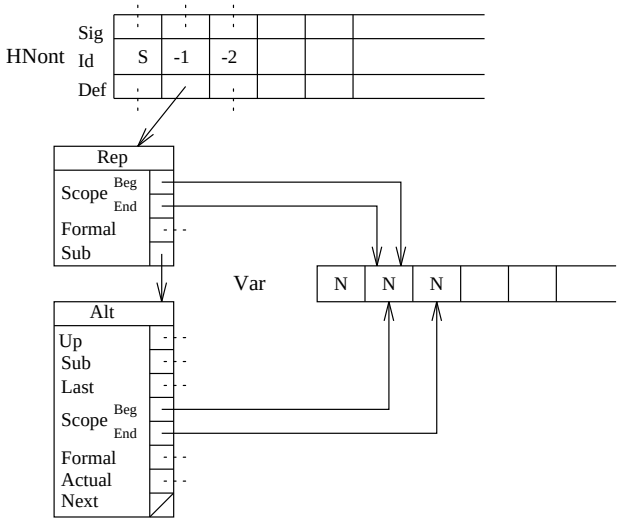



Abbildung 4.5: Erläuterung des Datentyps `Scope`

```
CONST
  analysed = 0; predicates = 1;
  parsable = 2; isSLEAG = 3;
  isSSweep = 4; hasEvaluator = 5;
VAR
  History: SET;

PROCEDURE Performed (Needed: SET): BOOLEAN;

PROCEDURE Init;

PROCEDURE WriteHNont (VAR Out: eIO.TextOut; Nont: INTEGER);
PROCEDURE WriteHTerm (VAR Out: eIO.TextOut; Term: INTEGER);
PROCEDURE WriteNamedHNont (VAR Out: eIO.TextOut; Nont: INTEGER);
PROCEDURE WriteVar (VAR Out: eIO.TextOut; V: INTEGER);
```

Die Variable `BaseName` beinhaltet den Namen der EAG, der von den einzelnen Bearbeitungsschritten zur Kennzeichnung (z.B. als Dateiname) verwendet werden kann. Die Länge des Namens wird von der Konstante `BaseNameLen` beschränkt.

Die Variable `History` spiegelt den Bearbeitungszustand der EAG wider. Als Einträge sind die Werte `analysed`, `predicates`, `parsable`, `isSLEAG`, `isSSweep` und `hasEvaluator` vordefiniert. Die Funktionsprozedur `Performed` gestattet es, einen Mindeststatus abzufragen. Ist dieser nicht erreicht, werden als Seiteneffekte Fehlermeldungen ausgegeben.

Die Prozeduren `Write`*`X`* ermöglichen eine einheitliche Ausgabe von Objekten des Datentyps *`X`*.

## 4.2 Das Einlesen von Spezifikationen

Der Analyser internalisiert die textuelle Darstellung einer EAG und legt ihre interne Repräsentation in dem Basismodul `EAG` ab.

Da in der Spezifikationssprache Epsilon (siehe Abschnitt A) die Signaturen der Hyper-Nichtter-

minale und die Meta-Regeln nicht in eigenen Abschnitten getrennt von den Hyper-Regeln angegeben werden, sind diese beim Einlesen der Hyper-Regeln im allgemeinen nicht bekannt. Da sie aber beispielsweise für die Syntaxanalyse der Affixformen vorliegen müssen, sind für die Internalisierung zwei Pässe erforderlich. Im *ersten* Pass wird die syntaktische Analyse weitestgehend durchgeführt und die Signaturen sowie die Meta-Regeln bestimmt. In der hierbei aufgebauten internen Repräsentation liegen unter Umständen Fehler vor, da aufgrund von Mehrdeutigkeiten in der kontextfreien Grammatik die aktuellen Parameter ohne Kenntnis der Signatur nicht eindeutig zugeordnet werden können. Auch läßt sich beim Einlesen nicht entscheiden, ob für einen EBNF-Operator ein anonymes Nichtterminal erzeugt werden muß oder nicht, weil der Kontext noch nicht bekannt ist. Da Affixformen nicht bearbeitet werden können, werden diese intern als Tokenfolge zwischengespeichert. Der *zweite* Pass erfolgt über der internen Repräsentation, da in diesem wesentlich die Parameter überprüft und die Struktur korrigiert werden. Hier erfolgt, wie in Abschnitt 4.4 beschrieben, die Parsierung der Affixformen nach dem Algorithmus von Earley. Für Fehlermeldungen werden Positionsangaben abgespeichert. Nach einer fehlerfreien Internalisierung werden typische Eigenschaften der kontextfreien Skelettgrammatik berechnet.

### 4.2.1 Erster Pass

Die syntaktische Analyse erfolgt durch einen rekursiv absteigenden Parser mit einem Vorgriffssymbol. Syntaktische Einheiten werden durch Prozeduren gleichen Namens erkannt. Der Aufbau der internen Darstellung erfolgt in diesen Prozeduren durch die im Modul `EAG` bereitgestellten Konstruktoranweisungen. Ein Konflikt in der kontextfreien Grammatik besteht darin, daß ein Bezeichner sowohl eine Hyper- als auch eine Meta-Regel einleitet. Dieser Konflikt wird durch Faktorisierung in der Prozedur `Specification` behoben. Ein weiterer Konflikt ergibt sich daraus, daß sowohl aktuelle als auch formale Parameter durch das Zeichen „<“ eingeleitet werden. Zur Lösung wird die Erkennung von formalen und aktuellen Parametern in der Prozedur `Params` zusammengefaßt. Werden formale Parameter erkannt, erfolgt die Übergabe in dem Prozedurparameter `Formal`, sonst in `Actual`. In dieser Prozedur wird die in Anhang A beschriebene abkürzende Schreibweise für formale Parameter erkannt.

Aufgrund folgender Mehrdeutigkeiten in der kontextfreien Grammatik lassen sich die, in den Beispielen durch <...> angedeuteten, aktuellen Parameter nicht eindeutig zuordnen.

1. `A: B` <...> `(...).`
2. `A: {... B` <...>`}.`

Im ersten Fall können die aktuellen Parameter sowohl zu `B` als auch zu dem nachfolgenden EBNF-Operator gehören. Im zweiten Fall können sie die aktuellen Parameter von `B` oder die sogenannten Rekursionsparameter einer Wiederholung darstellen. In diesen Fällen darf entweder `B` oder das Nichtterminal des entsprechenden EBNF-Operators keine Parameter besitzen, was mit Hilfe der Signatur überprüft werden kann. Da diese aber im allgemeinen nicht vorliegt, werden die aktuellen Parameter im ersten Pass der (textuell) frühestmöglichen Position zugeordnet. Im zweiten Pass erfolgt nach einer Überprüfung gegebenenfalls eine Korrektur.

Besteht die Hyper-Regel zu einem Nichtterminal aus einem EBNF-Operator ohne Kontext, soll die Transformation gemäß Tabelle 2.2 erfolgen. Da jedoch im ersten Pass ein hinterer Kontext noch nicht bekannt ist, wird für EBNF-Operatoren generell gemäß Tabelle 2.1 ein anonymes Nichtterminal eingeführt. Im zweiten Pass wird die so entstandene Struktur gegebenenfalls korrigiert.

Formale Parameter auf der linken Regelseite werden als abkürzende Schreibweise dafür erkannt, daß die Affixformen der formalen Parameter der nachfolgenden Alternativen identisch sind. In diesem Fall dürfen die nachfolgenden Alternativen keine formalen Parameter besitzen; jede dieser Alternativen erhält durch die Prozedur `Distribute` eine Kopie der Parameter.

Der Parser bietet eine Fehlerbehandlung, die nach Syntaxfehlern mit der Erkennung fortfährt.

### 4.2.2 Zweiter Pass

Im zweiten Pass werden zuerst die aufgrund von Mehrdeutigkeit falsch entstandenen Strukturen korrigiert. Hiernach erfolgt die Überprüfung der Parametrisierung und weiterer Kontextbedingungen.

Da alle Hyper-Regeln bekannt sind, läßt sich anhand der internen Struktur überprüfen, ob ein benanntes Nichtterminal vorliegt. Dieser Fall liegt vor, falls ein Nichtterminal intern keine formalen Parameter besitzt und der einzige Faktor der einzigen Alternative dieses Nichtterminals keine aktuellen Parameter hat und ein EBNF-Operator ist. Es wird dann durch die Prozedur `Shrink` die Regel des benannten Nichtterminals gelöscht und die des anonymen verschoben (s. Abbildung 4.6).

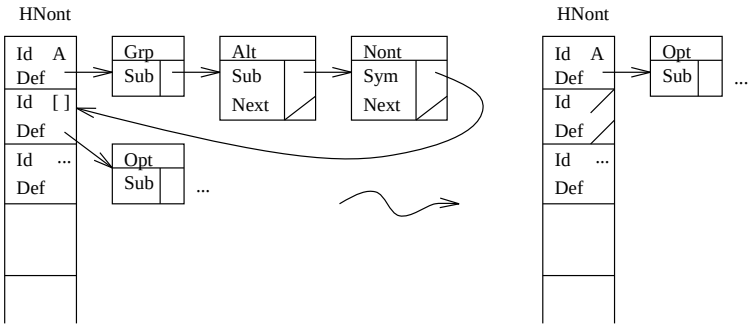



Abbildung 4.6: Beispiel einer Reduktion durch die Prozedur `Shrink`

Dadurch entstehen in dem Feld `EAG.HNont` Lücken. Deshalb gibt es die Menge `EAG.All`, die diejenigen Indizes in `EAG.HNont` enthält, die tatsächlich ein Hyper-Nichtterminal darstellen.

Aufgrund der oben beschriebenen Mehrdeutigkeiten kann eine falsche Struktur entstanden sein. Intern muß in beiden Fällen eine aktuelle Parameterliste mit einer leeren vertauscht werden. Die Prozeduren `CheckActual` und `CheckRep` überprüfen die Struktur und korrigieren diese gegebenenfalls.

### 4.2.3 Analyse der Skelettgrammatik

Im folgenden wird die Berechnung der Mengen der erreichbaren, der leerableitbaren und der produktiven Hyper-Nichtterminale diskutiert.

Die Menge der erreichbaren Nichtterminale besteht aus dem Startsymbol und allen Nichtterminalen, die als Faktor einer Regel eines erreichbaren Nichtterminals vorkommen. Die Berechnung von `EAG.Reach` erfolgt durch Traversierung der Grammatik mittels der rekursiven Prozedur `ComputeReach`.

Ein Hyper-Nichtterminal ist leerableitbar, falls es zu diesem Nichtterminal eine terminale Ableitung durch die Grundgrammatik gibt, die das leere Wort darstellt. Somit sind alle Nichtterminale leerableitbar, die eine Alternative ohne Faktoren besitzen, ebenso Optionen und Wiederholungen; auch sind alle Nichtterminale leerableitbar, die eine Alternative besitzen, in der alle Faktoren leerableitbar sind. Der Induktionsschritt läßt sich durch iteratives Löschen der Nichtterminale in Alternativen und Testen, ob eine Alternative leer ist, nachahmen. Dies erfolgt in der Implementierung symbolisch durch Dekrementieren eines Zählers, der die Anzahl der Nichtterminale einer Alternative darstellt (`Deg`), und dem Vergleich auf 0. Die Initialmenge wird durch einen Keller

repräsentiert (`Stack`). Für die Berechnung der leerableitbaren Nichtterminale werden Alternativen mit Terminalen ausgeblendet. Damit der Algorithmus linear ist, wird in dem Feld `Edge` eine Struktur aufgebaut, in der von Nichtterminalen auf deren Vorkommen verwiesen wird. Die ersten Einträge dieses Feldes werden als Einsprungstellen genutzt.

Hyper-Terminale sind produktiv, und ein Hyper-Nichtterminal ist produktiv, falls es eine Alternative besitzt, in der alle Faktoren produktiv sind. Wie bei der Berechnung der leerableitbaren Nichtterminale wird ein iteratives Löschen in Alternativen durchgeführt, wobei hier zusätzlich alle Terminale gelöscht werden. Dies wird ebenfalls durch entsprechendes Dekrementieren des Zählers implementiert. Konkret wird die Bestimmung der produktiven Nichtterminale als Fortsetzung der Bestimmung der leerableitbaren durch Löschen der Terminale realisiert.

## 4.3 Der Scanner

Der Scanner wird zum zeichenweisen Einlesen von Epsilon-Spezifikationen verwendet. Er zerlegt einen Eingabetext anhand der dort verwendeten regulären Sprachanteile und liefert eine entsprechende Tokenfolge; für Zeichenketten, Bezeichner und Zahlen abstrahiert er dabei von deren textueller Repräsentation. Er bietet die folgende Schnittstelle:

```
CONST
  eot = 0X; str = 22X; ide = "A"; num = "0";

VAR
  Val : INTEGER;
  Pos : eIO.Position;
  ErrorCounter : INTEGER;

PROCEDURE Get(VAR Tok : CHAR);
PROCEDURE Init(Input : eIO.TextIn);
PROCEDURE WriteRepr(Out : eIO.TextOut; Id : INTEGER);
PROCEDURE GetRepr(Id : INTEGER; VAR Name : ARRAY OF CHAR);
```

Nach der Initialisierung des Scanners auf einen Eingabetext kann die Prozedur `Get` zum Erhalt des jeweils nächsten Token verwendet werden, dessen Anfangsposition danach über die Variable `Pos` zur Verfügung steht. Für Zeichenketten, Bezeichner und Zahlen wird der Parameter `Tok` auf die Tokenkonstanten `str`, `ide` und `num` gesetzt. Das Eingabeende wird durch die Konstante `eot` angezeigt. Alle anderen (lesbaren) Zeichen werden durch sich selbst repräsentiert, bis auf das Zeichen „~“, das außer in Zeichenketten und Kommentaren als Eingabeende interpretiert wird.

Bei Erkennung einer Zahl wird ihr Wert der Variablen `Val` zugewiesen, bei Erkennung von Zeichenketten und Bezeichnern wird dort ein eindeutiger Verweis auf ihre textuelle Repräsentation abgelegt. Gleiche Zeichenketten bzw. Bezeichner erhalten dabei identische Verweise. Mit Hilfe der Prozeduren `WriteRepr` und `GetRepr` können über diese Verweise die textuellen Repräsentationen der zugehörigen Token ausgegeben bzw. in ein ausreichend langes Zeichenfeld kopiert werden.

Zwischenraum (Leerzeichen, Tabulatoren, Zeilenwechsel, ...) und Kommentare trennen Token und werden überlesen. Zahlen außerhalb des gültigen Bereichs sowie nicht geschlossene Zeichenketten und Kommentare werden als Fehler gemeldet; die Anzahl der erkannten Fehler wird in der Variablen `ErrorCounter` gespeichert.

Die Implementierung der Prozedur `Get` erfordert ein einzelnes Vorgriffszeichen in einer globalen Variablen `c`. Nach Überlesen von Zwischenraum und Kommentaren wird anhand dieses Vorgriffszeichens das zu erkennende Token bestimmt; die Erkennung von Zeichenketten, Bezeichnern, Zah-

len und Kommentaren erfolgt in entsprechenden Unterprozeduren. Zur Speicherung der textuellen Repräsentationen werden folgende Datenstrukturen verwendet:

```
CharBuf : POINTER TO ARRAY OF CHAR; NextChar : INTEGER;

Ident : POINTER TO ARRAY OF RECORD
          Repr     : INTEGER;
          HashNext : INTEGER
        END;
NextIdent : INTEGER;

HashTable : ARRAY 97 OF INTEGER;
```

In `CharBuf` werden lückenlos hintereinander die Zeichen aller bisher erkannten Bezeichner und Zeichenketten abgelegt. Bei Zeichenketten wird zur einfacheren Erkennung gleicher Einträge nur das öffnende Anführungszeichen gespeichert, das schließende wird von den Ausgabeprozeduren wieder angefügt. Ein Eintrag in `Ident` steht für einen Bezeichner oder eine Zeichenkette; die Komponente `Repr` verweist auf den Anfang der entsprechenden Zeichenfolge in `CharBuf`, die Länge ergibt sich aus der `Repr`-Komponente des jeweils nächsten Eintrags. Beide Felder sind expandierbar, um unnötige Beschränkungen zu vermeiden.

Bei Erkennung von Bezeichnern und Zeichenketten wird ihre textuelle Repräsentation hinter die bereits aufgenommenen Repräsentationen in `CharBuf` eingetragen. Für die Zuordnung eines eindeutigen Verweises muß die erkannte Zeichenfolge anschließend mit allen bisherigen Einträgen verglichen werden. Ist sie neu, so wird sie in das Feld `Ident` aufgenommen und der dortige Index wird der Variablen `Val` zugewiesen. Ansonsten wird der Verweis eines alten Eintrags verwendet und die neue Zeichenfolge wird durch bloßes Rücksetzen von `NextChar` „gelöscht“.

Um den Aufwand der Suche nach Zeichenfolgen zu verringern, wird zu jedem erkannten Bezeichner bzw. jeder erkannten Zeichenkette ein Hashwert berechnet. Der entsprechende Eintrag in `HashTable` verweist auf den Kopf der Liste aller bisherigen Zeichenfolgen mit diesem Hashwert; die Suche kann auf diese Liste beschränkt werden. Die Listen werden im Feld `Ident` durch die Komponente `HashNext` realisiert.

Die vorgestellte Technik ist äußerst effizient und erlaubt die Speicherung unbeschränkt langer Zeichenfolgen. Abbildung 4.7 veranschaulicht die Verwendung der Datenstrukturen.

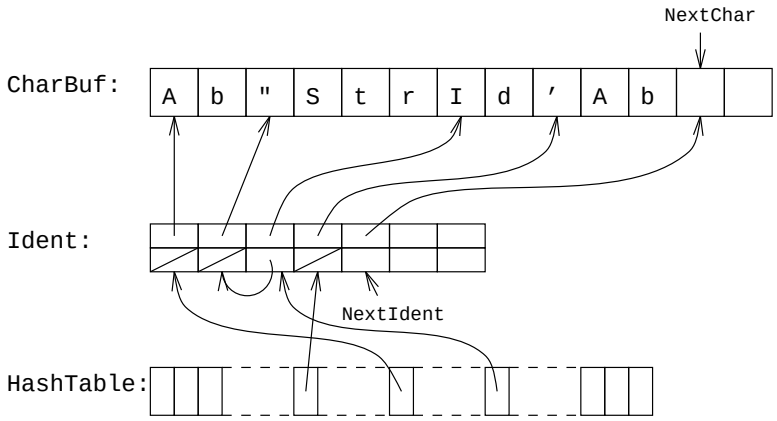



Abbildung 4.7: Situation nach Erkennung der Token `Ab`, `"Str"` , `Id`, `'Str'` und `'Ab'`

## 4.4 Der Earley-Parser

Die in Hyper-Regeln als Parameter angegebenen Affixformen müssen im Rahmen der Internalisierung durch den Analyser zum einen auf ihre syntaktische Korrektheit geprüft werden; zum anderen ist es im Hinblick auf später verwendete Algorithmen sinnvoll, im Modul `EAG` keine flache, textuelle Repräsentation der Affixformen abzulegen, sondern ihre Ableitungsbäume, aus denen benötigte Strukturinformationen abgelesen werden können. Das hier vorgestellte Modul `Earley` deckt diese beiden Aufgaben ab.

Syntaktisch korrekt sind Affixformen, wenn sie Satzformen zum jeweiligen Wertebereichssymbol sind, in denen Variablen anstelle von Meta-Nichtterminalen vorkommen. Eine Überprüfung muß für beliebige, eventuell auch mehrdeutige Meta-Grammatiken möglich sein und erfolgt hier wie in Eta durch einen von Earley vorgeschlagenen universellen Parser [Earley], der keine Generierung erfordert, sondern direkt durch eine Grammatik gesteuert wird. In der weiteren Beschreibung dieses Moduls müssen wir beim Leser die Kenntnis des originalen Algorithmus voraussetzen, da eine umfassende Erläuterung des Verfahrens den gesetzten Rahmen sprengen würde. Ein Earley-Parser kann als parallel vorgehender Bottom-up-Parser angesehen werden; während die Eingabe tokenweise gelesen wird, werden – vom Startsymbol ausgehend – bottom up alle möglichen Teilableitungsbäume für das bisher gelesene Eingabepräfix parallel in sogenannten *Itemlisten* gehalten. Für korrekte Eingaben liegt schließlich ein vollständiger Ableitungsbaum vor.

Da der ursprüngliche Earley-Parser keine Satzformen, sondern nur Sätze erkennen kann, wurde dieses Problem in Eta auf das Wortproblem zurückgeführt, indem die Meta-Grammatik um Regeln erweitert wurde, in denen die Meta-Nichtterminale als neue Terminale vorkommen. Dies führt jedoch ganz unnötig zu einer erhöhten Kompliziertheit und auch Ineffizienz, da sich der originale Algorithmus durch eine triviale Änderung im Scanner-Schritt auf die Erkennung von Satzformen erweitern läßt. Eine zweite Erweiterung des Earley-Parsers dient dazu, nicht nur die Korrektheit von Affixformen überprüfen, sondern auch die zugehörigen Ableitungsbäume aufbauen zu können.

Das Modul bietet die folgende Schnittstelle:

```
PROCEDURE Init;
PROCEDURE Finit;

PROCEDURE StartAffixform() : INTEGER;
PROCEDURE AppMSym(Sym, Num : INTEGER; Pos : eIO.Position);
PROCEDURE EndAffixform(Pos : eIO.Position);
PROCEDURE CopyAffixform(From : INTEGER; VAR To : INTEGER);

PROCEDURE Parse(Dom, Affixform : INTEGER; VAR Tree : INTEGER;
                Def : BOOLEAN);
```

Anfangs muß das Modul durch Aufruf von `Init` initialiert werden, zum Schluß können mit `Finit` die dabei dynamisch angelegten Datenstrukturen wieder freigegeben werden.

Während des Einlesens der Affixformen können diese mit den Prozeduren `StartAffixform`, `AppMSym` und `EndAffixform` in einen Puffer in diesem Modul übertragen werden. `StartAffixform` liefert dabei einen eindeutigen Verweis auf die gespeicherte Affixform zurück. Die Prozedur `CopyAffixform` wird vom Analyser bei der Auflösung der verkürzten Angabe von Parametern auf der linken Seite dazu verwendet, um Affixformen zu duplizieren.

Liegen alle Meta-Regeln im Modul `EAG` vor, so kann die Prozedur `Parse` zur Überprüfung einer vorher eingetragenen Affixform und dem Aufbau eines zugehörigen Ableitungsbaumes verwendet werden. Die Prozedur erwartet neben der zu behandelnden Affixform das Wertebereichssymbol sowie eine Markierung als Eingabe, die angibt, ob die betreffende Affixform auf definierender

oder applizierender Position steht. Diese Markierung wird unbesehen zu neu angelegten Variablen gespeichert und wird später vom Analyser benutzt. Als Ausgabe wird ein Verweis auf den Ableitungsbaum zurückgegeben, der in den Feldern `NodeBuf` und `Var` des Moduls `EAG` aufgebaut wurde.

Für die Zwischenspeicherung der Affixformen und die Parsierung durch den Algorithmus von Earley werden die folgenden globalen Datenstrukturen verwendet:

```
CONST
  end = MIN(INTEGER);
  nil = EAG.nil;

VAR
  MSymBuf : POINTER TO ARRAY OF RECORD
              Sym, Num : INTEGER;
              Pos : IO.Position
            END;
  NextMSym : INTEGER;

  ItemBuf : POINTER TO ARRAY OF RECORD
              Dot, Back, Left, Sub : INTEGER
            END;
  NextItem, CurList, PrevList : INTEGER;

  Predicted : POINTER TO ARRAY OF BOOLEAN;
```

In `MSymBuf` werden Affixformen in aufeinanderfolgenden Einträgen abgelegt. Jede Affixform wird dabei durch einen zusätzlichen Eintrag beendet, dessen Komponente `Sym` auf `end` gesetzt wird. Hier darf nicht die Konstante `nil` verwendet werden, da der Earley-Parser die Meta-Grammatik zum jeweiligen Wertebereichssymbol um eine neue Startregel erweitert, in der `end` als neues Terminal vorkommt.

Im Algorithmus von Earley wird für jedes Token einer Affixform eine Liste von Items erstellt; die Effizienz einer Implementierung wird fast ausschließlich von der Repräsentation dieser Itemlisten bestimmt. Hier werden Itemlisten in aufeinanderfolgenden Einträgen des Feldes `ItemBuf` realisiert. Das Ende einer Liste wird durch einen zusätzlichen Eintrag angezeigt, dessen Komponente `Dot` auf `nil` gesetzt ist. Eine eigene Datenstruktur für die Listen selbst ist nicht erforderlich, da eine Liste durch einen Verweis auf ihr erstes Element bezeichnet werden kann. Der Beginn der jeweils aktuellen und der vorherigen Itemliste wird in den beiden Variablen `CurList` und `PrevList` gespeichert.

Ein Item besteht konzeptionell aus ursprünglich nur zwei Komponenten. Die eine enthält eine Meta-Regel, in der durch Einfügung eines Punktes der bereits erkannte Anteil markiert ist. Dies wird hier durch die Komponente `Dot` realisiert, die einfach in das Feld `EAG.Memb` verweist, in dem die Meta-Regeln in einer auf den Earley-Parser zugeschnittenen Weise abgelegt sind. Die zweite Komponente eines Items, der sog. Back-Zeiger, zeigt auf die Itemliste, in der mit der Erkennung der Regel begonnen wurde. Hier wird also einfach direkt auf das erste Item dieser Liste verwiesen.

Um mit dem Earley-Parser auch Ableitungsbäume aufbauen zu können, werden die Items um die beiden Komponenten `Left` und `Sub` erweitert. `Left` zeigt jeweils auf den „Vorgänger“, also das Item, bei dem der Punkt ein Symbol weiter links steht; steht in einem Item der Punkt bereits am Anfang, wurde es also in einem Predictor-Schritt eingefügt, so ist `Left` auf `nil` gesetzt. Der `Sub`-Zeiger eines Items wird gesetzt, wenn es im Completer-Schritt eingefügt wird. In diesem Fall zeigt `Sub` auf dasjenige Item in der gleichen Liste, in dem der Punkt ganz hinten steht und das damit den Completer-Schritt auslöste. Diese Komponente verweist also letztlich auf einen „Unterbaum“.

Nach Erkennung einer korrekten Affixform besteht die letzte Itemliste aus nur einem Item, von dem aus über die `Left`- und `Sub`-Verkettung ein Erkennungsweg rückwärts nachvollzogen werden kann. Die über diese beiden Komponenten vom letzten Item aus erreichbaren Items bilden zusammen mit diesen Komponenten selbst als Kanten einen Baum, aus dem in einer Traversierung ein Ableitungsbaum erstellt werden kann. Für eine mehrdeutige Meta-Grammatik spiegelt dieser Ableitungsbaum die erste gefundene Ableitung wider; insbesondere ist relevant, daß für eine mehrdeutig ableitbare triviale Affixform, die nur aus einer Meta-Variable besteht, diese triviale Ableitung erkannt wird. Um den Baumaufbau zu erleichtern, werden die für eine Affixform zu erstellenden Variablen bereits während der Erkennung im Scanner-Schritt in `EAG.Var` angelegt; Verweise auf sie werden als negative Zahlen in den Komponenten `Sub` abgelegt und später in den Ableitungsbaum übertragen.

Abbildung 4.8 zeigt eine kleine Grammatik und den Ableitungsbaum einer Affixform mit Variablen. Abbildung 4.9 zeigt die vom Earley-Parser dafür erstellten und von der Wurzel über `Left` und `Sub` erreichbaren Items in der erwähnten Baumdarstellung. `Left`-Zeiger sind darin als nach links zeigende Pfeile dargestellt, `Sub`-Zeiger als nach rechts zeigende Pfeile. Die drei gepunkteten Pfeile sind Verweise auf bereits angelegte Variablen.

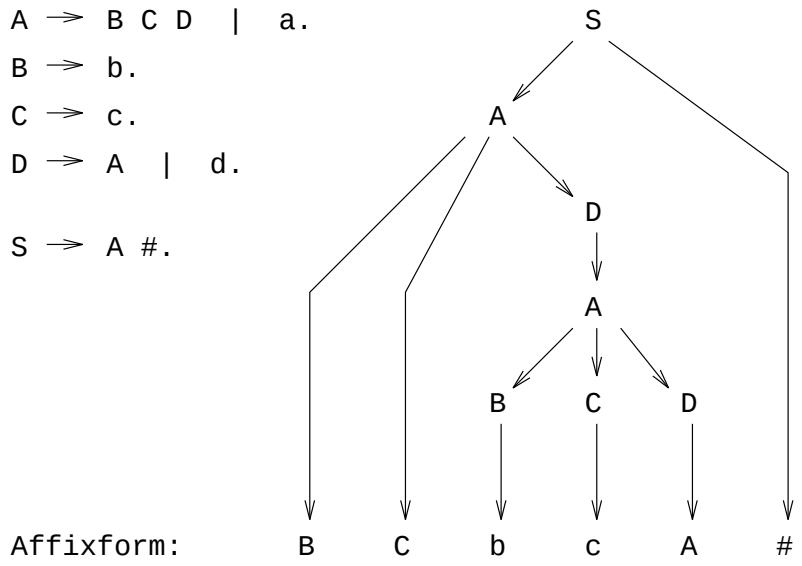



Abbildung 4.8: Meta-Grammatik und Ableitungsbaum für eine Affixform

In dieser Implementierung wurden noch einige wesentliche Optimierungen eingebaut. So wird eine Vorausschau von einem Token verwendet, um unnötige Aufnahmen von Items in die aktuelle Liste zu vermeiden; ein Item wird nicht eingefügt, wenn das Symbol hinter dem Punkt ein Terminal ist und nicht mit dem nächsten Eingabesymbol übereinstimmt. Dies führt nicht nur zu einer Beschleunigung, sondern auch zu mitunter deutlich kleineren Itemlisten. Insbesondere konnte dadurch auf Hashing beim Suchen von Items verzichtet werden, da der hierbei konstante Aufwand der Initialisierung der Hashtabelle bei Anlegen einer neuen Liste jetzt – für übliche Affixformen und Meta-Grammatiken – größer ist als die Einsparung beim Suchen in diesen Listen.

Ferner wird das boolesche Feld `Predicted` verwendet, um den Predictor-Schritt höchstens einmal für jedes Nichtterminal pro Itemliste durchzuführen.

Eine letzte Optimierung ergab sich aus der Analyse üblicher EAGen: Annähernd 90% aller Affixformen bestehen entweder trivial nur aus dem Wertebereichssymbol selbst oder einfach aus einer rechten Regelseite. Die Erkennung und der Aufbau der zugehörigen Ableitungsbäume kann direkt viel schneller als mit dem Earley-Parser erfolgen. Diese Behandlung einfacher Affixformen ist in

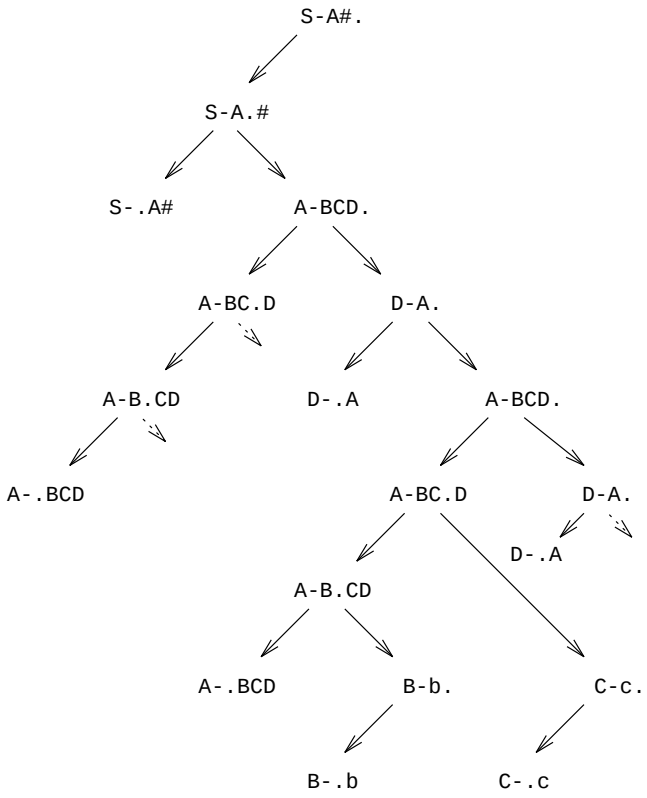



Abbildung 4.9: Baumdarstellung von Items des Earley-Parsers

der Prozedur `SimpleParse` implementiert. Der Earley-Parser wird nur noch nach einem Scheitern dieser Prozedur aufgerufen.

## 4.5 Die Bestimmung der Prädikate

In dem Modul `Predicates.Mod` werden die Prädikate einer EAG bestimmt. Prädikate sind diejenigen Hyper-Nichtterminale, die unter Ausblendung der Parameter kontextfrei höchstens zum leeren Wort ableitbar sind. Auch sollten die Prädikate produktiv sein.

Um diese Menge zu bestimmen, wird das Komplement der Menge der Prädikate berechnet. Diese Menge besteht aus dem Komplement der Menge der leerableitbaren Nichtterminale erweitert um die Nichtterminale mit Terminalen auf der rechten Regelseite. Steht ein Nichtterminal aus dem Komplement der Prädikate auf der rechten Seite, so gehört das Nichtterminal der linken Seite ebenfalls zu dieser Menge.

In der Implementierung erfolgt die Berechnung der Prädikate in der Prozedur `Check`. Die Berechnung der Komplementmenge erfolgt in zwei Schritten. Im ersten Schritt, der durch die Prozedur `BuildEdge` durchgeführt wird, wird die Initialmenge berechnet, die durch einen Keller (`Stack`) repräsentiert wird. Damit der Algorithmus linear ist, wird in dem Feld `Edge` eine Struktur aufgebaut, in der von Nichtterminalen auf deren Vorkommen verwiesen wird. Im zweiten Schritt werden in der Prozedur `ClearStack` für jedes Nichtterminal auf dem Keller alle Nichtterminale, auf deren rechten Regelseite es vorkommt, in die Komplementmenge der Prädikate aufgenommen und auf den Keller gelegt. Die Menge der Prädikate ergibt sich nun durch Komplementbildung. Die Anzahl der berechneten Prädikate wird ausgegeben.

Nach Berechnung der Prädikate können diese mit dem Kommando `List` ausgegeben werden.

## 4.6 Implementierungen

### 4.6.1 eEAG.Mod

```
MODULE eEAG; (* JoDe 22.10.96, Version 1.01 *)

  IMPORT Sets := eSets, IO := eIO, Scanner := eScanner;

  CONST
    nil* = 0;  empty* = 0;
    analysed* = 0; predicates* = 1; parsable* = 2; isSLEAG* = 3; isSSweep* = 4; hasEvaluator* = 5;
  VAR
    History*: SET;

  CONST
    firstParam* = 0;
    firstHAlt* = 0;
    firstHFactor* = 0;
  TYPE
    ParamsDesc* = RECORD Params*: INTEGER; Pos*: IO.Position END;
    OpenParamBuf =  POINTER TO ARRAY OF RECORD
      Affixform*: INTEGER; Pos*: IO.Position; isDef*: BOOLEAN
    END;
  VAR
    ParamBuf*: OpenParamBuf; NextParam*: INTEGER;

  CONST
    firstNode* = 1;
    firstVar* = 1;
  TYPE
    ScopeDesc* = RECORD Beg*, End*: INTEGER END;
    OpenNodeBuf* = POINTER TO ARRAY OF INTEGER;
    OpenVar* = POINTER TO ARRAY OF RECORD
                Sym*, Num*, Neg*: INTEGER; Pos*: IO.Position; Def*: BOOLEAN
               END;
  VAR
    NodeBuf*: OpenNodeBuf; NextNode*: INTEGER;
    Var*: OpenVar; NextVar*, Scope*: INTEGER;

  TYPE
    Rule* = POINTER TO RuleDesc;
    Alt* = POINTER TO AltDesc;
    RuleDesc = RECORD Sub*: Alt END;
    Factor* = POINTER TO FactorDesc;
    AltDesc = RECORD
              Ind*, Up*: INTEGER; Next*: Alt; Sub*, Last*: Factor;
              Formal*, Actual*: ParamsDesc;
              Scope*: ScopeDesc;
              Pos*: IO.Position
            END;
    Grp*  = POINTER TO RECORD (RuleDesc) END;
    Opt*  = POINTER TO RECORD (RuleDesc)
            EmptyAltPos*: IO.Position;
            Scope*: ScopeDesc;
            Formal*: ParamsDesc
          END;
    Rep*  = POINTER TO RECORD (RuleDesc)
            EmptyAltPos*: IO.Position;
            Scope*: ScopeDesc;
            Formal*: ParamsDesc
          END;
    FactorDesc = RECORD Ind*: INTEGER; Prev*, Next*: Factor END;
    Term* = POINTER TO RECORD (FactorDesc) Sym*: INTEGER; Pos*: IO.Position END;
    Nont* = POINTER TO RECORD (FactorDesc) Sym*: INTEGER; Actual*: ParamsDesc; Pos*: IO.Position END;
  VAR NextHAlt*, NextHFactor*, NOAlt, NOTerm, NONont*, NOOpt, NORep, NOGrp: INTEGER;

  CONST
    firstDom* = 0;
  TYPE
    OpenDomBuf = POINTER TO ARRAY OF INTEGER;
  VAR
    DomBuf*: OpenDomBuf; NextDom*, CurSig: INTEGER;

  CONST
    firstMAlt* = 1;
    firstMemb* = 1;
  TYPE
```

```
  OpenMAlt* = POINTER TO ARRAY OF RECORD Left*, Right*, Arity*, Next*: INTEGER END;
  OpenMembBuf* = POINTER TO ARRAY OF INTEGER;
VAR
  MAlt*: OpenMAlt; NextMAlt*: INTEGER; MaxMArity*: INTEGER;
  MembBuf*: OpenMembBuf; NextMemb*: INTEGER;

CONST
  firstMTerm* = 1;
  firstMNont* = 1;
  firstHTerm* = 0;
  firstHNont* = 0;
TYPE
  OpenMTerm* = POINTER TO ARRAY OF RECORD Id*: INTEGER END;
  OpenMNont* = POINTER TO ARRAY OF RECORD Id*, MRule*, Last: INTEGER; IsToken*: BOOLEAN END;
  OpenHTerm* = POINTER TO ARRAY OF RECORD Id*: INTEGER END;
  OpenHNont* = POINTER TO ARRAY OF RECORD
                 Id*, NamedId*, Sig*: INTEGER; Def*: POINTER TO RuleDesc; IsToken*: BOOLEAN
               END;
VAR
  MTerm*: OpenMTerm; NextMTerm*: INTEGER;
  MNont*: OpenMNont; NextMNont*: INTEGER;
  HTerm*: OpenHTerm; NextHTerm*: INTEGER;
  HNont*: OpenHNont; NextHNont*, NextAnonym: INTEGER;

CONST BaseNameLen* = 18;
VAR
  All*, Prod*, Reach*, Null*, Pred*: Sets.OpenSet; StartSym*: INTEGER;
  BaseName*: ARRAY BaseNameLen OF CHAR;

PROCEDURE Expand*;
  VAR
    ParamBuf1: OpenParamBuf;
    HNont1: OpenHNont;
    HTerm1: OpenHTerm;
    MNont1: OpenMNont;
    MTerm1: OpenMTerm;
    DomBuf1: OpenDomBuf;
    MAlt1: OpenMAlt;
    MembBuf1: OpenMembBuf;
    NodeBuf1: OpenNodeBuf;
    Var1: OpenVar;
    i: LONGINT;

  PROCEDURE NewLen(ArrayLen: LONGINT): LONGINT;
  BEGIN
    IF ArrayLen < MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen + 1 ELSE HALT(99) END
  END NewLen;

BEGIN  (* Expand; *)
  IF NextParam >= LEN(ParamBuf^) THEN NEW(ParamBuf1, NewLen(LEN(ParamBuf^)));
    FOR i := firstParam TO LEN(ParamBuf^) - 1 DO ParamBuf1[i] := ParamBuf[i] END; ParamBuf := ParamBuf1
  END;
  IF NextMTerm >= LEN(MTerm^) THEN NEW(MTerm1, NewLen(LEN(MTerm^)));
    FOR i := firstMTerm TO LEN(MTerm^) -1 DO MTerm1[i] := MTerm[i] END; MTerm := MTerm1
  END;
  IF NextMNont >= LEN(MNont^) THEN NEW(MNont1, NewLen(LEN(MNont^)));
    FOR i := firstMNont TO LEN(MNont^) - 1 DO MNont1[i] := MNont[i] END; MNont:= MNont1
  END;
  IF NextHTerm >= LEN(HTerm^) THEN NEW(HTerm1, NewLen(LEN(HTerm^)));
    FOR i := firstHTerm TO LEN(HTerm^) - 1 DO HTerm1[i] := HTerm[i] END; HTerm := HTerm1
  END;
  IF NextHNont >= LEN(HNont^) THEN NEW(HNont1, NewLen(LEN(HNont^)));
    FOR i := firstHNont TO LEN(HNont^) - 1 DO HNont1[i] := HNont[i] END; HNont := HNont1
  END;
  IF NextDom >= LEN(DomBuf^) THEN NEW(DomBuf1, NewLen(LEN(DomBuf^)));
    FOR i := firstDom TO LEN(DomBuf^) - 1 DO DomBuf1[i] := DomBuf[i] END; DomBuf := DomBuf1
  END;
  IF NextMAlt >= LEN(MAlt^) THEN NEW(MAlt1, NewLen(LEN(MAlt^)));
    FOR i := firstMAlt TO LEN(MAlt^) - 1 DO MAlt1[i] := MAlt[i] END; MAlt := MAlt1
  END;
  IF NextMemb >= LEN(MembBuf^) THEN NEW(MembBuf1, NewLen(LEN(MembBuf^)));
    FOR i := firstMemb TO LEN(MembBuf^) - 1 DO MembBuf1[i] := MembBuf[i] END; MembBuf := MembBuf1
  END;
  IF NextNode >= LEN(NodeBuf^) THEN NEW(NodeBuf1, NewLen(LEN(NodeBuf^)));
    FOR i := firstNode TO LEN(NodeBuf^) - 1 DO NodeBuf1[i] := NodeBuf[i] END; NodeBuf := NodeBuf1
  END;
  IF NextVar >= LEN(Var^) THEN NEW(Var1, NewLen(LEN(Var^)));
    FOR i := firstVar TO LEN(Var^) - 1 DO Var1[i] := Var[i] END; Var := Var1
  END
END Expand;
```

```
PROCEDURE AppParam*(Affixform: INTEGER; Pos: IO.Position);
BEGIN
  ParamBuf[NextParam].Affixform := Affixform; ParamBuf[NextParam].Pos := Pos;
  INC(NextParam); IF NextParam >= LEN(ParamBuf^) THEN Expand END
END AppParam;

PROCEDURE FindMTerm*(Id: INTEGER): INTEGER;
  VAR Sym: INTEGER;
BEGIN
  Sym := firstMTerm; MTerm[NextMTerm].Id := Id;
  WHILE Id # MTerm[Sym].Id DO INC(Sym) END;
  IF Sym = NextMTerm THEN INC(NextMTerm); IF NextMTerm >= LEN(MTerm^) THEN Expand END END;
  RETURN Sym
END FindMTerm;

PROCEDURE FindMNont*(Id: INTEGER): INTEGER;
  VAR Sym: INTEGER;
BEGIN
  Sym := firstMNont; MNont[NextMNont].Id := Id;
  WHILE Id # MNont[Sym].Id DO INC(Sym) END;
  IF Sym = NextMNont THEN
    MNont[NextMNont].MRule := nil;
    MNont[NextMNont].Last := nil;
    MNont[NextMNont].IsToken := FALSE;
    INC(NextMNont); IF NextMNont >= LEN(MNont^) THEN Expand END
  END;
  RETURN Sym
END FindMNont;

PROCEDURE FindHTerm*(Id: INTEGER): INTEGER;
  VAR Sym: INTEGER;
BEGIN
  Sym := firstHTerm; HTerm[NextHTerm].Id := Id;
  WHILE Id # HTerm[Sym].Id DO INC(Sym) END;
  IF Sym = NextHTerm THEN INC(NextHTerm); IF NextHTerm >= LEN(HTerm^) THEN Expand END END;
  RETURN Sym
END FindHTerm;

PROCEDURE FindHNont*(Id: INTEGER): INTEGER;
  VAR Sym: INTEGER;
BEGIN
  Sym := firstHNont; HNont[NextHNont].Id := Id;
  WHILE Id # HNont[Sym].Id DO INC(Sym) END;
  IF Sym = NextHNont THEN
    HNont[NextHNont].NamedId := Id; HNont[NextHNont].Sig := -1; HNont[NextHNont].Def := NIL;
    HNont[NextHNont].IsToken := FALSE;
    INC(NextHNont); IF NextHNont >= LEN(HNont^) THEN Expand END
  END;
  RETURN Sym
END FindHNont;

PROCEDURE NewAnonymNont*(Id: INTEGER): INTEGER;
BEGIN
  HNont[NextHNont].Id := NextAnonym;
  HNont[NextHNont].NamedId := Id;
  HNont[NextHNont].Sig := -1;
  HNont[NextHNont].Def := NIL;
  HNont[NextHNont].IsToken := FALSE;
  DEC(NextAnonym);
  INC(NextHNont); IF NextHNont >= LEN(HNont^) THEN Expand END;
  RETURN NextHNont - 1
END NewAnonymNont;

PROCEDURE AppDom*(Dir: CHAR; Dom: INTEGER);
BEGIN
  IF Dir = "-" THEN Dom := - Dom END;
  DomBuf[NextDom] := Dom;
  INC(NextDom); IF NextDom >= LEN(DomBuf^) THEN Expand END
END AppDom;

PROCEDURE WellMatched*(Sig1, Sig2: INTEGER): BOOLEAN;
BEGIN
  IF Sig1 = Sig2 THEN RETURN TRUE
  ELSE
    WHILE (DomBuf[Sig1] = DomBuf[Sig2]) & (DomBuf[Sig1] # nil) & (DomBuf[Sig2] # nil) DO
      INC(Sig1); INC(Sig2)
    END;
    RETURN (DomBuf[Sig1] = nil) & (DomBuf[Sig2] = nil)
  END
```

```
END WellMatched;

PROCEDURE SigOK*(Sym: INTEGER): BOOLEAN;
BEGIN
  IF HNont[Sym].Sig < 0 THEN
    HNont[Sym].Sig := CurSig;
    DomBuf[NextDom] := nil;
    INC(NextDom); IF NextDom >= LEN(DomBuf^) THEN Expand END;
    CurSig := NextDom;
    RETURN TRUE
  ELSE
    DomBuf[NextDom] := nil; NextDom := CurSig;
    RETURN WellMatched(HNont[Sym].Sig, CurSig)
  END
END SigOK;

PROCEDURE NewMAlt*(Sym, Right: INTEGER): INTEGER;
  VAR Arity, i: INTEGER;
BEGIN
  MAlt[NextMAlt].Next := nil; (* MNont[Sym].MRule; *)
  IF MNont[Sym].MRule = nil THEN
    MNont[Sym].MRule := NextMAlt
  ELSE
    MAlt[MNont[Sym].Last].Next := NextMAlt
  END;
  MNont[Sym].Last := NextMAlt; (* MNont[Sym].MRule := NextMAlt; *)
  MAlt[NextMAlt].Left := Sym;
  MAlt[NextMAlt].Right := Right;
  i := Right; Arity := 0;
  WHILE MembBuf[i] # 0 DO IF MembBuf[i] > 0 THEN INC(Arity) END; INC(i) END;
  MAlt[NextMAlt].Arity := Arity;
  IF Arity > MaxMArity THEN MaxMArity := Arity END;
  INC(NextMAlt); IF NextMAlt >= LEN(MAlt^) THEN Expand END;
  RETURN NextMAlt-1
END NewMAlt;

PROCEDURE AppMemb*(Val: INTEGER);
BEGIN
  MembBuf[NextMemb] := Val;
  INC(NextMemb); IF NextMemb >= LEN(MembBuf^) THEN Expand END
END AppMemb;

PROCEDURE FindVar*(Sym, Num: INTEGER; Pos: IO.Position; Def: BOOLEAN): INTEGER;
  VAR V: INTEGER;
BEGIN
  V := Scope; Var[NextVar].Sym := Sym; Var[NextVar].Num := Num;
  WHILE (Var[V].Sym # Sym) OR (Var[V].Num # Num) DO INC(V) END;
  IF V = NextVar THEN
    V := Scope; Var[NextVar].Num := - Num;
    WHILE (Var[V].Sym # Sym) OR (Var[V].Num # - Num) DO INC(V) END;
    IF V # NextVar THEN
      Var[V].Neg := NextVar; Var[NextVar].Neg := V
    ELSE
      Var[NextVar].Neg := nil
    END;
    V := NextVar; Var[NextVar].Num := Num;
    Var[NextVar].Pos := Pos; Var[NextVar].Def := Def;
    INC(NextVar); IF NextVar >= LEN(Var^) THEN Expand END
  ELSE
    Var[V].Def := Var[V].Def OR Def
  END;
  RETURN V
END FindVar;

PROCEDURE NewTerm*(VAR F: Factor; Sym: INTEGER; Pos: IO.Position);
  VAR F1: Term;
BEGIN
  INC(NOTerm);
  NEW(F1); F1.Next := NIL; F1.Sym := Sym; F1.Pos := Pos;
  F1.Ind := NextHFactor; INC(NextHFactor);
  IF F = NIL THEN F := F1; F.Prev := NIL
  ELSE F(Factor).Next := F1; F1.Prev := F; F := F(Factor).Next END
END NewTerm;

PROCEDURE NewNont*(VAR F: Factor; Sym: INTEGER; Actual: ParamsDesc; Pos: IO.Position);
  VAR F1: Nont;
BEGIN
  INC(NONont);
  NEW(F1); F1.Next := NIL; F1.Sym := Sym; F1.Actual := Actual; F1.Pos := Pos;
  F1.Ind := NextHFactor; INC(NextHFactor);
```

```
  IF F = NIL THEN F := F1; F.Prev := NIL
  ELSE F(Factor).Next := F1; F1.Prev := F; F := F(Factor).Next END
END NewNont;

PROCEDURE NewGrp*(Sym: INTEGER; Sub: Alt);
  VAR N: Grp; A: Alt;
BEGIN
  IF HNont[Sym].Def = NIL THEN INC(NOGrp);
    NEW(N); N.Sub := Sub;
    HNont[Sym].Def := N
  ELSE
    A := HNont[Sym].Def(Grp).Sub;
    WHILE A.Next # NIL DO A := A.Next  END;
    A.Next := Sub
  END
END NewGrp;

PROCEDURE NewOpt*(Sym: INTEGER; Sub: Alt; Formal: ParamsDesc; Pos: IO.Position);
  VAR N: Opt;
BEGIN
  INC(NOOpt);
  NEW(N); N.Sub := Sub; N.EmptyAltPos := Pos;
  N.Scope.Beg := nil; N.Scope.End := nil;
  N.Formal := Formal; HNont[Sym].Def := N
END NewOpt;

PROCEDURE NewRep*(Sym: INTEGER; Sub: Alt; Formal: ParamsDesc; Pos: IO.Position);
  VAR N: Rep;
BEGIN
  INC(NORep);
  NEW(N); N.Sub := Sub; N.EmptyAltPos := Pos;
  N.Scope.Beg := nil; N.Scope.End := nil;
  N.Formal := Formal; HNont[Sym].Def := N
END NewRep;

PROCEDURE NewAlt*(VAR A: Alt; Sym: INTEGER; Formal, Actual: ParamsDesc; Sub, Last: Factor; Pos: IO.Position);
  VAR A1: Alt;
BEGIN
  INC(NOAlt);
  NEW(A1); A1.Next := NIL; A1.Up := Sym;
  A1.Scope.Beg := nil; A1.Scope.End := nil;
  A1.Formal := Formal; A1.Actual := Actual;
  A1.Sub := Sub; A1.Last := Last; A1.Pos := Pos; A1.Ind := NextHAlt; INC(NextHAlt);
  IF A = NIL THEN A := A1 ELSE A.Next := A1; A := A.Next END
END NewAlt;

PROCEDURE WriteHTerm*(VAR Out: IO.TextOut; Term: INTEGER);
BEGIN
  Scanner.WriteRepr(Out, HTerm[Term].Id);
END WriteHTerm;

PROCEDURE WriteHNont*(VAR Out: IO.TextOut; Nont: INTEGER);
BEGIN
  IF HNont[Nont].Id < 0 THEN IO.Write(Out, "A"); IO.WriteInt(Out, - HNont[Nont].Id)
  ELSE Scanner.WriteRepr(Out, HNont[Nont].Id)
  END
END WriteHNont;

PROCEDURE WriteVar*(VAR Out: IO.TextOut; V: INTEGER);
BEGIN
  IF Var[V].Num < 0 THEN IO.Write(Out, "#") END;
  Scanner.WriteRepr(Out, MNont[Var[V].Sym].Id);
  IF ABS(Var[V].Num) > 1 THEN IO.WriteInt(Out, ABS(Var[V].Num) - 2) END;
END WriteVar;

PROCEDURE WriteNamedHNont*(VAR Out: IO.TextOut; Nont: INTEGER);
BEGIN
  Scanner.WriteRepr(Out, HNont[Nont].NamedId)
END WriteNamedHNont;

PROCEDURE Performed*(Needed: SET): BOOLEAN;
BEGIN
  Needed := Needed - History;
  IF Needed = {} THEN RETURN TRUE
  ELSE
    IF analysed IN Needed THEN IO.WriteText(IO.Msg, "\n\tanalyse a specification first") END;
    IF predicates IN Needed THEN IO.WriteText(IO.Msg, "\n\tcheck for predicates first") END;
    IF parsable IN Needed THEN IO.WriteText(IO.Msg, "\n\ttest for ELL1 attribute first") END;
    IF isSLEAG IN Needed THEN IO.WriteText(IO.Msg, "\n\ttest for SLEAG attribute first") END;
    IF isSSweep IN Needed THEN IO.WriteText(IO.Msg, "\n\ttest for single sweep attribute first") END;
```

```
      IF hasEvaluator IN Needed THEN IO.WriteText(IO.Msg, "\n\tgenerate an evaluator first") END;
      IO.Update(IO.Msg);
      RETURN FALSE
    END;
  END Performed;

  PROCEDURE Init*;
  BEGIN
    NEW(ParamBuf, 1023); NextParam := firstParam;
    ParamBuf[NextParam].Affixform := nil; INC(NextParam); (* ParamBuf[firstParam] := empty *)
    NEW(MTerm, 255); NextMTerm := firstMTerm;
    NEW(MNont, 255); NextMNont := firstMNont;
    NEW(HTerm, 255); NextHTerm := firstHTerm;
    NEW(HNont, 255); NextHNont := firstHNont; NextAnonym := -1;
    NEW(DomBuf, 255); NextDom := firstDom;
    DomBuf[NextDom] := nil; INC(NextDom); (* DomBuf[firstDom] := empty *) CurSig := NextDom;
    NEW(MAlt, 255); NextMAlt := firstMAlt;
    NEW(MembBuf, 255); NextMemb := firstMemb;
    NEW(NodeBuf, 1023); NextNode := firstNode;
    NEW(Var, 511); NextVar := firstVar; Scope := NextVar;
    NextHAlt := firstHAlt; NextHFactor := firstHFactor;
    Null := NIL; Prod := NIL; Pred := NIL;
    NOAlt := 0; NOTerm := 0; NONont := 0; NOGrp := 0; NOOpt := 0; NORep := 0;
    History := {}; BaseName := "nothing";
    MaxMArity := 0
  END Init;

BEGIN
  History := {}; BaseName := "nothing";
  IO.WriteText(IO.Msg, "Epsilon 1.02   JoDe/SteWe  22.11.96\n"); IO.Update(IO.Msg)
END eEAG.
```

### 4.6.2 eAnalyser.Mod

```
MODULE eAnalyser;  (* JoDe 07.11.96, Version 1.02 *)

  IMPORT Sets := eSets, IO := eIO, Scanner := eScanner, EAG := eEAG, Earley := eEarley;

  CONST nil = EAG.nil;

  VAR Tok: CHAR; ErrorCounter: INTEGER; NameNotified: BOOLEAN;

  PROCEDURE Str(s: ARRAY OF CHAR);
  BEGIN IO.WriteText(IO.Msg, s)
  END Str;

  PROCEDURE Error(Pos: IO.Position; ErrMsg: ARRAY OF CHAR);
  BEGIN
    INC(ErrorCounter);
    IF ErrorCounter > 25 THEN Str("\nToo many errors !\n"); IO.Update(IO.Msg); HALT(99) END;
    Str("\n  "); IO.WritePos(IO.Msg, Pos); Str("\t"); Str(ErrMsg)
  END Error;

  PROCEDURE Specification;
    (* Specification:
     *  (MetaRule | HyperRule) {MetaRule | HyperRule} .
     *)
    VAR Id: INTEGER; IsToken: BOOLEAN;

    PROCEDURE MetaRule(Id: INTEGER; IsToken: BOOLEAN);
      (* MetaRule:
       *  ident "=" MetaExpr ".".
       *)
      VAR MNont: INTEGER;

      PROCEDURE MetaExpr;
        (* MetaExpr:
         *  MetaTerm {"|" MetaTerm}.
         *)
        VAR Rhs: INTEGER;

        PROCEDURE MetaTerm;
          (* MetaTerm:
           *  {ident | string}.
           *)

        BEGIN
          LOOP
            IF Tok = Scanner.ide THEN
              EAG.AppMemb(EAG.FindMNont(Scanner.Val));
              Scanner.Get(Tok);
              IF Tok = Scanner.num THEN Error(Scanner.Pos, "number is not allowed here"); Scanner.Get(Tok) END
            ELSIF Tok = Scanner.str THEN
              EAG.AppMemb(- EAG.FindMTerm(Scanner.Val));
              Scanner.Get(Tok)
            ELSE
              EAG.AppMemb(EAG.nil); EXIT
            END
          END
        END MetaTerm;

      BEGIN (* MetaExpr; *)
        LOOP
          Rhs := EAG.NextMemb;
          MetaTerm;
          EAG.AppMemb(EAG.NewMAlt(MNont, Rhs));
          IF Tok = "|" THEN Scanner.Get(Tok) ELSE EXIT END
        END
      END MetaExpr;

    BEGIN (* MetaRule(Id: INTEGER; IsToken: BOOLEAN); *)
      MNont := EAG.FindMNont(Id);
      EAG.MNont[MNont].IsToken := EAG.MNont[MNont].IsToken OR IsToken;
      IF Tok = "=" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "'=' expected") END;
      MetaExpr;
      IF Tok = "." THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "'.' expected") END
    END MetaRule;

    PROCEDURE SetBaseName;
      VAR Name: ARRAY 256 OF CHAR; i: INTEGER;
    BEGIN
```

```
  EAG.StartSym := EAG.firstHNont;
  IF EAG.NextHNont > EAG.firstHNont THEN
    Scanner.GetRepr(EAG.HNont[EAG.StartSym].Id, Name);
    i := 0; WHILE (Name[i] # 0X) & (i < EAG.BaseNameLen - 1) DO EAG.BaseName[i] := Name[i]; INC(i) END;
    EAG.BaseName[i] := 0X;
  END;
END SetBaseName;

PROCEDURE HyperRule(Id: INTEGER; IsToken: BOOLEAN);
  (* HyperRule:
   *  ident [FormalParams] ":" HyperExpr "." .
   *)
  VAR HNont, Sig: INTEGER; HExpr: EAG.Alt; Actual, Formal: EAG.ParamsDesc; AltPos: IO.Position;

  PROCEDURE Distribute(Sym: INTEGER; A: EAG.Alt; Sig: INTEGER; Formal: EAG.ParamsDesc);

    PROCEDURE CopyParams(VAR s, d: INTEGER);
      VAR Affixform, P: INTEGER;
    BEGIN
      d := EAG.NextParam; P := s;
      WHILE EAG.ParamBuf[P].Affixform # nil DO
        Earley.CopyAffixform(EAG.ParamBuf[P].Affixform, Affixform);
        EAG.AppParam(Affixform, EAG.ParamBuf[P].Pos);
        INC(P)
      END;
      EAG.AppParam(nil, EAG.ParamBuf[P].Pos)
    END CopyParams;

  BEGIN (* Distribute(Sym: INTEGER; A: EAG.Alt; Sig: INTEGER; Formal: EAG.ParamsDesc) *)
    EAG.HNont[Sym].Sig := Sig;
    A.Formal.Pos := Formal.Pos; A.Formal.Params := Formal.Params; A := A.Next;
    WHILE A # NIL DO
      A.Formal.Pos := Formal.Pos;
      CopyParams(Formal.Params, A.Formal.Params);
      A := A.Next
    END
  END Distribute;

  PROCEDURE Params(VAR Actual, Formal: EAG.ParamsDesc);
    (* FormalParams:
     *  "<" ("+" | "-") Affixform ":" ident {"," ("+" | "-") Affixform ":" ident} ">".
     * ActualParams:
     *  "<" Affixform {"," Affixform} ">".
     *)
    VAR P: EAG.ParamsDesc; isFormal: BOOLEAN; Dir: CHAR; Sym: INTEGER;

    PROCEDURE Affixform(VAR Sym: INTEGER);
      (* Affixform:
       *  {string | ["#"] ident [number]}.
       *)
      VAR Uneq: SHORTINT; Cnt, Num: INTEGER; Pos: IO.Position;
    BEGIN
      Cnt := 0;
      LOOP
        Pos := Scanner.Pos;
        IF Tok = Scanner.str THEN
          Sym := - EAG.FindMTerm(Scanner.Val);
          Num := 0;
          Scanner.Get(Tok);
          INC(Cnt)
        ELSIF (Tok = "#") OR (Tok = Scanner.ide) THEN
          IF Tok = "#" THEN Uneq := -1; Scanner.Get(Tok) ELSE Uneq := 1 END;
          IF Tok = Scanner.ide THEN
            Sym := EAG.FindMNont(Scanner.Val);
            Scanner.Get(Tok);
            IF Tok = Scanner.num THEN
              Num := Uneq*(Scanner.Val+2); Scanner.Get(Tok)
            ELSE Num := Uneq
            END;
            INC(Cnt)
          ELSE
            Error(Scanner.Pos, "Metanonterminal expected")
          END
        ELSE
          Earley.EndAffixform(Pos);
          EXIT
        END;
        Earley.AppMSym(Sym, Num, Pos);
      END;
      IF Cnt # 1 THEN Sym := -1 END
```

```
  END Affixform;

BEGIN (* Params(VAR Actual, Formal: EAG.ParamsDesc); *)
  P.Params := EAG.empty; P.Pos := Scanner.Pos;
  Actual := P; Formal := P;
  IF Tok = "<" THEN Scanner.Get(Tok);
    isFormal := (Tok="+") OR (Tok="-"); P.Params := EAG.NextParam;
    LOOP
      IF (Tok="+") OR (Tok="-") THEN
        IF ~isFormal THEN Error(Scanner.Pos, "'+' or '-' not allowed in actual params") END;
        Dir := Tok; Scanner.Get(Tok)
      ELSE
        IF isFormal THEN Error(Scanner.Pos, "'+' or '-' expected") END
      END;
      EAG.AppParam(Earley.StartAffixform(), Scanner.Pos);
      Affixform(Sym);
      IF isFormal THEN
        IF (Sym < 0) OR (Tok = ":") THEN
          IF Tok = ":" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "':' expected") END;
          IF Tok = Scanner.ide THEN
            EAG.AppDom(Dir, EAG.FindMNont(Scanner.Val)); Scanner.Get(Tok)
          ELSE
            Error(Scanner.Pos, "Metanonterminal expected")
          END;
          IF Tok = Scanner.num THEN
            Error(Scanner.Pos, "number is not allowed here"); Scanner.Get(Tok)
          END
        ELSE
          EAG.AppDom(Dir, Sym)
        END
      END;
      WHILE (Tok # ",") & (Tok # ">") & (Tok # Scanner.eot) DO
        Error(Scanner.Pos, "symbol not allowed"); Scanner.Get(Tok)
      END;
      IF Tok = "," THEN Scanner.Get(Tok) ELSE EXIT END;
    END;
    EAG.AppParam(EAG.nil, Scanner.Pos);
    IF Tok = ">" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "'>' expected") END;
    IF isFormal THEN Formal.Params := P.Params ELSE Actual.Params := P.Params END
  END
END Params;

PROCEDURE HyperExpr(HNont, Id: INTEGER; Left: CHAR; VAR HExpr: EAG.Alt; AltPos: IO.Position);
  (* HyperExpr:
   *  [FormalParams] HyperTerm [ActualParams] {"|" [FormalParams] HyperTerm [ActualParams]}.
   *)
  VAR Actual, Formal, Formal1: EAG.ParamsDesc; Last: EAG.Alt; FirstF, LastF: EAG.Factor;

  PROCEDURE HyperTerm(VAR Actual: EAG.ParamsDesc; VAR First, Last: EAG.Factor);
    (* HyperTerm:
     *   { ident [ActualParams]
     *  | string
     *  | [ActualParams] ( "(" HyperExpr ")"
     *              | "[" HyperExpr "]" [FormalParams]
     *              | "{" HyperExpr "}" [FormalParams]  )
     *  } .
     *)
    VAR
      HNont: INTEGER; HExpr: EAG.Alt;
      Formal: EAG.ParamsDesc;
      Left: CHAR; Pos: IO.Position;

  BEGIN (* HyperTerm(VAR Actual: EAG.ParamsDesc; VAR First, Last: EAG.Factor); *)
    First := NIL; Last := NIL;
    LOOP
      IF Tok = Scanner.ide THEN
        IF Actual.Params # EAG.empty THEN
          Error(Actual.Pos, "actual params not allowed here"); Actual.Params := EAG.empty
        END;
        HNont := EAG.FindHNont(Scanner.Val); Pos := Scanner.Pos; Scanner.Get(Tok);
        IF Tok = Scanner.num THEN Error(Scanner.Pos, "number is not allowed here!"); Scanner.Get(Tok) END;
        Params(Actual, Formal);
        IF Formal.Params # EAG.empty THEN Error(Formal.Pos, "formal params not allowed here") END;
        EAG.NewNont(Last, HNont, Actual, Pos); Actual.Params := EAG.empty
      ELSIF Tok = Scanner.str THEN
        IF Actual.Params # EAG.empty THEN
          Error(Actual.Pos, "actual params not allowed here"); Actual.Params := EAG.empty
        END;
        EAG.NewTerm(Last, EAG.FindHTerm(Scanner.Val), Scanner.Pos); Scanner.Get(Tok)
      ELSE
```

```
            IF Actual.Params = EAG.empty THEN
              Params(Actual, Formal);
              IF Formal.Params # EAG.empty THEN Error(Formal.Pos, "formal params not allowed here") END
            END;
            IF (Tok="(") OR (Tok="[") OR (Tok="{") THEN
              Pos := Scanner.Pos;
              HNont := EAG.NewAnonymNont(Id);
              EAG.NewNont(Last, HNont, Actual, Pos); Actual.Params := EAG.empty;
              IF Tok = "(" THEN
                Scanner.Get(Tok);
                HyperExpr(HNont, Id, "(", HExpr, Pos);
                IF Tok = ")" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "')' expected") END;
                EAG.NewGrp(HNont, HExpr)
              ELSE Left := Tok;
                Scanner.Get(Tok);
                HyperExpr(HNont, Id, Left, HExpr, Pos);
                Pos := Scanner.Pos;
                IF Left="{" THEN
                  IF Tok = "}" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "'}' expected") END
                ELSE
                  IF Tok = "]" THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "']' expected") END
                END;
                Params(Actual, Formal);
                IF ~ EAG.SigOK(HNont) THEN Error(Formal.Pos, "formal params differ") END;
                IF Left = "{" THEN
                  EAG.NewRep(HNont, HExpr, Formal, Pos)
                ELSE
                  EAG.NewOpt(HNont, HExpr, Formal, Pos)
                END
              END
            ELSE (* Actual.Params will be passed to HyperExpr *)
              RETURN
            END
          END;
          IF First = NIL THEN First := Last END
        END
      END HyperTerm;

    BEGIN (* HyperExpr(HNont, Id: INTEGER; Left: CHAR; VAR HExpr: EAG.Alt; AltPos: IO.Position); *)
      HExpr := NIL; Last := NIL;
      LOOP
        Params(Actual, Formal);
        IF ~ EAG.SigOK(HNont) THEN Error(Formal.Pos, "formal params differ") END;
        HyperTerm(Actual, FirstF, LastF);
        IF (Left="{") & (Actual.Params = EAG.empty) THEN
          Params(Actual, Formal1);
          IF Formal1.Params # EAG.empty THEN Error(Formal1.Pos, "formal params not allowed here") END
        ELSIF (Left # "{") & (Actual.Params # EAG.empty) THEN
          Error(Actual.Pos, "actual params not allowed here"); Actual.Params := EAG.empty
        END;
        EAG.NewAlt(Last, HNont, Formal, Actual, FirstF, LastF, AltPos);
        IF HExpr = NIL THEN HExpr := Last END;
        IF Tok = "|" THEN AltPos := Scanner.Pos; Scanner.Get(Tok) ELSE EXIT END
      END
    END HyperExpr;

  BEGIN (* HyperRule(Id: INTEGER; IsToken: BOOLEAN); *)
    HNont := EAG.FindHNont(Id);
    IF ~ NameNotified & (HNont = EAG.firstHNont) & (ErrorCounter = 0) & (Scanner.ErrorCounter = 0) THEN
      NameNotified := TRUE; SetBaseName; Str(EAG.BaseName); IO.Update(IO.Msg)
    END;
    EAG.HNont[HNont].IsToken := EAG.HNont[HNont].IsToken OR IsToken;
    Params(Actual, Formal);
    IF Actual.Params # EAG.empty THEN Error(Actual.Pos, "actual params not allowed here") END;
    IF (Formal.Params # EAG.empty) & (EAG.SigOK(HNont)) THEN
      Sig := EAG.HNont[HNont].Sig; EAG.HNont[HNont].Sig := EAG.empty
    END;
    IF Tok = ":" THEN AltPos := Scanner.Pos; Scanner.Get(Tok) ELSE Error(Scanner.Pos, "':' expected") END;
    HyperExpr(HNont, Id, "(", HExpr, AltPos);
    IF Formal.Params # EAG.empty THEN Distribute(HNont, HExpr, Sig, Formal) END;
    EAG.NewGrp(HNont, HExpr);
    IF Tok = "." THEN Scanner.Get(Tok) ELSE Error(Scanner.Pos, "'.' expected") END
  END HyperRule;

BEGIN (* Specification; *)
  Scanner.Get(Tok);
  REPEAT
    IsToken := FALSE;
    IF Tok = Scanner.ide THEN Id := Scanner.Val; Scanner.Get(Tok)
    ELSE Error(Scanner.Pos, "identifier of rule expected")
```

```
    END;
    IF Tok = Scanner.num THEN Error(Scanner.Pos, "number is not allowed here"); Scanner.Get(Tok) END;
    IF Tok = "*" THEN IsToken := TRUE; Scanner.Get(Tok) END;
    IF Tok = "=" THEN
      MetaRule(Id, IsToken)
    ELSE
      HyperRule(Id, IsToken)
    END;
    IF (Tok # Scanner.ide) & (Tok # Scanner.eot) THEN
      Error(Scanner.Pos, "not allowed symbol");
      REPEAT Scanner.Get(Tok) UNTIL (Tok = ".") OR (Tok = Scanner.eot);
      IF Tok # Scanner.eot THEN Scanner.Get(Tok) END;
      Error(Scanner.Pos, "    restart point")
    END
  UNTIL Tok = Scanner.eot;
  INC(ErrorCounter, Scanner.ErrorCounter)
END Specification;

PROCEDURE CheckSemantics;
  VAR Sym, n: INTEGER;

  PROCEDURE Shrink;
    VAR A: EAG.Alt; F: EAG.Nont;
  BEGIN
    IF (EAG.HNont[Sym].Id >= 0) & (EAG.HNont[Sym].Def IS EAG.Grp) THEN
      A := EAG.HNont[Sym].Def(EAG.Grp).Sub;
      IF (A.Formal.Params = EAG.empty) & (A.Next = NIL)
      & (A.Sub # NIL) & (A.Sub IS EAG.Nont) THEN F := A.Sub(EAG.Nont);
        IF (EAG.HNont[F.Sym].Id < 0) & (F.Actual.Params = EAG.empty) & (F.Next = NIL) THEN
          EAG.HNont[Sym].Def := EAG.HNont[F.Sym].Def; EAG.HNont[F.Sym].Def := NIL;
          EAG.HNont[Sym].Sig := EAG.HNont[F.Sym].Sig;
          A := EAG.HNont[Sym].Def.Sub; REPEAT A.Up := Sym; A := A.Next UNTIL A = NIL
        END
      END
    END
  END Shrink;

  PROCEDURE Traverse(Sym: INTEGER);
    VAR Node: EAG.Rule; A: EAG.Alt; F: EAG.Factor; Sig: INTEGER;

    PROCEDURE CheckParamList(Dom: INTEGER; Par: EAG.ParamsDesc; Lhs: BOOLEAN);
      VAR P: INTEGER;
    BEGIN
      P := Par.Params;
      WHILE (EAG.DomBuf[Dom] # nil) & (EAG.ParamBuf[P].Affixform # nil) DO
        EAG.ParamBuf[P].isDef := (Lhs & (EAG.DomBuf[Dom] < 0)) OR (~ Lhs & (EAG.DomBuf[Dom] > 0));
        Earley.Parse(ABS(EAG.DomBuf[Dom]), EAG.ParamBuf[P].Affixform, EAG.ParamBuf[P].Affixform,
          EAG.ParamBuf[P].isDef);
        IF EAG.ParamBuf[P].Affixform = EAG.nil THEN INC(ErrorCounter) END;
        INC(Dom); INC(P)
      END;
      IF EAG.DomBuf[Dom] # EAG.ParamBuf[P].Affixform THEN
        Error(Par.Pos, "number of affixforms differs from signature")
      END
    END CheckParamList;

    PROCEDURE CheckActual(F: EAG.Nont);
    BEGIN
      IF EAG.WellMatched(EAG.HNont[F.Sym].Sig, EAG.empty)
          & (F.Actual.Params # EAG.empty)
          & (F.Next # NIL) & (F.Next IS EAG.Nont)
          & (F.Next(EAG.Nont).Actual.Params = EAG.empty)
          & (EAG.HNont[F.Next(EAG.Nont).Sym].Id < 0) THEN
        F.Next(EAG.Nont).Actual := F.Actual;
        F.Actual.Params := EAG.empty
      END
    END CheckActual;

    PROCEDURE CheckRep(A: EAG.Alt);
      VAR F: EAG.Nont;
    BEGIN
      IF (A.Last # NIL) & (A.Last IS EAG.Nont) THEN F := A.Last(EAG.Nont);
        IF EAG.WellMatched(EAG.HNont[F.Sym].Sig, EAG.empty)
            & (F.Actual.Params # EAG.empty)
            & (A.Actual.Params = EAG.empty) THEN
          A.Actual := F.Actual;
          F.Actual.Params := EAG.empty
        END
      END;
    END CheckRep;
```

```
  BEGIN (* Traverse(Sym: INTEGER) *)
    Node := EAG.HNont[Sym].Def; Sig := EAG.HNont[Sym].Sig;
    IF Node # NIL THEN
      IF Node IS EAG.Rep THEN
        EAG.Scope := EAG.NextVar; Node(EAG.Rep).Scope.Beg := EAG.NextVar;
        CheckParamList(Sig, Node(EAG.Rep).Formal, TRUE);
        Node(EAG.Rep).Scope.End := EAG.NextVar;
      ELSIF Node IS EAG.Opt THEN
        EAG.Scope := EAG.NextVar; Node(EAG.Opt).Scope.Beg := EAG.NextVar;
        CheckParamList(Sig, Node(EAG.Opt).Formal, TRUE);
        Node(EAG.Opt).Scope.End := EAG.NextVar;
      END;
      A := Node.Sub;
      REPEAT
        EAG.Scope := EAG.NextVar; A.Scope.Beg := EAG.NextVar;
        CheckParamList(Sig, A.Formal, TRUE);
        IF Node IS EAG.Rep THEN
          CheckRep(A);
          CheckParamList(Sig, A.Actual, FALSE)
        END; F := A.Sub;
        WHILE F # NIL DO
          IF F IS EAG.Nont THEN
            CheckActual(F(EAG.Nont));
            CheckParamList(EAG.HNont[F(EAG.Nont).Sym].Sig, F(EAG.Nont).Actual, FALSE)
          END;
          F := F.Next
        END;
        A.Scope.End := EAG.NextVar;
        A := A.Next
      UNTIL A = NIL
    END
  END Traverse;

BEGIN (* CheckSemantics; *)
  Sets.New(EAG.All, EAG.NextHNont);
  IF EAG.firstHNont = EAG.NextHNont THEN INC(ErrorCounter); Str("\n  EAG needs at least one hyperrule ") END;
  FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF EAG.HNont[Sym].Def = NIL THEN
      IF EAG.HNont[Sym].Id >= 0 THEN INC(ErrorCounter);
        Str("\n  Hypernonterminal "); Scanner.WriteRepr(IO.Msg, EAG.HNont[Sym].Id); Str(" undefined")
      END
    ELSE
      Sets.Incl(EAG.All, Sym); Shrink
    END
  END;
  FOR Sym := EAG.firstMNont TO EAG.NextMNont - 1 DO
    IF EAG.MNont[Sym].MRule = nil THEN INC(ErrorCounter);
      Str("\n  Metanonterminal "); Scanner.WriteRepr(IO.Msg, EAG.MNont[Sym].Id); Str(" undefined")
    END
  END;
  IF ErrorCounter = 0 THEN
    FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
      Traverse(Sym);
    END;
    FOR n := EAG.firstVar TO EAG.NextVar - 1 DO
      IF (EAG.Var[n].Num < 0) & (EAG.Var[n].Neg = EAG.nil) THEN
        Error(EAG.Var[n].Pos, "#-operator not allowed")
      END;
      IF ~ EAG.Var[n].Def THEN INC(ErrorCounter);
        Str("\n  "); IO.WritePos(IO.Msg, EAG.Var[n].Pos); Str("\tVariable '");
        EAG.WriteVar(IO.Msg, n); Str("' never on defining position!")
      END
    END;
    IF        (EAG.DomBuf[EAG.HNont[EAG.StartSym].Sig] = EAG.nil)
        OR (EAG.DomBuf[EAG.HNont[EAG.StartSym].Sig] < 0)
        OR (EAG.DomBuf[EAG.HNont[EAG.StartSym].Sig + 1] # EAG.nil) THEN
      INC(ErrorCounter); Str("\n  Startsymbol '");
      Scanner.WriteRepr(IO.Msg, EAG.HNont[EAG.StartSym].Id);
      Str("' has to have exactly one synthesized attribute!")
    END;
    IF EAG.firstMNont = EAG.NextMNont THEN
      INC(ErrorCounter); Str("\n  EAG needs at least one metarule ")
    END;
  END
END CheckSemantics;

PROCEDURE ComputeEAGSets;
  VAR
    Edge: POINTER TO ARRAY OF RECORD Dest: EAG.Alt; Next: INTEGER END; NextEdge: INTEGER;
```

```
    Deg: POINTER TO ARRAY OF INTEGER;
    Stack: POINTER TO ARRAY OF INTEGER; Top: INTEGER;
    Sym: INTEGER; A: EAG.Alt; F: EAG.Factor; Prod: Sets.OpenSet;
    Warnings: LONGINT; TermFound: BOOLEAN;

  PROCEDURE ComputeReach(Sym: INTEGER);
    VAR A: EAG.Alt; F: EAG.Factor;
  BEGIN
    Sets.Incl(EAG.Reach, Sym);
    A := EAG.HNont[Sym].Def.Sub;
    REPEAT
      F := A.Sub;
      WHILE F # NIL DO
        IF (F IS EAG.Nont) & ~ Sets.In(EAG.Reach, F(EAG.Nont).Sym) THEN ComputeReach(F(EAG.Nont).Sym) END;
        F := F.Next
      END;
      A := A.Next
    UNTIL A = NIL
  END ComputeReach;

  PROCEDURE NewEdge(From: INTEGER; To: EAG.Alt);
  BEGIN
    Edge[NextEdge].Dest := To; Edge[NextEdge].Next := Edge[From].Next;
    Edge[From].Next := NextEdge; INC(NextEdge)
  END NewEdge;

  PROCEDURE TestDeg(A: EAG.Alt);
  BEGIN
    IF Deg[A.Ind] = 0 THEN
      IF ~ Sets.In(Prod, A.Up) THEN Sets.Incl(Prod, A.Up); Stack[Top] := A.Up; INC(Top) END
    END
  END TestDeg;

  PROCEDURE Prune;
    VAR E: INTEGER; A: EAG.Alt;
  BEGIN
    WHILE Top > 0 DO
      DEC(Top); E := Edge[Stack[Top]].Next;
      WHILE E # nil DO A := Edge[E].Dest; DEC(Deg[A.Ind]); TestDeg(A); E := Edge[E].Next END
    END
  END Prune;

BEGIN (* ComputeEAGSets; *)
  Warnings := 0;
  Sets.New(EAG.Reach, EAG.NextHNont);
  ComputeReach(EAG.StartSym);
  FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF (EAG.HNont[Sym].Def # NIL) & ~ Sets.In(EAG.Reach, Sym) & (EAG.HNont[Sym].Id >= 0) THEN
      INC(Warnings)
    END
  END;
  NEW(Deg, EAG.NextHAlt); NEW(Stack, EAG.NextHNont); Top := 0;
  NEW(Edge, EAG.NextHNont + EAG.NONont + 1); NextEdge := EAG.NextHNont;
  FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO Edge[Sym].Next := nil END;
  Sets.New(EAG.Null, EAG.NextHNont); Sets.New(Prod, EAG.NextHNont);
  FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF EAG.HNont[Sym].Def # NIL THEN
      IF (EAG.HNont[Sym].Def IS EAG.Opt) OR (EAG.HNont[Sym].Def IS EAG.Rep) THEN
        Sets.Incl(Prod, Sym); Stack[Top] := Sym; INC(Top)
      END;
      A := EAG.HNont[Sym].Def.Sub;
      REPEAT
        TermFound := FALSE; Deg[A.Ind] := 0;
        F := A.Sub;
        WHILE F # NIL DO
          IF F IS EAG.Term THEN TermFound := TRUE ELSE INC(Deg[A.Ind]); NewEdge(F(EAG.Nont).Sym, A) END;
          F := F.Next
        END;
        IF TermFound THEN INC(Deg[A.Ind], MIN(INTEGER)) ELSE TestDeg(A) END;
        A := A.Next
      UNTIL A = NIL
    END
  END;
  Prune;
  Sets.Assign(EAG.Null, Prod);
  FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF EAG.HNont[Sym].Def # NIL THEN
      A := EAG.HNont[Sym].Def.Sub;
      REPEAT
        IF Deg[A.Ind] < 0 THEN DEC(Deg[A.Ind], MIN(INTEGER)); TestDeg(A) END; A := A.Next
```

```
        UNTIL A = NIL
      END
    END;
    Prune;
    EAG.Prod := Prod;
    FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
      IF Sets.In(EAG.All, Sym) & ~ Sets.In(EAG.Prod, Sym) THEN INC(Warnings) END
    END;
    IF Warnings > 0 THEN
      Str("\n  "); IO.WriteInt(IO.Msg, Warnings); Str(" warnings occurred   eAnalyser.Warnings")
    END
  END ComputeEAGSets;

  PROCEDURE Analyse*;
    VAR Name: ARRAY 512 OF CHAR; In: IO.TextIn; OpenError: BOOLEAN;
  BEGIN
    Str("Analysing ...      "); IO.Update(IO.Msg);
    IO.InputName(Name); IO.OpenIn(In, Name, OpenError);
    IF OpenError THEN Str("\n  error: cannot open input")
    ELSE
      Scanner.Init(In); EAG.Init; Earley.Init; ErrorCounter := 0; NameNotified := FALSE;
      Specification;
      IF ErrorCounter = 0 THEN CheckSemantics; Earley.Finit END;
      IF ErrorCounter = 0 THEN ComputeEAGSets END;
      IF ErrorCounter = 0 THEN
        INCL(EAG.History, EAG.analysed); Str("   ok ")
      ELSE
        EXCL(EAG.History, EAG.analysed); Str("\nerrors occurred");
        IF ~ NameNotified THEN Str(" in "); Str(EAG.BaseName) END
      END
    END;
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END Analyse;

  PROCEDURE Warnings*;
    VAR Sym: INTEGER; Unreach, Unprod: Sets.OpenSet; NoWarnings: BOOLEAN;
  BEGIN
    Str("Analyser"); IO.Update(IO.Msg);
    IF EAG.Performed({EAG.analysed}) THEN
      Sets.New(Unreach, EAG.NextHNont); Sets.New(Unprod, EAG.NextHNont);
      Sets.Difference(Unreach, EAG.All, EAG.Reach); Sets.Difference(Unprod, EAG.All, EAG.Prod);
      NoWarnings := Sets.IsEmpty(Unreach) & Sets.IsEmpty(Unprod);
      IF NoWarnings THEN Str(": no") END;
      Str(" warnings on "); Str(EAG.BaseName); Str("'s hypernonterminals");
      IF ~ NoWarnings THEN IO.Write(IO.Msg, ":");
        FOR Sym := EAG.firstHNont TO EAG.NextHNont - 1 DO
          IF Sets.In(Unreach, Sym) & (EAG.HNont[Sym].Id >= 0) THEN
            Str("\n  '"); EAG.WriteHNont(IO.Msg, Sym); Str("' unreachable")
          END;
          IF Sets.In(Unprod, Sym) THEN
            Str("\n  ");
            IF EAG.HNont[Sym].Id < 0 THEN Str(" in ") END; IO.Write(IO.Msg,"'");
            EAG.WriteNamedHNont(IO.Msg, Sym); Str("' cannot be derived")
          END
        END
      END;
    END;
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END Warnings;

END eAnalyser.
```

### 4.6.3 eScanner.Mod

```
MODULE eScanner;    (* SteWe 11/96  -  Ver 1.7 *)

  IMPORT IO := eIO;

  CONST
    nil = 0;
    firstChar = 0; firstIdent = 1; errorIdent = firstIdent;
    eot* = 0X; str* = 22X; num* = "0"; ide* = "A";

  TYPE
    OpenCharBuf = POINTER TO ARRAY OF CHAR;
    OpenIdent = POINTER TO ARRAY OF RECORD Repr : INTEGER; HashNext : INTEGER END;

  VAR
    Val* : INTEGER;
    Pos* : IO.Position;
    c : CHAR;
    CharBuf : OpenCharBuf; NextChar : INTEGER;
    Ident : OpenIdent; NextIdent : INTEGER;
    HashTable : ARRAY 97 OF INTEGER;
    In : IO.TextIn;
    ErrorCounter* : INTEGER;

  PROCEDURE Error(String : ARRAY OF CHAR);
  BEGIN
    IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, Pos); IO.WriteText(IO.Msg, "  ");
    IO.WriteText(IO.Msg, String);
    IO.Update(IO.Msg); INC(ErrorCounter)
  END Error;

  PROCEDURE Expand;
    VAR
      i : LONGINT;
      CharBuf1 : OpenCharBuf;
      Ident1 : OpenIdent;

  BEGIN
    IF NextChar >= LEN(CharBuf^) THEN
      IF LEN(CharBuf^) <= MAX(INTEGER) DIV 2 THEN NEW(CharBuf1, LEN(CharBuf^) * 2)
      ELSE Error("internal error: CharBuf overflow\n"); HALT(99)
      END;
      FOR i := firstChar TO LEN(CharBuf^) - 1 DO CharBuf1[i] := CharBuf[i] END;
      CharBuf := CharBuf1
    END;
    IF NextIdent >= LEN(Ident^) THEN
      IF LEN(Ident^) <= MAX(INTEGER) DIV 2 THEN NEW(Ident1, LEN(Ident^) * 2)
      ELSE Error("internal error: Ident overflow\n"); HALT(99)
      END;
      FOR i := firstIdent TO LEN(Ident^) - 1 DO Ident1[i] := Ident[i] END;
      Ident := Ident1
    END
  END Expand;

  PROCEDURE Init*(Input: IO.TextIn);
    VAR i : INTEGER;
  BEGIN
    c := " ";
    CharBuf[firstChar] := str; CharBuf[firstChar + 1] := "e"; CharBuf[firstChar + 2] := "r";
    CharBuf[firstChar + 3] := "r"; NextChar := firstChar + 4;
    Ident[errorIdent].Repr := firstChar; Ident[errorIdent + 1].Repr := NextChar;
    NextIdent := firstIdent + 1;
    FOR i := 0 TO LEN(HashTable) - 1 DO HashTable[i] := nil END;
    In := Input; ErrorCounter := 0
  END Init;

  PROCEDURE GetRepr*(Id : INTEGER; VAR Name : ARRAY OF CHAR);
    VAR k, m, n : INTEGER; c : CHAR;
  BEGIN
    k := Ident[Id].Repr; c := CharBuf[k]; n := Ident[Id + 1].Repr - k;
    IF (LEN(Name) < n + 1) OR (LEN(Name) < n + 2) & ((c = str) OR (c = "'")) THEN
      IO.WriteText(IO.Msg, "\n  internal error: symbol too long\n"); IO.Update(IO.Msg); HALT(99)
    END;
    FOR m := 0 TO n - 1 DO Name[m] := CharBuf[k + m] END;
    IF (c = str) OR (c = "'") THEN Name[n] := c; INC(n) END;
    Name[n] := 0X
  END GetRepr;
```

```
PROCEDURE WriteRepr*(Out : IO.TextOut; Id : INTEGER);
  VAR k, m : INTEGER; c : CHAR;
BEGIN
  k := Ident[Id].Repr; c := CharBuf[k];
  FOR m := k TO Ident[Id + 1].Repr - 1 DO IO.Write(Out, CharBuf[m]) END;
  IF (c = str) OR (c = "'") THEN IO.Write(Out, c) END
END WriteRepr;

PROCEDURE Get*(VAR Tok : CHAR);

  PROCEDURE Comment;  (* a "!" starts a one line comment inside this comment *)
    VAR  Lev : INTEGER;  c1 : CHAR;
  BEGIN
    Lev := 1; c := " ";
    LOOP
      c1 := c; IO.Read(In, c);
      IF (c1 = "(") & (c = "*") THEN IO.Read(In, c); INC(Lev)
      ELSIF (c1 = "*") & (c = ")") THEN IO.Read(In, c); DEC(Lev);
                              IF Lev = 0 THEN EXIT END
      END;
      IF c = "!" THEN REPEAT IO.Read(In, c) UNTIL (c = IO.eol) OR (c = eot) END;
      IF c = eot THEN Error("open comment at end of text"); EXIT END
    END
  END Comment;

  PROCEDURE LookUp(OldNextChar : INTEGER);
    VAR
      Len, First, Last : INTEGER;
      HashIndex : INTEGER;
      m, n : INTEGER;

  BEGIN    (* Tok = str OR Tok = ide *) (* empty strings "" and '' would be different *)
    IF Tok = str THEN
      First := ORD(CharBuf[OldNextChar + 1]); Len := NextChar - OldNextChar + 1
    ELSE First := ORD(CharBuf[OldNextChar]); Len := NextChar - OldNextChar
    END;
    Last := ORD(CharBuf[NextChar - 1]);
    HashIndex := (((First + Last) * 2 - Len) * 4 - First) MOD LEN(HashTable);
    Val := HashTable[HashIndex];
    WHILE Val # nil DO
      n := OldNextChar; m := Ident[Val].Repr;
      IF (Tok = str) & ((CharBuf[m] = str) OR (CharBuf[m] = "'")) THEN INC(n); INC(m) END;
      WHILE CharBuf[n] = CharBuf[m] DO INC(n); INC(m) END;
      IF (n = NextChar) & (m = Ident[Val + 1].Repr) THEN NextChar := OldNextChar; RETURN
      ELSE Val := Ident[Val].HashNext
      END
    END;
    Val := NextIdent;
    Ident[Val].Repr := OldNextChar;
    Ident[Val].HashNext := HashTable[HashIndex];
    HashTable[HashIndex] := Val;
    INC(NextIdent); IF NextIdent = LEN(Ident^) THEN Expand END;
    Ident[NextIdent].Repr := NextChar
  END LookUp;

  PROCEDURE String;
    VAR Terminator : CHAR; OldNextChar : INTEGER;
  BEGIN
    OldNextChar := NextChar;
    Terminator := c;
    REPEAT
      IF NextChar = LEN(CharBuf^) THEN Expand END;
      CharBuf[NextChar] := c; INC(NextChar); IO.Read(In, c);
      IF (c = IO.eol) OR (c = eot) THEN
        Error("string terminator not in this line");
        NextChar := OldNextChar; Val := errorIdent; RETURN
      ELSIF c < " " THEN
        Error("illegal character in string");
        NextChar := OldNextChar; Val := errorIdent;
        REPEAT IO.Read(In, c) UNTIL (c = Terminator) OR (c = IO.eol) OR (c = eot);
        IF c = Terminator THEN IO.Read(In, c) END; RETURN
      END
    UNTIL c = Terminator;
    IO.Read(In, c);
    IF NextChar = OldNextChar + 1 THEN
      Error("illegal empty string"); NextChar := OldNextChar; Val := errorIdent; RETURN
    END;
    IF NextChar = LEN(CharBuf^) THEN Expand END;
    CharBuf[NextChar] := IO.eol;
    LookUp(OldNextChar)
```

```
    END String;

    PROCEDURE Ident;
      VAR OldNextChar : INTEGER;
    BEGIN
      OldNextChar := NextChar;
      REPEAT
        IF NextChar = LEN(CharBuf^) THEN Expand END;
        CharBuf[NextChar] := c; INC(NextChar); IO.Read(In, c)
      UNTIL ((c < "A") OR ("Z" < c)) & ((c < "a") OR ("z" < c));
      IF NextChar = LEN(CharBuf^) THEN Expand END;
      CharBuf[NextChar] := IO.eol;
      LookUp(OldNextChar)
    END Ident;

    PROCEDURE Number;
      VAR d : INTEGER; Ok : BOOLEAN;
    BEGIN
      Val := 0; Ok := TRUE;
      REPEAT
        IF Ok THEN
          d := ORD(c) - ORD("0");        (* 0 <= d <= 9 *)
          IF Val <= 999 THEN Val := Val * 10 + d
          ELSE Error("number out of range 0 ... 9999"); Ok := FALSE; Val := 0
          END
        END;
        IO.Read(In, c)
      UNTIL (c < "0") OR ("9" < c)
    END Number;

  BEGIN  (* Get *)
    LOOP
      WHILE (c <= " ") & (c # eot) DO IO.Read(In, c) END;
      IF c = "!" THEN REPEAT IO.Read(In, c) UNTIL (c = IO.eol) OR (c = eot)
      ELSIF c = "(" THEN IO.PrevPos(In, Pos); IO.Read(In, c);
        IF c = "*" THEN Comment ELSE Tok := "("; RETURN END
      ELSE EXIT
      END
    END;
    IO.PrevPos(In, Pos);
    IF (c = str) OR (c = "'") THEN Tok := str; String
    ELSIF (("A" <= c) & (c <= "Z")) OR (("a" <= c) & (c <= "z")) THEN Tok := ide; Ident
    ELSIF ("0" <= c) & (c <= "9") THEN Tok := num; Number
    ELSIF (c = "~") OR (c = eot) THEN Tok := eot
    ELSE Tok := c; IO.Read(In, c)
    END  (* Tok can be set to: eot, str, ide, num, any > " ", but not to !'"a..zA..Z0..9~ *)
  END Get;

BEGIN
  NEW(CharBuf, 1023); NEW(Ident, 255)
END eScanner.
```

### 4.6.4 eEarley.Mod

```
MODULE eEarley;    (* SteWe 06/96  -  Ver 1.0 *)

  IMPORT IO := eIO, EAG := eEAG;

  CONST
    end = MIN(INTEGER);
    nil = EAG.nil;
    firstMSym = 1;
    firstItem = 1;

  TYPE
    OpenMSymBuf = POINTER TO ARRAY OF RECORD Sym, Num : INTEGER; Pos : IO.Position END;
    OpenItemBuf = POINTER TO ARRAY OF RECORD Dot, Back, Left, Sub : INTEGER END;

  VAR
    MSymBuf : OpenMSymBuf; NextMSym : INTEGER;
    ItemBuf : OpenItemBuf; NextItem, CurList : INTEGER;
    Predicted : POINTER TO ARRAY OF BOOLEAN;

  PROCEDURE Expand;
    VAR
      i : LONGINT;
      MSymBuf1 : OpenMSymBuf;
      ItemBuf1 : OpenItemBuf;

    PROCEDURE NewLen(ArrayLen: LONGINT): LONGINT;
    BEGIN
      IF ArrayLen <= MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen ELSE HALT(99) END
    END NewLen;

  BEGIN
    IF NextMSym >= LEN(MSymBuf^) THEN NEW(MSymBuf1, NewLen(LEN(MSymBuf^)));
      FOR i := firstMSym TO LEN(MSymBuf^) - 1 DO MSymBuf1[i] := MSymBuf[i] END; MSymBuf := MSymBuf1
    END;
    IF NextItem >= LEN(ItemBuf^) THEN NEW(ItemBuf1, NewLen(LEN(ItemBuf^)));
      FOR i := firstItem TO LEN(ItemBuf^) - 1 DO ItemBuf1[i] := ItemBuf[i] END; ItemBuf := ItemBuf1
    END
  END Expand;

  PROCEDURE StartAffixform*() : INTEGER;
  BEGIN
    RETURN NextMSym
  END StartAffixform;

  PROCEDURE AppMSym*(Sym, Num : INTEGER; Pos : IO.Position);
  BEGIN
    IF NextMSym >= LEN(MSymBuf^) THEN Expand END;
    MSymBuf[NextMSym].Sym := Sym;
    MSymBuf[NextMSym].Num := Num;
    MSymBuf[NextMSym].Pos := Pos;
    INC(NextMSym)
  END AppMSym;

  PROCEDURE EndAffixform*(Pos : IO.Position);
  BEGIN
    AppMSym(end, 0, Pos)
  END EndAffixform;

  PROCEDURE CopyAffixform*(From : INTEGER; VAR To : INTEGER);
  BEGIN
    To := NextMSym;
    LOOP
      AppMSym(MSymBuf[From].Sym, MSymBuf[From].Num, MSymBuf[From].Pos);
      IF MSymBuf[From].Sym = end THEN EXIT ELSE INC(From) END
    END
  END CopyAffixform;

  PROCEDURE Parse*(Dom, Affixform : INTEGER; VAR Tree : INTEGER; Def : BOOLEAN);

    PROCEDURE SimpleParse(Dom, Affixform : INTEGER; VAR Tree : INTEGER; Def : BOOLEAN);
      VAR A, m, n : INTEGER;
    BEGIN
      IF (MSymBuf[Affixform].Sym = Dom) & (MSymBuf[Affixform + 1].Sym = end)  (* Trivial *)
      THEN Tree := - EAG.FindVar(MSymBuf[Affixform].Sym, MSymBuf[Affixform].Num,
                          MSymBuf[Affixform].Pos, Def)
      ELSE Tree := nil; A := EAG.MNont[Dom].MRule;  (* Direct *)
        REPEAT m := EAG.MAlt[A].Right; n := Affixform;
```

```
      WHILE EAG.MembBuf[m] = MSymBuf[n].Sym DO INC(m); INC(n) END;
      IF (EAG.MembBuf[m] = 0) & (MSymBuf[n].Sym = end) THEN
        Tree := EAG.NextNode;
        EAG.NodeBuf[EAG.NextNode] := A; INC(EAG.NextNode);
        IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END;
        n := Affixform;
        WHILE MSymBuf[n].Sym # end DO
          IF MSymBuf[n].Sym > 0 THEN
            EAG.NodeBuf[EAG.NextNode] :=
              - EAG.FindVar(MSymBuf[n].Sym, MSymBuf[n].Num, MSymBuf[n].Pos, Def);
            INC(EAG.NextNode);
            IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END
          END;
          INC(n)
        END;
        RETURN
      END;
      A := EAG.MAlt[A].Next
    UNTIL A = nil
  END
END SimpleParse;

PROCEDURE EarleyParse(Dom, Affixform : INTEGER; VAR Tree : INTEGER; Def : BOOLEAN);
  VAR PrevList, CurSym, i : INTEGER;

  PROCEDURE AddItem(Dot, Back, Left, Sub : INTEGER);
    VAR I : INTEGER;
  BEGIN
    IF (EAG.MembBuf[Dot] >= 0) OR (EAG.MembBuf[Dot] = MSymBuf[CurSym + 1].Sym) THEN
      ItemBuf[NextItem].Dot := Dot; ItemBuf[NextItem].Back := Back;
      I := CurList;
      WHILE (ItemBuf[I].Dot # Dot) OR (ItemBuf[I].Back # Back) DO INC(I) END;
      IF I = NextItem THEN
        ItemBuf[NextItem].Left := Left; ItemBuf[NextItem].Sub := Sub;
        INC(NextItem); IF NextItem >= LEN(ItemBuf^) THEN Expand END
      END;
      ItemBuf[NextItem].Dot := nil
    END
  END AddItem;

  PROCEDURE Scanner;
    VAR I, Sub : INTEGER;
  BEGIN
    IF MSymBuf[CurSym].Sym > 0 THEN
      Sub := - EAG.FindVar(MSymBuf[CurSym].Sym, MSymBuf[CurSym].Num,
                 MSymBuf[CurSym].Pos, Def)
    ELSE Sub := nil
    END;
    I := PrevList;
    REPEAT
      IF EAG.MembBuf[ItemBuf[I].Dot] = MSymBuf[CurSym].Sym THEN
        AddItem(ItemBuf[I].Dot + 1, ItemBuf[I].Back, I, Sub)
      END;
      INC(I)
    UNTIL ItemBuf[I].Dot = nil
  END Scanner;

  PROCEDURE Closure;
    VAR I, I1, I2, N, A : INTEGER; Ready : BOOLEAN;
  BEGIN
    REPEAT I := CurList; Ready := TRUE;
      REPEAT
        IF EAG.MembBuf[ItemBuf[I].Dot] > 0 THEN    (* Predictor *)
          N := EAG.MembBuf[ItemBuf[I].Dot];
          IF ~Predicted[N] THEN
            A := EAG.MNont[N].MRule;
            REPEAT AddItem(EAG.MAlt[A].Right, CurList, nil, nil); A := EAG.MAlt[A].Next
            UNTIL A = nil;
            Predicted[N] := TRUE
          END
        ELSIF EAG.MembBuf[ItemBuf[I].Dot] = 0 THEN    (* Completer *)
          N := EAG.MAlt[EAG.MembBuf[ItemBuf[I].Dot + 1]].Left;
          I1 := ItemBuf[I].Back;
          REPEAT
            IF EAG.MembBuf[ItemBuf[I1].Dot] = N THEN
              I2 := NextItem;
              AddItem(ItemBuf[I1].Dot + 1, ItemBuf[I1].Back, I1, I);
              IF (I2 < NextItem) & (ItemBuf[I1].Back = CurList) THEN
                Ready := FALSE
              END
```

```
            END;
            INC(I1)
          UNTIL ItemBuf[I1].Dot = nil
        END;
        INC(I)
      UNTIL I = NextItem
    UNTIL Ready
  END Closure;

  PROCEDURE Init(Start : INTEGER);
    VAR i : INTEGER;
  BEGIN
    IF (Predicted = NIL) OR (LEN(Predicted^) < EAG.NextMNont) THEN
      NEW(Predicted, EAG.NextMNont)
    END;
    IF MAX(INTEGER) - 3 >= EAG.NextMemb THEN INC(EAG.NextMemb, 3) ELSE HALT(99) END;
    WHILE EAG.NextMemb >= LEN(EAG.MembBuf^) DO EAG.Expand END;
    DEC(EAG.NextMemb, 3);
    EAG.MembBuf[EAG.NextMemb] := Start;
    EAG.MembBuf[EAG.NextMemb + 1] := end;
    EAG.MembBuf[EAG.NextMemb + 2] := 0;
    EAG.MembBuf[EAG.NextMemb + 3] := EAG.NextMAlt;
    EAG.MAlt[EAG.NextMAlt].Left := EAG.NextMNont;
    EAG.MAlt[EAG.NextMAlt].Right := EAG.NextMemb;
    NextItem := firstItem; CurList := NextItem;
    FOR i := EAG.firstMNont TO EAG.NextMNont - 1 DO Predicted[i] := FALSE END;
    AddItem(EAG.NextMemb, CurList, nil, nil); Closure
  END Init;

  PROCEDURE CreateTree(I : INTEGER; VAR Tree : INTEGER);
    VAR SubTree, A : INTEGER;
  BEGIN  (* Memb[ItemBuf[I].Dot] = 0 *)
    A := EAG.MembBuf[ItemBuf[I].Dot + 1];
    EAG.NodeBuf[EAG.NextNode] := A; Tree := EAG.NextNode;
    IF MAX(INTEGER) - EAG.MAlt[A].Arity - 1 >= EAG.NextNode THEN
      INC(EAG.NextNode, EAG.MAlt[A].Arity + 1)
    ELSE HALT(99)
    END;
    WHILE EAG.NextNode >= LEN(EAG.NodeBuf^) DO EAG.Expand END;
    SubTree := EAG.NextNode - 1;
    REPEAT
      IF ItemBuf[I].Sub > 0 THEN CreateTree(ItemBuf[I].Sub, EAG.NodeBuf[SubTree]); DEC(SubTree)
      ELSIF ItemBuf[I].Sub < 0 THEN EAG.NodeBuf[SubTree] := ItemBuf[I].Sub; DEC(SubTree)
      END;
      I := ItemBuf[I].Left
    UNTIL  I = nil
  END CreateTree;

BEGIN (* EarleyParse *)
  CurSym := Affixform - 1; Init(Dom);
  REPEAT
    PrevList := CurList; INC(NextItem); CurList := NextItem;
    IF NextItem >= LEN(ItemBuf^) THEN Expand END;
    FOR i := EAG.firstMNont TO EAG.NextMNont - 1 DO Predicted[i] := FALSE END;
    INC(CurSym); Scanner;
    IF CurList = NextItem THEN
      IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, MSymBuf[CurSym].Pos);
      IO.WriteText(IO.Msg, "  affixform incorrect"); IO.Update(IO.Msg);
      Tree := nil; RETURN
    ELSE Closure
    END
  UNTIL MSymBuf[CurSym].Sym = end;
  IF ItemBuf[ItemBuf[CurList].Left].Sub < 0 THEN
    Tree := ItemBuf[ItemBuf[CurList].Left].Sub
  ELSE CreateTree(ItemBuf[ItemBuf[CurList].Left].Sub, Tree)
  END;
END EarleyParse;

BEGIN (* Parse *)
  SimpleParse(Dom, Affixform, Tree, Def);
  IF Tree = nil THEN EarleyParse(Dom, Affixform, Tree, Def) END
END Parse;

PROCEDURE Init*;
BEGIN
  NEW(MSymBuf, 2047); NextMSym := firstMSym;
  NEW(ItemBuf, 1023); NextItem := firstItem;
  Predicted := NIL
END Init;
```

```
  PROCEDURE Finit*;
  BEGIN
    MSymBuf := NIL; ItemBuf := NIL; Predicted := NIL
  END Finit;

END eEarley.
```

### 4.6.5 ePredicates.Mod

```
MODULE ePredicates;  (* JoDe 22.10.96, Version 1.01 *)

  IMPORT Sets := eSets, IO := eIO, EAG := eEAG;

  PROCEDURE List*;
    VAR N: INTEGER;
  BEGIN
    IO.WriteString(IO.Msg, "Predicates in     ");
    IO.WriteString(IO.Msg, EAG.BaseName); IO.WriteText(IO.Msg, ": ");
    IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
      FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
        IF Sets.In(EAG.Pred, N) THEN
          IO.WriteText(IO.Msg, "\n\t");
          IO.WritePos(IO.Msg,  EAG.HNont[N].Def.Sub.Pos); IO.WriteString(IO.Msg, " :  ");
          EAG.WriteHNont(IO.Msg, N)
        END
      END
    END; IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END List;

  PROCEDURE Check*;
    VAR
      HNont: POINTER TO ARRAY OF INTEGER;
      Edge: POINTER TO ARRAY OF RECORD Dest, Next: INTEGER END; NextEdge: INTEGER;
      Stack: POINTER TO ARRAY OF INTEGER; Top: INTEGER;
      CoPred, Pred: Sets.OpenSet;
      NOPreds, N: INTEGER;

    PROCEDURE NewEdge(From, To: INTEGER);
    BEGIN
      Edge[NextEdge].Dest := To; Edge[NextEdge].Next := HNont[From];
      HNont[From] := NextEdge; INC(NextEdge)
    END NewEdge;

    PROCEDURE PutCoPred(N: INTEGER);
    BEGIN
      IF ~ Sets.In(CoPred, N) THEN Sets.Incl(CoPred, N); Stack[Top] := N; INC(Top) END
    END PutCoPred;

    PROCEDURE BuiltEdge;
      VAR A: EAG.Alt; F: EAG.Factor; N: INTEGER;
    BEGIN
      FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO HNont[N] := -1 END;
      FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
        IF Sets.In(EAG.All, N) THEN
          IF ~ Sets.In(EAG.Null, N) THEN PutCoPred(N) END;
          A := EAG.HNont[N].Def.Sub;
          REPEAT
            F := A.Sub;
            WHILE F # NIL DO
              IF F IS EAG.Term THEN PutCoPred(N)
              ELSE NewEdge(F(EAG.Nont).Sym, N)
              END;
              F := F.Next
            END;
            A := A.Next
          UNTIL A = NIL
        END
      END
    END BuiltEdge;

    PROCEDURE ClearStack;
      VAR Dep: INTEGER;
    BEGIN
      WHILE Top > 0 DO
        DEC(Top); Dep := HNont[Stack[Top]];
        WHILE Dep >= 0 DO PutCoPred(Edge[Dep].Dest); Dep := Edge[Dep].Next END
      END
    END ClearStack;

  BEGIN (* Check; *)
    IO.WriteString(IO.Msg, "Predicates in     ");
    IO.WriteString(IO.Msg, EAG.BaseName); IO.Update(IO.Msg);
    IF EAG.Performed({EAG.analysed}) THEN
      EXCL(EAG.History, EAG.predicates);
      NEW(HNont, EAG.NextHNont); NEW(Edge, EAG.NONont+1); NextEdge := 0;
      NEW(Stack, EAG.NextHNont); Top := 0;
```

```
      Sets.New(CoPred, EAG.NextHNont); Sets.New(Pred, EAG.NextHNont);
      BuiltEdge;
      ClearStack;
      Sets.Difference(Pred, EAG.Prod, CoPred);
      Sets.Excl(Pred, EAG.StartSym);
      EAG.Pred := Pred; INCL(EAG.History, EAG.predicates);
      NOPreds := 0;
      FOR N := EAG.firstHNont TO EAG.NextHNont DO
        IF Sets.In(Pred, N) THEN INC(NOPreds) END
      END;
      IF NOPreds > 0 THEN
        IO.WriteString(IO.Msg, ":  "); IO.WriteInt(IO.Msg, NOPreds);
        IO.WriteString(IO.Msg, "      ePredicates.List ")
      ELSE
        IO.WriteString(IO.Msg, ":  none. ")
      END;
    END; IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END Check;

END ePredicates.
```

# Kapitel 5

# Scanner- und Parsergenerator

## 5.1 Der Scannergenerator

Effiziente Scannergenerierung ist schwierig. Dies trifft im besonderen für die Scannergenerierung aus einer EAG zu, da hier die regulären Bestandteile einer Quellsprache durch Teile der kontextfreien Grundgrammatik beschrieben werden. Daher wird in Epsilon trotz der sich ergebenden Ineffizienz die Erkennung der echt regulären Sprachanteile sowie die zugehörige Parameterberechnung zusätzlich von dem generierten Parser durchgeführt und nur der durch die Hyper-Terminale beschriebene konstante Sprachanteil von einem Scanner erkannt.

Die Erkennung der echt regulären Sprachanteile erfolgt dabei anhand der vom Scanner erkannten Token, die von der Repräsentation der Hyper-Nichtterminale abstrahieren. Wird bei der Tokenerkennung das bei Scannern übliche longest-match-Prinzip angewendet, so werden aber beispielsweise Bezeichner vom Parser nicht mehr wie in höhereren Programmiersprachen erkannt, da durch den Scanner im Quelltext möglichst lange Zeichenfolgen als Token erkannt werden. Etwa würde in der Zeichenfolge „BEGINNING“ das Schlüsselwort „BEGIN“ erkannt werden, was das Erkennen des Bezeichners „BEGINNING“ durch den Parser verhinderte. Damit der Parser die regulären Strukturen von Bezeichnern und Strings wie in (z.B.) den Programmiersprachen Oberon und Modula-2 erkennen kann, wird der Quelltext vom Scanner in Bezeichner, Symbole, Strings und Kommentare zerlegt. Um diese Zerlegung zu vereinfachen, werden die Tokenrepräsentationen in die Klassen der Schlüsselwörter und der Symbole unterteilt: Ein Schlüsselwort besteht aus einer Folge von Buchstaben und Ziffern, ein Symbol aus einer Folge der übrigen „schreibbaren“ Zeichen. Die nicht schreibbaren Zeichen markieren Zwischenraum.

Symbole werden durch den Scanner nach dem longest-match-Prinzip erkannt.

Eine Folge von Buchstaben und Ziffern, die ein Schlüsselwort darstellt, wird vom Scanner als ein Token erkannt. Ansonsten stellt diese einen Bezeichner dar, der vom Parser erkannt wird. Dazu liefert der Scanner für jedes Zeichen dieser Folge ein Token.

Ein String besteht wie in Modula-2 aus einer beliebigen Zeichenfolge, die von den Zeichen „'“ bzw. „"“ umgeben wird. Weil die Erkennung von Strings durch den Parser erfolgt, erkennt der Scanner nach einem solchen Begrenzer solange zeichenweise Token, bis entweder ein entsprechender Begrenzer den String abschließt oder das Ende der Zeile bzw. des Textes erreicht ist. Um die Erkennung von Leerzeichen in einem String zu ermöglichen, ist ein einzelnes Leerzeichen als Token zulässig.

Kommentare sind beliebige Zeichenfolgen, die „geschachtelt“ durch die Symbole „`(*`“ und „`*)`“ begrenzt werden. Falls das Symbol „`(*`“ nicht explizit als Token vorkommt, werden Kommentare

vom Scanner als Zwischenraum behandelt.

Um den Generierungsaufwand zu reduzieren, wurde ein parametrischer Scanner entworfen. Dieser wird mit Paaren aus Tokenwerten und Repräsentationen von Hyper-Terminalen parametrisiert. Zwischen Token und deren Repräsentation besteht eine eindeutige Zuordnung.

### 5.1.1 Der parametrische Scanner

Zur Analyse der endlichen Sprache der Tokenrepräsentationen einer EAG werden Entscheidungsbäume verwendet. Da die Realisierung dieser Bäume recht einfach ist, kann eine vom Generator zu erstellende Tabelle vermieden werden, indem die Parametrisierung des Scanners durch eingefügte Prozduraufrufe erfolgt, wodurch die Bäume bei der Initialisierung aufgebaut werden (s. Abb. 5.1). Die Erkennung einer Tokenrepräsentation beginnt dann an der Wurzel des zugehörigen Entscheidungsbaums. In jedem Schritt wird das eingelesene Zeichen mit demjenigen des Knotens verglichen. Bei Gleichheit erfolgt ein Übergang zum Sohn (senkrechte Pfeile) zusammen mit dem Lesen des nächsten Zeichens, bei Ungleichheit wird das aktuelle Zeichen dem Bruder (waagerechte Pfeile) übergeben. Wenn es keinen Sohn mehr gibt, ist eine Tokenrepräsentation vollständig erkannt. Dagegen zeigt das erfolglose Durchsuchen aller Brüder an, daß das aktuelle Zeichen nicht zur Tokenrepräsentation gehört. Jedoch kann an einem Vorgänger bereits ein Anfang der eingelesenen Zeichenfolge vollständig erkannt worden sein. Dies spiegelt ein vorhandener Tokenwert wider.

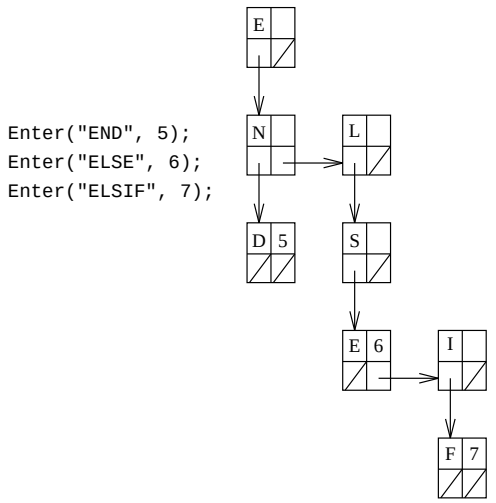



Abbildung 5.1: Beispiel eines Entscheidungsbaums.

Bei der Erkennung eines Symbols wird das longest-match-Prinzip dadurch realisiert, daß der letzte Knoten mit einem Tokenwert gewählt wird. Für eventuell überflüssige Zeichen beginnt die Erkennung erneut an der Wurzel eine Entscheidungsbaums.

Dagegen muß bei der Erkennung von Schlüsselwörtern explizit ausgeschlossen werden, daß sie nur den Anfang eines Bezeichners bilden, d.h., nach der vollständigen Erkennung darf das eingelesenen Zeichen weder Buchstabe noch Ziffer sein. Das Scheitern der Erkennung führt hier dazu, daß der überlesene Anfang zusammen mit den nachfolgenden Buchstaben und Ziffern als einzelne Zeichen in Tokenwerte umgesetzt wird. Dieses abweichende Verhalten wird in der globalen Variablen `Mode` angezeigt. Entsprechend erfolgt die Erkennung von Strings.

Zwischenraum trennt reguläre Strukturen. Dem Parser, durch den die Erkennung regulärer Strukturen erfolgt, steht im Normalfall diese Information nicht zur Verfügung, da Zwischenraum durch den Scanner überlesen wird. Daher kann der parametrische Scanner diesen als Token erkennen.

```
CONST
  nil = 0;
  eot = 0; undef = 1; whitespace = 2; comment = MIN(INTEGER);
VAR
  Node: POINTER TO ARRAY OF RECORD
        Ch: CHAR;
        Tok, Next, Sub: INTEGER
      END;
  ReprTab: ARRAY maxTok, maxTokLen OF CHAR;
  IsWhitespace, IsIdent: ARRAY 256 OF BOOLEAN;
CONST (* Mode *)
  string = 0; ident = 1;  none = 2;
VAR
  Ch, StringCh: CHAR;
  Mode: INTEGER;
  Pos*: IO.Position; Get*: PROCEDURE (VAR Tok: INTEGER);

PROCEDURE Enter(Tok: INTEGER; Repr: ARRAY OF CHAR);
PROCEDURE WriteRepr*(VAR Out: IO.TextOut; Tok: INTEGER);
PROCEDURE Symbol(VAR Tok: INTEGER);
PROCEDURE Keyword(VAR Tok: INTEGER);
PROCEDURE Comment;
PROCEDURE Get2*(VAR Tok: INTEGER);
PROCEDURE Get3*(VAR Tok: INTEGER);
PROCEDURE Init*;
```

Die Entscheidungsbäume werden im Feld `Node` repräsentiert; die Komponente `Sub` verweist auf den Sohn eines Knotens, `Next` auf den Bruder. Damit zu jedem Zeichen direkt auf die Wurzel des zugehörigen Baums zugegriffen werden kann, sind die ersten Einträge des Feldes reserviert. Diese Einträge werden auch für die zeichenweise erfolgende Umsetzung in Tokenwerte verwendet.

Die Prozeduren `Get2` bzw. `Get3` erkennen nach einer Initialisierung des Scanners Token in einem Quelltext. Während `Get2` Zwischenraum überliest, wird dieser von `Get3` als Token geliefert. Ein einheitlicher Aufruf ist durch die Prozedurvariable `Get` möglich. Als Seiteneffekt wird dabei die Position des erkannten Tokens im Quelltext der globalen Variablen `Pos` zugewiesen. Die Übergabe des erkannten Tokens erfolgt in `Tok`. Die Tokenwerte `eot`, `undef` und `whitespace` sind vordefiniert.

In den beiden oben beschriebenen Fällen (teilweise Rücksetzung bei Symbolen bzw. vollständige Rücksetzung bei Bezeichnern) erfolgt eine Pufferung der Eingabezeichen; ansonsten erfolgt das Lesen der Eingabe ungepuffert. Hierfür gibt es folgende Datenstrukturen.

```
VAR
  ChBuf: ARRAY 512 OF CHAR;
  PosBuf: ARRAY LEN(ChBuf) OF IO.Position;
  CurCh, NextCh: INTEGER;

PROCEDURE CopyBuf;
PROCEDURE GetCh(VAR Ch: CHAR);
PROCEDURE GetBufCh(VAR Ch: CHAR);
PROCEDURE GetPos;
```

Der Puffer wird in dem Feld `ChBuf` dargestellt. Anfang und Ende des Puffers werden durch die Indizes `CurCh` und `NextCh` markiert. Die zugehörigen Positionen für Fehlermeldungen werden

parallel im Feld `PosBuf` abgelegt.

Das puffernde Lesen der Eingabe ist durch die Prozedur `GetBufCh` implementiert, das ungepufferte Lesen durch die Prozedur `GetCh`. Beide Prozeduren lesen Zeichen vorrangig aus dem Puffer. Zur Rücksetzung der Eingabe muß nach pufferndem Lesen nur der Anfangsindex `CurCh` zurückgesetzt werden. Droht ein Überlauf, wird der Puffer durch die Prozedur `CopyBuf` an den Anfang des Feldes `ChBuf` verschoben. Zur Überprüfung reicht eine Abfrage pro Token aus, da die maximale Tokenlänge bekannt ist.

### 5.1.2 Der Generator

Die Erstellung eines Scanners besteht in dieser Implementierung nur aus der Parametrisierung eines festen Scannerrumpfes. Wie bei der Parsergenerierung werden die konstanten Teile aus einer Textdatei in den zu generierenden Scanner kopiert. An den Einfügemarken „`$`“ werden die variablen Teile eingefügt. Dies sind der Modulname, die Länge des längsten Token sowie die Anzahl der Token. In die Prozedur `BuildTree` des generierten Scanners werden ferner Initialisierungsaufrufe für alle Hyper-Terminale der EAG eingefügt, die dem Scanner deren textuelle Repräsentation und ihre Tokennummer bekanntmachen.

Aus der Implementierung des Scanners resultieren die folgenden Wohlgeformtheitsbedingungen für Tokenrepräsentationen:

Erlaubt ist nur

- ein einzelnes Leerzeichen,
- ein einzelnes einfaches oder doppeltes Anführungszeichen,
- eine nichtleere Folge von Groß- und Kleinbuchstaben sowie Ziffern,
- eine nichtleere Zeichenfolge, die keine Zeichen kleiner als das Leerzeichen und keine der bereits oben angeführten Zeichen enthält.

Alle Terminale der EAG werden auf diese Bedingungen hin überprüft und Verstöße als Fehler gemeldet; die Generierung des Scanners unterbleibt in diesem Fall.

## 5.2 Der Parsergenerator

Diese Implementierung der EAGen geht von der konzeptionellen Trennung von Syntaxanalyse und Affixauswertung aus. Die Syntaxanalyse erfolgt klassisch anhand der kontextfreien Grundgrammatik. Besondere Beachtung erfordern die EBNF-Konstrukte (Alternativen, Optionen und Wiederholungen), die aus Effizienzgründen durch entsprechende Anweisungen implementiert werden sollen. Andererseits werden formal Ableitungsbäume anhand der Transformation aus Abschnitt 2.2 eingeführt. Um einen statischen Ableitungsbaum für beliebige Evaluationsstrategien zu erstellen, muß der Parser um entsprechende Konstruktoranweisungen erweitert werden. Für eine eingeschränkte Evaluationsstrategie kann in den eingefügten Anweisungen direkt die Affixauswertung (Parameterberechnung) erfolgen. Im Fall eines erweiterten LL(1)-Parsers entspricht dies den handgeschriebenen Compilern nach der Methode des rekursiven Abstiegs, wobei der Ableitungsbaum dynamisch in Form von Prozeduraufrufen vorliegt. Damit kann nun ein Parser, der einen Ableitungsbaum erstellt, als spezieller Ein-Pass-Compiler generiert werden, dessen Spezifikation entsprechend der erwähnten Transformation automatisch erzeugt wird.

Für die Verschränkung von Syntaxanalyse und Affixauswertung im generierten Compiler sind Bottom-up-Verfahren ungeeignet, da eine effiziente Auswertung eine zu stark eingeschränkte Strategie zur Folge hat. Bereits die Meldung von Konflikten ist nicht leicht benutzerfreundlich zu gestalten, und die Behandlung von EBNF-Konstrukten erscheint eher problematisch.

### 5.2.1 Die generierten Parser

Wir generieren Parser nach der Methode des rekursiven Abstiegs mit einem Vorgriffssymbol in der Generatorzielsprache Oberon. Wie üblich wird für jedes (nicht anonyme) Nichtterminal eine Prozedur erzeugt. Der Evaluationscode wird vom Evaluatorgenerator direkt in diese eingebettet, was gegenüber der Auslagerung in eigene Prozeduren einen erheblichen Geschwindigkeitsvorteil verspricht. Ferner wird eine benutzerfreundliche Fehlerbehandlung implementiert. Wegen ihrer Auswirkung auf die Parserstruktur wird diese zuerst vorgestellt.

#### 5.2.1.1 Fehlerbehandlung

Die Parser verfügen über eine automatische Fehlerbehandlung nach einem Implementierungsvorschlag von Grosch [Grosch] in Anlehnung an das Verfahren von Röhrich [Röhrich]. Alle dafür benötigten Informationen können vom Generator aus der Grammatik abgelesen und in den Parser eingearbeitet werden; es sind keine zusätzlichen Angaben durch den Spezifikateur erforderlich. Syntaxfehler werden nicht nur erkannt und gemeldet, sondern auch gemäß der Grammatik repariert. Der Evaluationscode kann sich daher stets auf formal korrekte Ableitungsbäume verlassen.

Während der LL(1)-Syntaxanalyse sind als Fehlersituationen die Fälle zu behandeln, daß bei Analyse eines Terminals das aktuelle Eingabetoken verschieden vom erwartenen Terminal ist, daß bei Analyse von Alternativen das Eingabetoken in keiner Direktormenge enthalten ist sowie daß bei Analyse von Optionen bzw. Wiederholungen das Eingabetoken weder in der First- noch in der Followmenge des Konstrukts enthalten ist. Für eine gute Fehlerbehandlung muß der Test auf Enthaltensein in der Followmenge rechtzeitig durchgeführt werden, so daß die Option oder Wiederholung noch zur Reparatur genutzt werden kann.

Bei Auftreten eines Syntaxfehlers wird die Fehlerposition zusammen mit einer Teilmenge der Token gemeldet, die eine korrekte Eingabefortsetzung wären (Expectedmenge). Anschließend wird die Menge aller Token berechnet, die als Wiederaufsetzpunkte dienen können (Recoverymenge). Die kürzeste Tokenfolge bis zu einem dieser Token wird überlesen und der Wiederaufsetzpunkt gemeldet. Das Parsen wird nun im Reparaturmodus fortgesetzt. Der Parser mitsamt Affixauswertungsanweisungen verhält sich dabei wie gewöhnlich, nur werden keine Token von der Eingabe gelesen, sondern es wird eine „einfache“ Tokenfolge als Einfügung ermittelt und gemeldet. Der Parser verbleibt in diesem Modus, bis das aktuelle Eingabetoken akzeptiert werden kann. Der Eingabetext kann als repariert angesehen werden, wenn die überlesenen Token durch die als eingefügt gemeldeten ersetzt werden; ein Tool kann diese Modifikationen wirklich durchführen.

Sowohl die exakten Followmengen als auch die Expected- und Recoverymengen sind abhängig vom Kontext und somit von der dynamischen Aufrufhierarchie. Zu ihrer Bestimmung ist es nötig, während des Parsens darüber Informationen mitzuführen. Für die Analyse korrekter Eingaben genügt als (zu große) Followmenge aber die Vereinigung über alle Kontexte. Bei fehlerhaften Eingaben werden Fehler dann zwar textuell noch so früh wie möglich erkannt, der Parserzustand kann aber bereits so weit fortgeschritten sein, daß nicht mehr alle Fortsetzungen möglich sind und nur eine Teilmenge der exakten Expectedmenge gemeldet werden kann.

Aus Effizienzgründen werden wie üblich die zu großen Followmengen verwendet. Ferner werden nur die Teilmengen der erwarteten Token gemeldet, die bereits zur Generierungszeit bestimmt werden können; diese Mengen werden vom Generator für jede mögliche Fehlerposition berechnet.

Die Recoverymengen aber müssen für ein sinnvolles Verhalten der Fehlerbehandlung exakt sein. Ihre Berechnung kann einen beträchtlichen Teil der Laufzeit beanspruchen. Aus Effizienzgründen wird daher bereits zur Generierungszeit für jeden Faktor aller Alternativen eine regellokale Recoverymenge vorberechnet und im generierten Parser in einem Feld gespeichert. Zur Laufzeit des Parsers wird ein Laufzeitkeller nachgebildet, der für jede Prozedurinkarnation der jeweiligen Aufrufhierarchie den Index der entsprechenden regellokalen Recoverymenge in diesem Feld enthält. Die globale Recoverymenge besteht aus der Vereinigung der lokalen Recoverymengen. Solange kein Fehler auftritt, genügt es aber, vor dem Aufruf einer Prozedur, die ein Nichtterminal analysiert, den Index der zugehörigen lokalen Recoverymenge auf den Keller zu legen und hinterher wieder zu entfernen. Die aufwendige Berechnung der Vereinigung wird so verzögert und nur im Fehlerfall durchgeführt.

Die Fehlerbehandlung repariert jede inkorrekte Eingabe durch Überlesen und Einfügen von Token in eine syntaktisch korrekte. Die Token werden dabei ganz einfach dadurch eingefügt, daß das Parsen fortgesetzt wird, als läge gar kein Fehler vor. Wird ein Terminal erwartet, das vom aktuellen Eingabetoken verschieden ist, so wird es als eingefügt gemeldet. Wird eine Alternative analysiert und das Eingabetoken ist in keiner Direktormenge enthalten, so wird eine vom Generator festgelegte Alternative ausgewählt. Die Beschränkung auf nichtrekursive Alternativen garantiert die Termination des Verfahrens. Die Evaluations- bzw. Baumkonstruktoranweisungen werden während der Fehlerreparatur wie gewöhnlich ausgeführt.

Die beiden Prozeduren `RecoveryTerminal` und `ErrorRecovery` dienen der Einleitung der Fehlerreparatur bei Analyse eines Terminals bzw. eines EBNF-Konstrukts. Sie geben eine Fehlermeldung aus, berechnen die globale Recoverymenge, überlesen Token bis zum Wiederaufsetzpunkt und schalten schließlich in den Reparaturmodus um. Im Reparaturmodus werden keine Fehler mehr gemeldet oder Token überlesen, stattdessen werden nur die eingefügten Token gemeldet. Sobald die Analyse eines Terminals erfolgreich ist, ist die Fehlerbehandlung abgeschlossen, und dieser Modus wird wieder verlassen.

#### 5.2.1.2 Überblick über die generierten Parser

Die generierten Parser bestehen aus einem Modul. Die Struktur ist aus dem Programmfragment ersichtlich. Die wesentlichen Prozeduren der Fehlerbehandlung sind mit angegeben.

```
MODULE Parsername;
IMPORT Scanner := Scannername;
CONST  tokSetLen = ...;
       firstRecStack = ...;

TYPE TokSet = ARRAY tokSetLen OF SET;

(* globale Datenstrukturen *)
VAR Tok : INTEGER;
    Set : ARRAY ... OF TokSet;
    RecStack : POINTER TO ARRAY OF INTEGER; RecTop : INTEGER;
    IsRepairMode : BOOLEAN;
    ...

(* globale Prozeduren des Parsers und des eingefügten Evaluators *)
PROCEDURE SkipTokens(Recover : INTEGER);
  VAR GlobalRecoverySet : TokSet; i, j : INTEGER;
```

```
BEGIN
  GlobalRecoverySet := Set[Recover];
  FOR i := firstRecStack TO RecTop - 1 DO
    FOR j := 0 TO tokSetLen - 1 DO
      GlobalRecoverySet[j] := GlobalRecoverySet[j] +
                              Set[RecStack[i]][j]
    END
  END;
  WHILE ~ (Tok MOD (MAX(SET) + 1) IN
        GlobalRecoverySet[Tok DIV (MAX(SET) + 1)]) DO
    Scanner.Get(Tok)
  END;
  RestartMessage(Scanner.Pos);
  IsRepairMode := TRUE
END SkipTokens;

PROCEDURE ErrorRecovery(Expected, Recover : INTEGER);
BEGIN
  IF ~ IsRepairMode THEN
    ErrorMessageTokSet(Scanner.Pos, Set[Expected]);
    SkipTokens(Recover)
  END
END ErrorRecovery;

PROCEDURE RecoveryTerminal(ExpectedTok, Recover : INTEGER);
BEGIN
  IF ~ IsRepairMode THEN
    ErrorMessageTok(Scanner.Pos, ExpectedTok);
    SkipTokens(Recover)
  END;
  IF Tok # ExpectedTok THEN RepairMessage(Scanner.Pos, ExpectedTok)
  ELSE IF Tok # endTok THEN Scanner.Get(Tok) END; IsRepairMode := FALSE
  END
END RecoveryTerminal;
...

PROCEDURE P0(VAR V1 : HeapType);   (* Startsymbol *)
... (* Analysiert die Eingabe, benutzt weitere Prozeduren *)
END P0;

(* ... weitere Prozeduren für Nichtterminale: P1, P2, P3, ... *)
...

PROCEDURE Emit(Ptr : HeapType);
... (* Gibt die Übersetzung der Eingabe aus *)
END Emit;

PROCEDURE Compile*;
  VAR V1 : HeapType;
BEGIN
  EvalInit; ParserInit; Scanner.Init; Scanner.Get(Tok);
  P0(V1); Emit(V1)
END Compile;
```

```
BEGIN ReadParserTab("Parsername.Tab")
END Parsername.
```

Die benötigten First-, Follow-, Expected- und Recoverymengen werden im Modulrumpf gleich beim Laden des Parsers aus einer Parsertabelle eingelesen. Exportiert wird nur die parameterlose Prozedur `Compile` (Kommando), die nach Initialisierungen die Prozedur des Startsymbols der Grundgrammatik aufruft. Diese Prozedur analysiert die gesamte Eingabe; der eingebundene Evaluatorcode überprüft während des Parsens durch Affixberechnungen die statische Semantik und erzeugt die Übersetzung der Eingabe, die durch die Variable `V1` der Ausgabeprozedur übergeben wird.

#### 5.2.1.3 Codeschemata

Die Umsetzung von Symbolvorkommen und EBNF-Konstrukten auf rechten Regelseiten in Oberon-Anweisungen läßt sich durch folgende Codeschemata beschreiben:

```
(* Terminal t *)
IF Tok # t THEN RecoveryTerminal(t, lokale Recoverymenge von t)
ELSE Scanner.Get(Tok); IsRepairMode := FALSE
END;

(* Nichtterminal N *)
Synthese der Eingabeparameter zu N
IF RecTop >= LEN(RecStack^) THEN ParserExpand END;
RecStack[RecTop] := lokale Recoverymenge von N; INC(RecTop);
N(...);
DEC(RecTop);
Analyse der Ausgabeparameter zu N

(* Prädikat N *)
Synthese der Eingabeparameter zu N
IF ~ N(...)  THEN Fehlerbehandlung END;
Analyse der Ausgabeparameter zu N

(* Alternative A = A1 | ... | An *)
```

```
LOOP
  CASE Tok OF
  | Direktormenge von A_1 :
      Analyse der Eingabeparameter der linken Seite von A_1
      Code für A_1
      Synthese der Ausgabeparameter der linken Seite von A_1
      EXIT
       ⋮
  | Direktormenge von A_n :
      Analyse der Eingabeparameter der linken Seite von A_n
      Code für A_n
      Synthese der Ausgabeparameter der linken Seite von A_n
      EXIT
  ELSE
    IF IsRepairMode THEN
      Analyse der Eingabeparameter der linken Seite von A_default
      Code für A_default
      Synthese der Ausgabeparameter der linken Seite von A_default
      EXIT
    END;
    ErrorRecovery(Expectedmenge von A, lokale Recoverymenge von A)
  END
END;

(* Option N = [X] *)
LOOP
  IF Tok ∈ first(X) THEN Code für X; EXIT
  ELSIF Tok ∈ follow(N) OR IsRepairMode THEN
    Analyse der Eingabeparameter für den Abbruch
    Synthese der Ausgabeparameter für den Abbruch
    EXIT
  END;
  ErrorRecovery(Expectedmenge von N, lokale Recoverymenge von N)
END;

(* Wiederholung N = {X} *)
LOOP
  IF Tok ∈ first(X) THEN Code für X
  ELSIF Tok ∈ follow(N) OR IsRepairMode THEN EXIT
  ELSE ErrorRecovery(Expectedmenge von N, lokale Recoverymenge von N)
  END
END;
Analyse der Eingabeparameter für den Abbruch
Synthese der Ausgabeparameter für den Abbruch
```

Die LOOP-Anweisung ist nur im Falle der Wiederholung eine echte Schleife, sie dient ansonsten nur dazu, den Rumpf nicht zweimal hintereinander schreiben zu müssen. Bei den Standardalternativen ist diese Verdoppelung jedoch nicht vermieden und kann auch geschachtelt auftreten.

Vorkommen von nur nach leer ableitbaren Nichtterminalen – die in der Grundgrammatik weggelassenen sogenannten Prädikate einer EAG – auf rechten Regelseiten werden in Aufrufe von booleschen Funktionsprozeduren umgesetzt, die der Evaluatorgenerator erstellt. Zur kontextfreien Definition der Quellsprache tragen Prädikate nichts bei; das Codeschema ist hier nur der Vollständigkeit halber angegeben.

An vielen Stellen bieten sich von obigen Schemata abweichende Optimierungsmöglichkeiten an.

So stehen die Oberon-Anweisungen für Terminale häufig im Kontext von Alternativen, Optionen oder Wiederholungen. Garantieren deren Token-Abfragen, daß zur Laufzeit des Parsers an einer Programmstelle nur das erwartete Token oder auch niemals das erwartete Token möglich ist, so kann die IF-Abfrage des Terminalschemas dort weggelassen werden; entweder der THEN-Teil oder der ELSE-Teil wird einfach direkt ausgeführt.

In der Umsetzung von Nichtterminalen muß der Laufzeitkeller nur beim ersten Vorkommen eines Nichtterminals in einer Alternative auf Überlauf getestet werden. Stehen zudem zwei oder mehr Nichtterminale in einer Alternative direkt hintereinander, so können auch die aufeinanderfolgenden Dekrement- und Inkrementoperationen vermieden werden.

Oberon-Anweisungen für Alternativen brauchen nur für echte Alternativen mit mindestens zwei Fällen vorhanden sein. Die LOOP-Schleife sowie der ELSE-Teil der CASE-Anweisung sind ferner nur dann nötig, wenn es zur Laufzeit an dieser Stelle möglich ist, daß das aktuelle Eingabetoken in keiner der Direktormengen enthalten ist.

Bei Optionen und Wiederholungen kann der Test, ob das Eingabetoken in der First- oder Followmenge enthalten ist, für kleine Mengen (insbesondere einelementige) als direkter Vergleich codiert werden.

#### 5.2.1.4 Speicherung der First- und Followmengen

In den Oberon-Anweisungen für Optionen und Wiederholungen wird getestet, ob das aktuelle Eingabetoken in der First- oder Followmenge enthalten ist. Zu jedem Nichtterminal *N* werden diese Mengen üblicherweise in einem `ARRAY OF SET` gespeichert; ein Test auf Enthaltensein in beispielsweise der Firstmenge wird dann folgendermaßen umgesetzt:

```
First : ARRAY MaxNont, MaxTok DIV 32 + 1 OF SET;
Tok ∈ first(N)   ≡   (Tok MOD 32) IN First[N][Tok DIV 32]
```

Dieser Ausdruck erfordert zwei Divisionen zur Laufzeit. Grosch [Grosch] bemerkt, daß es wesentlich vorteilhafter ist, nicht zu jedem Nichtterminal seine Firstmenge zu speichern, sondern zu jedem Token die Menge der Nichtterminale, in deren Firstmenge es enthalten ist. Dies entspricht dem Transponieren der Relation $first \subseteq [0 \ldots MaxNont] \times [0 \ldots MaxTok]$. Daraus resultiert die folgende Umsetzung:

```
First : ARRAY MaxNont DIV 32 + 1, MaxTok OF SET;
Tok ∈ first(N)   ≡   (N MOD 32) IN First[N DIV 32][Tok]
```

Die beiden Divisionen sind jetzt konstant und können bereits vom Generator durchgeführt werden. Die Speicherung der Followmengen erfolgt analog.

### 5.2.2 Der Generator

Der Parsergenerator gliedert sich in folgende Teilaufgaben:

1. Berechnung der First-, Follow- und daraus der Direktormengen. Hierbei werden alle Konflikte und Fehler entdeckt und gemeldet.
2. Sammlung von Informationen für die Fehlerbehandlung.

   (a) Bestimmung der Fortsetzungsgrammatik für die Fehlerkorrektur.

   (b) Berechnung der Expected- und Recoverymengen für Fehlermeldungen und das Finden von Wiederaufsetzpunkten nach Fehlern.

3. Generierung des Parsermoduls.

4. Erzeugung einer Parsertabelle (Hilfsdatei) unter Benutzung oben berechneter Daten.

Da uns ein geeigneter Scannergenerator nicht zur Verfügung steht, haben wir als provisorische Lösung favorisiert, den Parser nicht nur den kontextfreien, sondern auch den regulären Sprachanteil einlesen zu lassen, wie es in einfacher Form auch schon unter Eta praktiziert wurde. Dafür wird ein parametrisierbarer Scanner benutzt, der aber lediglich alle Terminale als Token erkennt und einen Eingabetext anhand fester Regeln in eine Folge solcher Token zerlegt. Die eigentlich regulären Sprachanteile, wie beispielsweise Namen oder Zahlen, werden vom Parser als Folge von Buchstabentoken bzw. Zifferntoken erkannt. Zwischenraum (Leerzeichen, Zeilenwechsel, Tabulatoren, ...) wäre daher innerhalb dieser Sprachanteile erlaubt und könnte nicht zur Trennung der intendierten Token verwendet werden. Um dieses Problem zu beheben, wurden Markierungen eingeführt, mit denen Nichtterminale vom Spezifikateur als Token gekennzeichnet werden können. Scanner und Parsergenerator wurden dahingehend erweitert, daß die als Token markierten Nichtterminale durch Zwischenraum beendet werden und einige dadurch überflüssig gewordene Konfliktmeldungen unterdrückt werden. Auch zusätzliche Wohlgeformtheitsbedingungen für diese Nichtterminale werden im Parsergenerator überprüft. Das Verhalten der generierten Parser ist nun in vielen praktisch relevanten Fällen nicht mehr von einer echten Scanner-Parser-Kombination zu unterscheiden.

#### 5.2.2.1 Datenstrukturen

Die folgenden globalen Datenstrukturen werden im Verlauf der Generierung gefüllt und schließlich zur Erstellung von Parsermodul und -tabelle verwendet:

```
Nont : POINTER TO ARRAY OF RECORD
         First, Follow, IniFollow : Sets.OpenSet;
         DefaultAlt : EAG.Alt;
         Edge : INTEGER;
         AltRec, OptRec, AltExp, OptExp : INTEGER;
         FirstIndex, FollowIndex : INTEGER;
         Anonym : BOOLEAN
       END;

Alt : POINTER TO ARRAY OF RECORD
        Dir : Sets.OpenSet
      END;

Factor : POINTER TO ARRAY OF RECORD
           Rec : INTEGER
         END;

Edge : POINTER TO ARRAY OF RECORD
         Dest, Next : INTEGER
       END;
NextEdge : INTEGER;
```

```
GenSet  : POINTER TO ARRAY OF Sets.OpenSet; NextGenSet : INTEGER;
GenSetT : POINTER TO ARRAY OF Sets.OpenSet; NextGenSetT : INTEGER;
```

In den Feldern `Nont`, `Alt` und `Factor` liegen Informationen zu jedem Nichtterminal, jeder Alternative und jedem Faktor der Grundgrammatik vor. `Nont` ist als Parallelfeld zu `EAG.HNont` angelegt, bei Alternativen und Faktoren ist die `Ind`-Komponente dieser EAG-Datenstrukturen der Index der zugehörigen Informationen.

Zu jedem Nichtterminal wird seine First- und Followmenge sowie eine initiale Followmenge gespeichert, die in die Expectedmenge einfließt. Als Informationen für die Fehlerkorrektur werden eine Standardalternative (`DefaultAlt`) und gegebenenfalls die Indizes der First- und der Followmenge sowie der lokalen Recovery- und der Expectedmengen für die EBNF-Konstrukte Alternative und Option bzw. Wiederholung gespeichert. Die im generierten Parser benötigten Mengen werden im Generator in den Feldern `GenSet` und `GenSetT` gespeichert und letztlich in die Parsertabelle geschrieben. Die Komponenten `AltRec`, `OptRec`, `AltExp`, `OptExp` sowie `FirstIndex` und `FollowIndex` verweisen in diese Felder und werden in den generierten Parsercode als feste Zahlen eingetragen.

Die Komponente `Edge` in `Nont` dient zur Speicherung auslaufender Kanten in entsprechenden Graphen; `Edge` ist der Index des ersten Listeneintrags im Feld `Edge`, in dem die Adjazenzlisten eines (Multi-) Graphen repräsentiert werden. `Anonym` ist nur eine leichter lesbare Kennzeichnung anonymer Nichtterminale (`EAG.HNont[N].Id < 0`).

Zu Alternativen wird ihre Direktormenge gespeichert, zu Faktoren der Index der lokalen Recoverymenge.

Weiterhin dienen folgende Variablen zur Steuerung des Generators:

```
TestNonts, GenNonts, RegNonts, ConflictNonts : Sets.OpenSet;
nToks : INTEGER;
Error, Warning, ShowMod, Compiled, UseReg : BOOLEAN;
```

Die Mengen `TestNonts` und `GenNonts` ($\subseteq$ `TestNonts`) enthalten die Nichtterminale, für die die LL(1)-Tests durchgeführt und eventuelle Konflikte gemeldet werden bzw. für die im Parser Code generiert wird. Der Hintergrund ist, daß die Generierung auf die produktiven und erreichbaren Nichtterminale der Grundgrammatik beschränkt sein sollte, für die insbesondere nichtrekursive Alternativen garantiert sind. Die Variable `nToks` enthält die Anzahl der Token; alle Tokenmengen werden dynamisch in dieser Größe angelegt. Die Tokennummern werden von Null an aufsteigend für die speziellen Token „Eingabeende“, „Undefiniert“ und „Zwischenraum“ (`endTok`, `undefTok`, `sepTok`) und daran anschließend für die Terminale der Grundgrammatik vergeben.

`Error`, `Warning`, `ShowMod`, `Compiled` und `UseReg` steuern das Verhalten des Parsergenerators. Nur ein aufgetretener Fehler verhindert die Generierung eines Parsers. `ShowMod` steuert, ob der generierte Parser angezeigt oder gleich compiliert wird; `Compiled` wird benötigt, um die Meldungsausgabe mit dem dazu verwendeten Compiler abzustimmen. `UseReg` schaltet die Modifikationen des Generators zur Beseitigung der Probleme mit Zwischenraum ein. Die Menge `RegNonts` enthält dann die regulär zu handhabenden Nichtterminale und `ConflictNonts` die Teilmenge davon, bei denen einige Konflikte am Ende unterdrückt werden.

Die Prozedur `Init` legt all diese Strukturen an und initialisiert sie, die Prozedur `Finit` gibt die dynamisch angelegten Daten zum Schluß wieder frei.

Expandierbar sind die Felder `Edge`, `GenSet` und `GenSetT`, deren Größe nicht a priori feststeht.

#### 5.2.2.2 Berechnung der First-, Follow- und Direktormengen

Die Berechnung dieser Mengen erfolgt in der Prozedur `ComputeDir`, zuerst die der Firstmengen, dann die der Followmengen, wobei die initialen Followmengen ebenfalls gespeichert werden. Abschließend werden aus diesen Mengen die Direktormengen für die Alternativen erstellt.

**Die Firstmengen** Die Firstmenge eines Nichtterminals $N$ ist die Menge aller Token, mit denen ein aus $N$ abgeleitetes Wort beginnen kann. Da Linksrekursion verboten ist, können Firstmengen durch eine rekursive Prozedur berechnet werden, wobei zuerst die Firstmengen der Faktoren am Anfang aller Alternativen zu bestimmen sind. Eine Markierung zur Verhinderung von mehrfachen Berechnungen kann auch zur Erkennung von Linksrekursion mitbenutzt werden.

Wir haben uns für eine mächtigere Lösung entschieden, die Linksrekursion nicht nur entdeckt, sondern die Meldung des gesamten linksrekursiven Zyklus erlaubt und sogar in diesem Falle korrekte Firstmengen berechnet.

Dafür wird ein gerichteter Graph aufgebaut, dessen Knoten alle Nichtterminale der Menge `TestNonts` sind; eine Kante verläuft von einem Nichtterminal $N_1$ nach $N_2$ gdw. $N_2$ am Anfang einer Alternative von $N_1$ steht, was bedeutet, daß alle Faktoren vor $N_2$ leerableitbare Nichtterminale sind.

Auf diesen Graphen wird eine Erweiterung des SCC-Algorithmus zur Bestimmung streng zusammenhängender Komponenten angewendet. Eine direkte Linksrekursion ist an einer Schlinge erkennbar, eine indirekte Linksrekursion führt zu einer nicht-trivialen streng zusammenhängenden Komponente. Diese Fälle werden erkannt und die gesamte streng zusammenhängende Komponente wird in einer Fehlermeldung ausgegeben, also alle an der Linksrekursion beteiligten Nichtterminale; gleichzeitig wird damit die Generierung eines Parsers unterdrückt.

Zusätzlich wird die Firstmenge jedes Nichtterminals durch Vereinigung seiner initialen Firstmenge mit den beim Wiederaufstieg berechneten Firstmengen der direkten Nachfolger berechnet. Für nichttriviale Komponenten erfolgt eine Nachbehandlung. Die initiale Firstmenge eines Nichtterminals ist die Menge der jeweils ersten Terminale seiner Alternativen, sofern diese in obigem Sinne am Anfang stehen.

Die Adjazenzlisten des Graphen werden im Feld `Edge` repräsentiert. Die Wurzeln dieser Listen sind in die `Edge`-Komponenten im Feld `Nont` eingetragen. Um die lineare Suche beim Einfügen von Kanten zu vermeiden, wird der Graph als Multigraph realisiert, d.h. Mehrfachkanten sind erlaubt.

**Die Followmengen** Die Followmenge eines Nichtterminals $N$ ist die Menge aller Token, die gemäß der Grammatik hinter einem aus $N$ abgeleiteten Wort stehen können. Zur Berechnung der Followmengen verwenden wir den gleichen Algorithmus wie für die Firstmengenberechnung, nur repräsentiert der Graph eine andere Relation, und Zyklen sind hier erlaubt.

Wieder sind alle Nichtterminale der Menge `TestNonts` die Knoten des Graphen; eine Kante verläuft diesmal aber von einem Nichtterminal $N_1$ nach $N_2$ gdw. $N_1$ am Ende einer Alternative von $N_2$ steht, was bedeutet, daß alle Faktoren hinter $N_1$ leerableitbare Nichtterminale sind.

Die initiale Followmenge eines Nichtterminals $N$ besteht aus allen Token, mit denen ein Wort beginnen kann, das aus einem in irgendeiner Regel der Grammatik hinter $N$ stehenden Faktor abgeleitet werden kann, wobei dazwischen nur leerableitbare Nichtterminale stehen dürfen. Gegeben die Firstmengen, lassen sich die initialen Followmengen in einem Durchlauf über die Grammatik bestimmen, wenn die rechten Regelseiten von rechts nach links gelesen werden (dazu gibt es eine entsprechende Verkettung). Diese initialen Followmengen werden in der Komponente `IniFollow` gespeichert. Diese eingeschränkte Followmenge berücksichtigt die grammatikglobalen

Zusammenhänge nicht, die durch Vorkommen von Nichtterminalen am Ende von Alternativen entstehen.

Die Berechnung der Followmengen erfolgt nun analog zu der der Firstmengen.

**Die Direktormengen** Die Direktormengen werden vom Parser dazu benutzt, um in Abhängigkeit vom aktuellen Eingabetoken zu einem Nichtterminal die passende Alternative zu wählen. Sie werden aus am Anfang von Alternativen stehenden Terminalen bzw. im Fall von Nichtterminalen aus deren Firstmengen gebildet. Für leerableitbare Alternativen wird noch die Followmenge des zugehörigen Nichtterminals beigefügt.

Verstöße gegen die LL(1)-Bedingung außer der schon oben erkannten Linksrekursion werden hierbei entdeckt, als Warnung gemeldet und aufzulösen versucht. Haben mehrere Alternativen nicht disjunkte Direktormengen, so wird im Konfliktfall die erste gewählt. Eine Option oder Wiederholung verursacht einen Konflikt, wenn sich ihre Followmenge mit den Direktormengen ihrer Alternativen überschneidet. Hier wird im Konfliktfall die Option gewählt bzw. die Wiederholung weiter ausgeführt.

Die geringe Mächtigkeit des LL(1)-Verfahrens kann durch diese geregelte Konfliktbehandlung etwas unterstützt werden. Sie ergibt sich in dieser Form aus dem üblichen Implementierungsschema und wird im Generator wie folgt realisiert: Überschneidet sich eine Direktormenge mit der Vereinigung der Direktormengen der vorherigen Alternativen, so wird in der Prozedur `Conflict` ein Direktormengenkonflikt gemeldet. Die problematische Schnittmenge wird ebenfalls ausgegeben und anschließend aus der Direktormenge entfernt. Wird diese dadurch zur leeren Menge, so bedeutet das, daß die zugehörige Alternative unerreichbar würde bzw. die Option immer ausgeführt würde oder die Wiederholung (für korrekte Eingaben) nicht abgebrochen werden könnte; diese Fälle werden als Entwurfsfehler gedeutet, und eine Parsergenerierung wird dann unterdrückt.

#### 5.2.2.3 Bestimmung der Fortsetzungsgrammatik

Wenn im Reparaturmodus eine Alternative analysiert werden soll und das Eingabetoken in keiner Direktormenge enthalten ist, so muß der Parser willkürlich oder unter Berücksichtigung der konkreten Eingabe eine der Alternativen zur Fortsetzung auswählen. Solange die erwarteten Token nun vom aktuellen Eingabetoken abweichen, werden sie als eingefügt gemeldet. Damit diese Einfügungen stets terminieren, darf dabei die im Fehlerfall gewählte Alternative aber nicht zu einem Zyklus führen.

Aus diesem Grund bestimmt bereits der Generator zu jedem Alternativenkonstrukt der Grundgrammatik eine geeignete Standardalternative für den Reparaturmodus und trägt diese fest in den generierten Parser ein. Die Grammatik, die nur aus den Standardalternativen besteht, ist die sogenannte Fortsetzungsgrammatik; sie erzeugt zu jedem Nichtterminal genau ein Wort, das im Reparaturmodus für dieses Nichtterminal eingefügt wird (für Ausnahmen siehe Abschnitt 5.2.2.4).

In einer reduzierten Grammatik gibt es immer geeignete nichtrekursive Alternativen, aber diese sind nicht eindeutig bestimmt. Ihre Auswahl beeinflußt jedoch in sehr hohem Maße das Verhalten des Parsers bei Eingabefehlern. Um dem Spezifikateur die Möglichkeit zu geben, diese Wahl zu beeinflussen, werden den Alternativen in der Reihenfolge ihres textuellen Auftretens Prioritäten zugewiesen. Die erste Alternative eines Nichtterminals erhält die höchste Priorität, die letzte die geringste. Unter den Fortsetzungsgrammatiken wird nun eine solche gewählt, bei der für jedes Nichtterminal eine Standardalternative mit möglichst hoher Priorität gewählt ist. Insbesondere wird für alle Nichtterminale die erste Alternative gewählt, wenn dies möglich ist. Werden in Spezifikationen die Alternativen stets in der Reihenfolge wachsender „Komplexität“ angegeben, so führt dies bei den generierten Parsern im Fehlerfall zur Einfügung „einfacher“ Tokenfolgen zur Reparatur.

Zur Bestimmung der Standardalternativen wird eine Modifikation des Algorithmus zur Berechnung leerableitbarer Nichtterminale verwendet (vgl. Abschnitt 4.2.3). Die effiziente Durchführung dieses Verfahrens erfordert geeignete erweiterte Datenstrukturen:

```
Alt : POINTER TO ARRAY OF RECORD
        Nont, Deg, Prio : INTEGER;
        Alt : EAG.Alt
      END;
Stack : POINTER TO ARRAY OF RECORD
          Nont, APrio : INTEGER;
          Alt : EAG.Alt
        END;
Top : INTEGER;
StackPos : POINTER TO ARRAY OF INTEGER;
```

Die Menge der Nichtterminale mit einer „gelöschten“ Alternative wird in `Stack` als Keller realisiert. Jeder Eintrag besteht aus einem Nichtterminal, einer zugehörigen Alternative, die ein Kandidat für die Standardalternative ist, sowie deren Priorität.

Im Feld `Alt` wird zu jeder Alternative das zugehörige Nichtterminal, der aktuelle Grad, die Priorität und ein Verweis zurück auf die Alternative gespeichert; diese Informationen werden benötigt, um bei „Löschung“ einer Alternative das zugehörige Nichtterminal in `Stack` aufzunehmen. Die Alternativenpriorität steuert die Wahl des Nichtterminals, das aus `Stack` entfernt und dessen Standardalternative damit festgelegt wird. Bei Aufnahme von Nichtterminalen in `Stack` wird die Alternativenpriorität maximiert.

Um schnellen Zugriff auf den Index eines Nichtterminals in `Stack` zu haben, wird dieser im Feld `StackPos` mitprotokolliert. Die Initialisierung `MAX(INTEGER)` zeigt dabei an, daß das Nichtterminal noch nicht in `Stack` aufgenommen ist, der Wert `-1`, daß es schon eine Standardalternative besitzt. Das Entfernen eines Nichtterminals von einer beliebigen Position in `Stack` wird durch Umkopieren des obersten Elements erreicht. Das Feld `StackPos` ist dabei entsprechend zu aktualisieren.

Die Verweise von Nichtterminalen auf die Alternativen, in denen sie vorkommen, werden in dem schon früher verwendeten globalen Feld `Edge` angelegt.

Eine besondere Beachtung erfordern die EBNF-Konstrukte Option und Wiederholung. In ihren Codeschemata ist der Abbruch, also die leere Alternative gemäß der Transformation aus Abschnitt 2.2, fest als Standardalternative vorgegeben. Die entsprechenden Nichtterminale benötigen daher keine explizite Standardalternative und können auf rechten Regelseiten einfach ignoriert werden.

Da dieses Verfahren genau die produktiven Nichtterminale berechnet, nur eben in einer speziellen Reihenfolge, terminiert es für jede reduzierte Grundgrammatik erfolgreich, d.h. für alle Nichtterminale außer denen für Optionen und Wiederholungen wird eine Standardalternative gewählt.

#### 5.2.2.4 Berechnung der Expected- und lokalen Recoverymengen

Die generierten Parser geben bei der Erkennung eines Syntaxfehlers eine Menge erwarteter Token aus. Aus Effizienzgründen werden hier nur die Teilmengen gemeldet, die schon zur Generierungszeit bestimmt werden können. Diese sogenannten Expectedmengen müssen vom Generator für alle möglichen Fehlerstellen berechnet werden.

Für ein Terminal ist gerade dieses das einzige erwartete Token; es kann in den Prozeduraufruf zur Fehlermeldung fest eingetragen werden und muß deshalb nicht separat gespeichert werden.

Für Alternativen (mit mindestens zwei Fällen) besteht die Expectedmenge aus der Menge der möglichen Token am Anfang der Alternativen, also der Firstmenge des zugehörigen Nichtterminals (nicht der Vereinigung der Direktormengen). Für Optionen und Wiederholungen besteht die Expectedmenge aus ihrer First- und initialen Followmenge. Eine Ausnahme bilden benannte Optionen und Wiederholungen. Da sie mehrfach auf rechten Regelseiten vorkommen können, darf als Expectedmenge nur ihre Firstmenge verwendet werden, damit bei einem Fehler nicht solche Token mitgenannt werden, die nur in einem anderen Kontext eine korrekte Programmfortsetzung wären.

Nach der Meldung der Expectedmenge überlesen die generierten Parser eine Tokenfolge bis zum sogenannten Wiederaufsetzpunkt. Die Menge derjenigen Token, die als Wiederaufsetzpunkt dienen können, wird als Recoverymenge bezeichnet. Das Verständnis der Definition dieser Menge erfordert die Kenntnis des Parserzustands bei Erkennung eines Syntaxfehlers:

Im Falle einer Grammatik ohne EBNF-Konstrukte wurde die Analyse der Eingabe durch den Aufruf der für das Startsymbol $N_1$ erzeugten Prozedur $P_1$ eingeleitet, die eine bestimmte Grammatikregel analysiert. Ein Nichtterminal $N_2$ in dieser Regel führte zum Aufruf der Prozedur $P_2$. Die Analyse weiterer Regeln bewirkte dann an den entsprechenden Nichtterminalen $N_3, \ldots, N_n$ die Aufrufe der Prozeduren $P_3, \ldots, P_n$. Die Prozedur $P_n$ entdeckt nun bei der Analyse des Faktors $F$ einen Syntaxfehler. Die entstandene Aufrufhierarchie läßt sich folgendermaßen veranschaulichen:

$$
\begin{array}{llllll}
P_1 & \textit{analysiert} & N_1 & \rightarrow \cdots & N_2 & \cdots . \\
P_2 & \textit{analysiert} & N_2 & \rightarrow \cdots & N_3 & \cdots . \\
 & \vdots & & \vdots & \vdots & \\
P_n & \textit{analysiert} & N_n & \rightarrow \cdots & F & \cdots .
\end{array}
$$

Wird nur die letzte Regel betrachtet, so könnte die Analyse des Parsers bei allen rechts von $F$ stehenden Faktoren fortgesetzt werden. Die lokale Recoverymenge $REC_n$ ist daher die Vereinigung der Firstmengen dieser Faktoren (die Firstmenge eines Terminals ist dieses selbst).

Im allgemeinen führt die Beschränkung auf die lokale Recoverymenge $REC_n$ aber zum Überlesen sehr großer Eingabeteile. Es müssen also auch die übrigen in Analyse befindlichen Regeln berücksichtigt werden; das Parsen kann auch bei allen hier noch nicht analysierten Faktoren fortgesetzt werden. Die zugehörigen Recoverymengen $REC_i \quad (i = 1, \ldots, n-1)$ sind dementsprechend die Vereinigungen der Firstmengen aller rechts von $N_{i+1}$ stehenden Faktoren. Die (globale) Recoverymenge $REC$ ist nun die Vereinigung aller lokalen Recoverymengen $REC_1, \ldots, REC_n$.

Nach einem Fehler werden in der Eingabe solange Token überlesen, bis das aktuelle Eingabetoken in $REC$ enthalten ist. Dessen Position ist der oben schon mehrfach erwähnte Wiederaufsetzpunkt. Das an dieser Stelle gefundene Token wird im weiteren Fortgang des Parsens im Reparaturmodus in einer der Prozeduren $P_1, \ldots, P_n$ erwartet und führt spätestens dort zum Verlassen dieses Modus. Bis dahin wird für jedes Terminal dieses selbst und für jedes Nichtterminal das gemäß der Fortsetzungsgrammatik erzeugte Wort als eingefügt gemeldet. Kommt das Eingabetoken dabei allerdings in einem der eingefügten Wörter vor, so wird in dieser Implementierung der Reparaturmodus verfrüht verlassen, was zu kleineren Einfügungen, dadurch aber manchmal auch zu weiteren Fehlermeldungen führt.

Die globalen Recoverymengen müssen zur Laufzeit berechnet werden, da sie, wie gezeigt, von der jeweiligen Aufrufhierarchie abhängen. Die lokalen Recoverymengen für alle Faktoren aller Regeln aber können bereits vom Generator berechnet werden. Zur Laufzeit des generierten Parsers werden diese dann als Indizes in einem Keller verwaltet, der die Aufrufhierarchie widerspiegelt; daraus kann bei Bedarf die globale Recoverymenge berechnet werden.

Für EBNF-Grammatiken ist die globale Recoverymenge über die Transformation aus Abschnitt 2.2 analog definiert. Zu beachten ist jedoch, daß anonyme Nichtterminale in eingebetteten Code umge-

setzt werden und daher zur Laufzeit keine Prozeduraufrufe für sie in der Aufrufhierarchie vorhanden sind; die Kellerung ihrer lokalen Recoverymengen könnte aber trotzdem durchgeführt werden. Diese explizite Kellerung zur Laufzeit kann jedoch dadurch eingespart werden, daß der Generator die Vereinigung der betreffenden lokalen Recoverymengen auf dem Laufzeitkeller vorwegnimmt, indem er die lokale Recoverymenge eines anonymen Faktors allen lokalen Recoverymengen dieses Nichtterminals hinzufügt. Die lokale Recoverymenge eines anonymen Faktors braucht dann zur Laufzeit nicht mehr auf den Keller gelegt zu werden. Die in den Codeschemata für Alternativen, Optionen und Wiederholungen benötigten lokalen Recoverymengen enthalten die Firstmenge des Nichtterminals, durch das sie realisiert werden; ist dieses ein anonymes Nichtterminal, so wird noch die lokale Recoverymenge des anonymen Faktors beigefügt, wie gerade beschrieben.

Die Prozedur `ComputeRec` führt alle nötigen Berechnungen für EBNF-Grammatiken durch. Die Alternativen werden iterativ von hinten nach vorne durchlaufen; dabei wird je Alternative eine aktuelle lokale Recoverymenge um entsprechende Token angereichert und zu den Faktoren gespeichert. Anonyme Nichtterminale für EBNF-Konstrukte werden durch eine rekursive Traversierung mit Übergabe der aktuellen lokalen Recoverymenge als Initialisierung behandelt.

In manchen Fällen wird nach einem Syntaxfehler der gesamte Rest der Eingabe überlesen; in der (globalen) Recoverymenge $REC$ muß daher stets auch das spezielle Token für das Eingabeende enthalten sein. In dieser Implementierung wird dies dadurch garantiert, daß es allen lokalen Recoverymengen des Startsymbols explizit hinzugefügt wird, indem es für alle Alternativen des Startsymbols in die initiale lokale Recoverymenge eingetragen wird. Um das Verhalten der Parser bei überflüssigen Token in der Eingabe zu verbessern, wird außerdem noch der Faktor $F$, bei dem der Fehler auftrat, in $REC$ aufgenommen, falls es sich um ein Terminal handelt.

Die Definition der Recoverymengen für EBNF-Grammatiken über die Transformation aus Abschnitt 2.2 führt bei den Wiederholungen zur Aufnahme der Firstmenge des zugehörigen Nichtterminals in die lokalen Recoverymengen der Faktoren des Konstrukts; in der rechtsrekursiven Formulierung der Wiederholung steht als letzter Faktor jeder Alternative nämlich der rekursive Wiederaufruf. Damit ergibt sich ein von der Verwendung dieses EBNF-Konstrukts unabhängiges konsistentes Wiederaufsetzverhalten. Die Aufnahme dieser Firstmenge scheint im Implementierungsvorschlag von Grosch vergessen worden zu sein.

#### 5.2.2.5 Speicherung der Mengen für den generierten Parser

Die im generierten Parser benötigten Mengen werden im Generator zu Bestandteilen der Grammatik berechnet und über eine externe Tabelle dem Parser verfügbar gemacht, in dem sie in globalen Feldern gespeichert werden. So können Mengen dort effizient durch ihren Index in einem der Felder bezeichnet werden. Um diese Indizes als feste Zahlen in das Parsermodul eintragen zu können, könnte die Parsertabelle vor dessen Generierung erstellt und dabei die Position jeder Menge als ihr Index mitprotokolliert werden. Eine Kompression der Tabelle sowie die in Abschnitt 5.2.1.4 vorgestellte Transposition der First- und Followmengen erfordern aber, die globalen Felder des generierten Parsers im Generator nachzubilden und erst später in die Parsertabelle zu übertragen.

In der Prozedur `ComputeSets` werden durch einmalige Traversierung der Grammatik alle benötigten First- und Followmengen im Feld `GenSetT` und alle Expected- und lokalen Recoverymengen im Feld `GenSet` abgelegt. Um in den generierten Parsern Speicherplatz einzusparen, werden gleiche Mengen nur einmal in jedes Feld eingetragen; dafür ist jeweils eine lineare Suche über die bisherigen Eintragungen nötig. Die Indizes der Mengen werden dabei in entsprechenden Komponenten zu Nichtterminalen und Faktoren gespeichert.

Für jede Option und jede Wiederholung wird eine First- und eine Followmenge benötigt. Da diese Mengen bereits alle zur Bestimmung der Direktormengen berechnet und bei den Nichtterminalen gespeichert wurden, muß die erforderliche Auswahl nur noch in das Feld `GenSetT` kopiert werden. Die Indizes der Mengen in diesem Feld werden in den `Nont`-Komponenten `FirstIndex` und

`FollowIndex` abgelegt.

Expectedmengen werden für die EBNF-Konstrukte Alternative (mit mindestens zwei Fällen), Option und Wiederholung benötigt. Nichtterminale können in dieser Implementierung gleichzeitig eine Alternative und eine Option bzw. Wiederholung realisieren. Die Indizes der Expectedmengen werden daher in den zwei `Nont`-Komponenten `AltExp` und `OptExp` gespeichert. Zu beachten ist, daß zur Generierung aber stets nur eine Expectedmenge benötigt wird, da in der Umsetzung einer Alternative der Fehlerbehandlungsteil wegoptimiert wird, wenn das entsprechende Nichtterminal auch eine Option oder eine Wiederholung realisiert.

Recoverymengen sind für jedes Terminal und jedes benannte Nichtterminal auf einer rechten Regelseite erforderlich. Die Indizes werden in der `Factor`-Komponente `Rec` abgelegt. Anonyme Nichtterminale auf rechten Regelseiten werden in eingebetteten Code umgesetzt; ihre lokale Recoverymenge wird daher schon vom Generator allen ihren lokalen Recoverymengen hinzugefügt und braucht selber nicht mehr gespeichert zu werden. Weitere Recoverymengen sind für obige EBNF-Konstrukte nötig. Die Indizes dieser Mengen werden bei den sie realisierenden Nichtterminalen in den `Nont`-Komponenten `AltRec` und `OptRec` gespeichert. Wie bei den Expectedmengen wird zur Generierung jedoch stets nur eine der beiden Komponenten benötigt. Berechnet werden die Expected- und lokalen Recoverymengen wie im vorigen Abschnitt beschrieben.

#### 5.2.2.6 Generierung des Parsermoduls

Die Erstellung eines Parsermoduls erfolgt durch die Prozedur `GenerateMod`. Das Modulgerüst und die festen Teile, wie beispielsweise die Prozeduren zur Fehlerreparatur, werden aus einer Textdatei in den zu generierenden Parser kopiert. Dabei werden an den Einfügemarken „`$`“ die spezifischen Teile eingefügt.

Die Evaluationsanweisungen für die Affixauswertung werden zur Verschränkung mit der Syntaxanalyse durch Benutzung von Generierungsprozeduren des Evaluatorgenerators in den Parsercode eingestreut; in ihrer Gesamtheit bilden sie zusammen mit der impliziten Traversierung durch den Parser den sogenannten Evaluator. Auf diese Weise wird der Parser zu einem Ein-Pass-Compiler erweitert.

Zuerst werden die festen Teile des Evaluators sowie die Prädikatprozeduren mittels solcher Generierungsprozeduren in den Parser eingefügt. Hinter die festen Teile des Parsers werden anschließend die Parserprozeduren für die Nichtterminale der Grundgrammatik geschrieben. Da sich diese Prozeduren im allgemeinen beliebig gegenseitig aufrufen können, werden einfach davor noch Vorwärtsdeklarationen für alle diese Prozeduren erzeugt. Zuletzt werden die Prozeduren für die Ausgabe der Übersetzungen des Compilers durch Verwendung des Moduls `EmitGen` in den Parser geschrieben und das exportierte Kommando `Compile` eingefügt.

Die Nichtterminale der Grundgrammatik werden in den Prozeduren `TraverseNont`, `TraverseAlts` und `TraverseFactors` gemäß den in Abschnitt 5.2.1.3 beschriebenen Codeschemata in je eine eigene Prozedur umgesetzt. Die dort angesprochenen Optimierungen werden durchgeführt, indem eine Menge von zur Laufzeit möglichen Token sowie ein Flag zur Erkennung des jeweils ersten Nichtterminalvorkommens einer Alternative mitgeführt werden. Die durch anonyme Nichtterminale repräsentierten EBNF-Konstrukte werden durch eine rekursive Traversierung in eingebetteten Code umgesetzt.

Wird, wie in Abschnitt 7 beschrieben, kein Ein-Pass-Compiler generiert, sondern ein Parser, der einen statischen Ableitungsbaum für einen nachfolgenden Evaluator aufbaut, so entfallen die Ausgabeprozeduren und in der Prozedur `Compile` wird zum Schluß der erstellte Baum diesem Evaluator übergeben.

#### 5.2.2.7 Erstellung der Parsertabelle

Die Parsertabelle wird in der Prozedur `WriteTab` als permanente Datei angelegt und enthält die vom generierten Parser benötigten First-, Follow-, Expected- und lokalen Recoverymengen. Diese Mengen könnten auch in den Quelltext des Parsermoduls eingetragen werden; da die Generatorzielsprache Oberon aber keine strukturierten Konstanten vorsieht, ist die Verwendung einer externen Tabelle die angemessenere Wahl.

Zuerst werden die im globalen Feld `GenSetT` abgelegten First- und Followmengen so als Folge von Werten des Basistyps SET in die Tabelle geschrieben, daß die in Abschnitt 5.2.1.4 beschriebene Transposition realisiert wird. Dahinter werden die Expected- und lokalen Recoverymengen aus dem Feld `GenSet` abgelegt, ebenfalls als Folge von Werten des Typs SET.

Einige zusätzliche Dateieinträge ermöglichen es dem generierten Parser, ein korrektes Einlesen zu überprüfen (insbesondere bei Portierungen auf andere Computer) bzw. festzustellen, ob auch die zu ihm gehörige Tabelle eingelesen wurde.

#### 5.2.2.8 Sonderbehandlung der Nichtterminale mit Tokenmarkierungen

Normalerweise beachtet ein generierter Parser Zwischenraum in der Eingabe nicht. Da dieser aber auch die Nichtterminale mit Tokenmarkierungen – im folgenden als Token-Nichtterminale bezeichnet – analysiert, wäre ohne Sonderbehandlung Zwischenraum innerhalb von Token zulässig und daher nicht zu ihrer Trennung verwendbar. Kommen Token-Nichtterminale in einer Regel direkt hintereinander vor, so kann sich dieser Umstand als nicht intendierte Mehrdeutigkeit äußern.

Als Abhilfe wurde der verwendete Scanner um eine zweite `Get`-Prozedur erweitert; diese verhält sich exakt so wie die ursprüngliche Prozedur, nur überliest sie Zwischenraum in der Eingabe nicht einfach, sondern liefert dafür das spezielle Token `sepTok`. Der Parsergenerator fügt ferner in Grundgrammatiken hinter jedem Vorkommen eines Token-Nichtterminals in einer Alternative eines gewöhnlichen Nichtterminals ein optionales Terminal für Zwischenraum ein. Diese Eintragungen finden nur virtuell statt, indem die Terminale einfach in den Prozeduren zur Berechnung von Follow-, Expected- und Recoverymengen sowie der Generierungsprozedur für Faktoren als eingefügt behandelt werden. Weiterhin verwenden die generierten Parser die neue `Get`-Prozedur für die Analysen der Token-Nichtterminale. Diese werden nun korrekt durch Zwischenraum beendet.

In der Prozedur `ComputeRegNonts` werden alle von Token-Nichtterminalen aus erreichbaren sogenannten Subtoken-Nichtterminale bestimmt und zusammen mit diesen selbst in der Menge `RegNonts` gespeichert; dies erleichtert die Überprüfung der folgenden Wohlgeformtheitsbedingungen in der Prozedur `GrammarOk`:

Um sich die Option eines echten Scannergenerators offen zu halten, dürfen die Token-Nichtterminale nicht leerableitbar sein. Für Subtoken-Nichtterminale wird Leerableitbarkeit erlaubt, dafür dürfen sie in den Alternativen gewöhnlicher Nichtterminale nicht vorkommen. Dies entspricht der üblichen Trennung der kontextfreien Grammatik von den Tokengrammatiken und stellt praktisch keine Einschränkung dar. Ferner darf das Startsymbol der Grundgrammatik nicht in `RegNonts` enthalten sein.

Ein letztes zu beseitigendes Ärgernis sind die LL(1)-Konfliktmeldungen aufgrund der oben erwähnten Mehrdeutigkeit. Kommen in einer Spezifikation beispielsweise zwei Token-Nichtterminale zum Erkennen von Namen direkt hintereinander vor, so überschneidet sich beim ersten die Direktormenge der Alternative zum Anhängen von Buchstabentoken mit der Direktormenge der Abbruchalternative. In beiden Mengen sind alle Buchstaben enthalten, in der zweiten zusätzlich das Token `sepTok`. Da zwei in der Eingabe nur durch Zwischenraum getrennte Namen jetzt aber korrekt eingelesen werden, ist eine Konfliktmeldung überflüssig; dies ist vom Benutzer aber oft nicht leicht zu erkennen, weil sich die Meldung meist nicht auf ein Token-Nichtterminal bezieht,

sondern auf darin vorkommende Subtoken-Nichtterminale.

Zur teilweisen Behebung des Problems werden für gängige mit EBNF-Konstrukten gebildete Token überflüssige Meldungen auf folgende Weise unterdrückt: Verläßt die Wahl des Abbruchs ein Token sicher, so wird bei Optionen und Wiederholungen eine Überschneidung der Direktormenge für diesen Abbruch (also der Followmenge) mit der Direktormenge für die Wahl der Option bzw. Wiederholung (also der Vereinigung der Direktormengen aller Alternativen) ignoriert, es sei denn, eine der Alternativen ist leerableitbar. Die Wahl des Abbruchs verläßt ein Token sicher, wenn das Nichtterminal, das die Option oder Wiederholung realisiert, in beliebigen Satzformen zu allen Token-Nichtterminalen nur ganz hinten vorkommt. Die Nichtterminale mit dieser Eigenschaft werden in der Prozedur `ComputeRegNonts` durch Traversieren der Grammatik bestimmt und in der Menge `ConflictNonts` gespeichert.

## 5.3 Implementierungen

### 5.3.1 eScanGen.Mod

```
MODULE eScanGen;    (* SteWe 10/96  -  Ver 2.0 *)

IMPORT IO := eIO, Scanner := eScanner, EAG := eEAG;

CONST firstUserTok = 3;  (* R *)
      lenOfPredefinedToken = 8;

VAR IsIdent, IsSymbol : ARRAY 256 OF BOOLEAN;
    i : INTEGER;

PROCEDURE Generate*;
  VAR
    Fix : IO.TextIn; Mod : IO.TextOut;
    Term, MaxTokLen, Len : INTEGER;
    Str : ARRAY 400 OF CHAR;
    Name : ARRAY EAG.BaseNameLen + 10 OF CHAR;
    Error, OpenError, CompileError, ShowMod : BOOLEAN;

  PROCEDURE TestToken(VAR s : ARRAY OF CHAR; VAR Len : INTEGER);
    VAR i : INTEGER;

    PROCEDURE Err(Msg : ARRAY OF CHAR);
      VAR i : INTEGER;
    BEGIN Error := TRUE;
      IO.WriteText(IO.Msg, "\n  error in token: ");
      i := 0; WHILE s[i] # 0X DO IO.Write(IO.Msg, s[i]); INC(i) END;
      IO.WriteString(IO.Msg, " - "); IO.WriteText(IO.Msg, Msg); IO.Update(IO.Msg)
    END Err;

  BEGIN
    Len := 0;
    IF ((s[0] # "'") & (s[0] # '"')) OR (s[1] = 0X) OR (s[1] = s[0]) THEN
      Err("must be non empty string"); RETURN
    END;
    IF (s[1] = "'") OR (s[1] = '"') OR (s[1] = " ") THEN i := 2
    ELSIF IsIdent[ORD(s[1])] THEN i := 2; WHILE IsIdent[ORD(s[i])] DO INC(i) END
    ELSIF IsSymbol[ORD(s[1])] THEN i := 2; WHILE IsSymbol[ORD(s[i])] DO INC(i) END
    ELSE Err("contains illegal char"); RETURN
    END;
    IF (s[i] # s[0]) OR (s[i + 1] # 0X) THEN Err("contains illegal char"); RETURN END;
    Len := i - 1
  END TestToken;

  PROCEDURE InclFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = 0X THEN
        IO.WriteText(IO.Msg, "\n  error: unexpected end of eScanGen.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Write(Mod, c); IO.Read(Fix, c)
    END
  END InclFix;

  PROCEDURE Append(VAR Dest : ARRAY OF CHAR; Src, Suf : ARRAY OF CHAR);
    VAR i, j : INTEGER;
  BEGIN
    i := 0; j := 0;
    WHILE (Src[i] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Src[i]; INC(i) END;
    WHILE (Suf[j] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Suf[j]; INC(i); INC(j) END;
    Dest[i] := 0X
  END Append;

BEGIN  (* Generate *)
  ShowMod := IO.IsOption("m");
  IO.WriteString(IO.Msg, "ScanGen writing "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteString(IO.Msg, "   "); IO.Update(IO.Msg);
  IF EAG.Performed({EAG.analysed}) THEN
    Error := FALSE; MaxTokLen := lenOfPredefinedToken;
    FOR Term := EAG.firstHTerm TO EAG.NextHTerm - 1 DO
      Scanner.GetRepr(EAG.HTerm[Term].Id, Str); TestToken(Str, Len);
```

```
      IF Len > MaxTokLen THEN MaxTokLen := Len END
    END;
    IF ~ Error THEN
      IO.OpenIn(Fix, "eScanGen.Fix", OpenError);
      IF OpenError THEN
        IO.WriteText(IO.Msg, "\n  error: cannot open eScanGen.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      Append(Name, EAG.BaseName, "Scan"); IO.CreateModOut(Mod, Name);
      InclFix("$"); IO.WriteString(Mod, Name);
      InclFix("$"); IO.WriteInt(Mod, MaxTokLen + 1);
      InclFix("$"); IO.WriteInt(Mod, EAG.NextHTerm - EAG.firstHTerm + firstUserTok);
      InclFix("$");
      FOR Term := EAG.firstHTerm TO EAG.NextHTerm - 1 DO
        IO.WriteText(Mod, "\t\tEnter(");
        IO.WriteInt(Mod, Term - EAG.firstHTerm + firstUserTok);
        IO.WriteText(Mod, ", ");
        Scanner.WriteRepr(Mod, EAG.HTerm[Term].Id);
        IO.WriteText(Mod, ");\n")
      END;
      InclFix("$"); IO.WriteString(Mod, Name);
      InclFix("$");
      IO.CloseIn(Fix); IO.Update(Mod);
      IF ShowMod THEN IO.Show(Mod); IO.WriteLn(IO.Msg)
      ELSE IO.Compile(Mod, CompileError); IF CompileError THEN IO.Show(Mod) END
      END;
      IO.CloseOut(Mod)
    ELSE IO.WriteLn(IO.Msg)
    END
  ELSE IO.WriteLn(IO.Msg)
  END;
  IO.Update(IO.Msg)
END Generate;

BEGIN
  FOR i := 0 TO LEN(IsIdent) - 1 DO IsIdent[i] := FALSE END;
  FOR i := ORD("A") TO ORD("Z") DO IsIdent[i] := TRUE END;
  FOR i := ORD("a") TO ORD("z") DO IsIdent[i] := TRUE END;
  FOR i := ORD("0") TO ORD("9") DO IsIdent[i] := TRUE END;
  FOR i := 0 TO ORD(" ") DO IsSymbol[i] := FALSE END;
  FOR i := ORD(" ") + 1 TO LEN(IsSymbol) - 1 DO IsSymbol[i] := ~ IsIdent[i] END;
  IsSymbol[ORD("'")] := FALSE; IsSymbol[ORD('"')] := FALSE
END eScanGen.
```

### 5.3.2 eScanGen.Fix

```
MODULE $; (* General purpose Scanner by JoDe Version 1.02 -- 22.11.96 *)

  IMPORT IO := eIO;

  CONST
    EOT = 0X; STR = 22X;
    firstChBuf = 0; chBufLen = 512;
  VAR
    ChBuf: ARRAY chBufLen OF CHAR; PosBuf: ARRAY chBufLen OF IO.Position; CurCh, NextCh: INTEGER;
    In: IO.TextIn;

  CONST
    nil = 0; firstNode = 1; maxTokLen = $; maxTok = $;
    eot = 0; undef = 1; whitespace = 2; comment = MIN(INTEGER);
    string = 0; ident = 1;  none = 2;
  TYPE
    OpenNode = POINTER TO ARRAY OF RECORD Ch: CHAR; Tok, Next, Sub: INTEGER END;
  VAR
    Node: OpenNode; NextNode: INTEGER;
    NameTab: ARRAY maxTok, maxTokLen OF CHAR;
    IsWhitespace, IsIdent: ARRAY 256 OF BOOLEAN;
    Ch: CHAR;
    Mode: INTEGER; StringCh: CHAR;
    Pos*: IO.Position; Get*: PROCEDURE (VAR Tok: INTEGER);

  PROCEDURE CopyBuf;
    VAR i, j: INTEGER;
  BEGIN
    i := CurCh; j := firstChBuf;
    WHILE i < NextCh DO ChBuf[j] := ChBuf[i]; PosBuf[j] := PosBuf[i]; INC(i); INC(j) END;
    CurCh := firstChBuf; NextCh := j
  END CopyBuf;

  PROCEDURE GetCh;
  BEGIN
    IF CurCh = NextCh THEN IO.Read(In, Ch)
    ELSE Ch := ChBuf[CurCh]; INC(CurCh)
    END
  END GetCh;

  PROCEDURE GetBufCh;
  BEGIN
    IF CurCh = NextCh THEN
      IO.Pos(In, PosBuf[NextCh]); IO.Read(In, Ch); ChBuf[NextCh] := Ch; INC(NextCh); INC(CurCh)
    ELSE
      Ch := ChBuf[CurCh]; INC(CurCh)
    END
  END GetBufCh;

  PROCEDURE GetPos;
  BEGIN
    IF CurCh = NextCh THEN IO.PrevPos(In, Pos)
    ELSE Pos := PosBuf[CurCh-1]
    END
  END GetPos;

  PROCEDURE Error(Txt: ARRAY OF CHAR);
  BEGIN
    IO.WritePos(IO.Msg, Pos); IO.WriteString(IO.Msg, "  ");
    IO.WriteString(IO.Msg, Txt); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END Error;

  PROCEDURE Enter(Tok: INTEGER; Name: ARRAY OF CHAR);
    VAR Ptr, i: INTEGER;

    PROCEDURE Insert(VAR Ptr: INTEGER; Ch: CHAR);

      PROCEDURE Expand;
        VAR i: LONGINT; Node1: OpenNode;
      BEGIN
        IF NextNode >= LEN(Node^) THEN
          IF LEN(Node^) < MAX(INTEGER) DIV 2 THEN NEW(Node1, 2 * LEN(Node^) + 1);
            FOR i := firstNode TO LEN(Node^) - 1 DO Node1[i] := Node[i] END; Node := Node1
          ELSE HALT(99)
          END
        END
      END Expand;
```

```
  BEGIN (* Insert(VAR Ptr: INTEGER; Ch: CHAR); *)
    Ptr := NextNode;
    Node[NextNode].Ch := Ch;
    Node[NextNode].Tok := undef;
    Node[NextNode].Next := nil;
    Node[NextNode].Sub := nil;
    INC(NextNode); IF NextNode >= LEN(Node^) THEN Expand END
  END Insert;

BEGIN (* Enter(Tok: INTEGER; Name: ARRAY OF CHAR) *)
  IF Tok >= 0 THEN COPY(Name, NameTab[Tok]) END;
  Ptr := ORD(Name[0]);
  i := 0;
  WHILE Name[i] # 0X DO
    WHILE (Node[Ptr].Ch # Name[i]) & (Node[Ptr].Next # nil) DO Ptr := Node[Ptr].Next END;
    IF Node[Ptr].Ch # Name[i] THEN Insert(Node[Ptr].Next, Name[i]); Ptr := Node[Ptr].Next END;
    INC(i);
    IF (Node[Ptr].Sub # nil) & (Name[i] # 0X) THEN
      Ptr := Node[Ptr].Sub
    ELSE
      WHILE i < LEN(Name) - 1 DO Insert(Node[Ptr].Sub, Name[i]); Ptr := Node[Ptr].Sub; INC(i) END
    END
  END;
  Node[Ptr].Tok := Tok
END Enter;

PROCEDURE WriteRepr*(VAR Out : IO.TextOut; Tok : INTEGER);
BEGIN
  IO.WriteString(Out, NameTab[Tok])
END WriteRepr;

PROCEDURE Symbol(VAR Tok: INTEGER);
  VAR Ptr, Mark: INTEGER;
BEGIN
  Ptr := ORD(Ch);
  Tok := Node[Ptr].Tok;
  IF Node[Ptr].Sub # nil THEN
    IF NextCh >= LEN(ChBuf) - maxTokLen THEN CopyBuf END;
    Mark := CurCh;
    REPEAT
      IF Node[Ptr].Tok # undef THEN Tok := Node[Ptr].Tok; Mark := CurCh END;
      Ptr := Node[Ptr].Sub;
      GetBufCh;
      WHILE (Ptr # nil) & (Node[Ptr].Ch # Ch) DO Ptr := Node[Ptr].Next END
    UNTIL Ptr = nil;
    CurCh := Mark
  END;
  GetCh
END Symbol;

PROCEDURE Keyword(VAR Tok: INTEGER);
  VAR Ptr, LastPtr, Mark: INTEGER;
BEGIN
  Ptr := ORD(Ch);
  Tok := Node[Ptr].Tok;
  IF NextCh >= LEN(ChBuf) - maxTokLen THEN CopyBuf END;
  Mark := CurCh;
  REPEAT
    LastPtr := Ptr; Ptr := Node[Ptr].Sub;
    GetBufCh;
    WHILE (Ptr # nil) & (Node[Ptr].Ch # Ch) DO Ptr := Node[Ptr].Next END
  UNTIL Ptr = nil;
  IF (Node[LastPtr].Tok # undef) & ~ IsIdent[ORD(Ch)] THEN
    Tok := Node[LastPtr].Tok; CurCh := NextCh - 1
  ELSE CurCh := Mark; Mode := ident
  END;
  GetCh
END Keyword;

PROCEDURE Comment;
  VAR Level: INTEGER;
BEGIN
  Level := 1;
  LOOP
    IF Ch = EOT THEN Error("Comment not closed"); EXIT
    ELSIF Ch = "(" THEN GetCh;
      IF Ch = "*" THEN GetCh; INC(Level) END
    ELSIF Ch = "*" THEN GetCh;
      IF Ch = ")" THEN GetCh; DEC(Level); IF Level = 0 THEN EXIT END END
```

```
    ELSE GetCh
    END
  END
END Comment;

PROCEDURE Get2*(VAR Tok: INTEGER);
BEGIN
  CASE Mode OF
    string:
      IF Ch = EOT THEN Tok := eot
      ELSE
        IF (Ch = StringCh) OR (Ch = IO.eol) THEN Mode := none END;
        GetPos; Tok := Node[ORD(Ch)].Tok; GetCh
      END
    | ident:
      IF IsIdent[ORD(Ch)] THEN
        GetPos; Tok := Node[ORD(Ch)].Tok; GetCh
      ELSE
        Mode := none; Get(Tok)
      END
  ELSE (* Mode = none *)
    WHILE IsWhitespace[ORD(Ch)] DO GetCh END;
    GetPos;
    IF Ch = EOT THEN Tok := eot
    ELSIF IsIdent[ORD(Ch)] THEN Keyword(Tok)
    ELSIF (Ch = '"') OR (Ch = "'") THEN
      StringCh := Ch; Mode := string;
      Tok := Node[ORD(Ch)].Tok; GetCh
    ELSE
      Symbol(Tok); IF Tok = comment THEN Comment; Get(Tok) END
    END
  END
END Get2;

PROCEDURE Get3*(VAR Tok: INTEGER);
BEGIN
  CASE Mode OF
    string:
      IF Ch = EOT THEN Tok := eot
      ELSE
        IF (Ch = StringCh) OR (Ch = IO.eol) THEN Mode := none END;
        GetPos; Tok := Node[ORD(Ch)].Tok; GetCh
      END
    | ident:
      IF IsIdent[ORD(Ch)] THEN
        GetPos; Tok := Node[ORD(Ch)].Tok; GetCh
      ELSE
        Mode := none; Get(Tok)
      END
  ELSE (* Mode = none *)
    GetPos;
    IF Ch = EOT THEN Tok := eot
    ELSIF IsWhitespace[ORD(Ch)] THEN
      REPEAT GetCh UNTIL ~ IsWhitespace[ORD(Ch)]; Tok := whitespace
    ELSIF IsIdent[ORD(Ch)] THEN Keyword(Tok)
    ELSIF (Ch = '"') OR (Ch = "'") THEN
      StringCh := Ch; Mode := string;
      Tok := Node[ORD(Ch)].Tok; GetCh
    ELSE
      Symbol(Tok); IF Tok = comment THEN Comment; Tok := whitespace END
    END
  END
END Get3;

PROCEDURE Init*;
  VAR Name: ARRAY 512 OF CHAR; Error: BOOLEAN;
BEGIN
  IO.InputName(Name);
  IO.OpenIn(In, Name, Error);
  IF Error THEN
    IO.WriteText(IO.Msg, "Fatal error: cannot open input\n"); IO.Update(IO.Msg); HALT(99)
  END;
  CurCh := firstChBuf; NextCh := firstChBuf; Ch := " ";
  Mode := none; Get := Get2
END Init;

PROCEDURE BuildTree;
  VAR i: INTEGER;
BEGIN
  FOR i := 0 TO LEN(IsIdent) - 1 DO
```

```
      IF (CHR(i) >= "A") & (CHR(i) <= "Z") THEN IsIdent[i] := TRUE
      ELSIF (CHR(i) >= "a") & (CHR(i) <= "z")THEN IsIdent[i] := TRUE
      ELSIF (CHR(i) >= "0") & (CHR(i) <= "9")THEN IsIdent[i] := TRUE
      ELSE IsIdent[i] := FALSE
      END
    END;
    FOR i := 0 TO LEN(IsWhitespace) - 1 DO
      IF (CHR(i) <= " ") OR (CHR(i) > "~") THEN IsWhitespace[i] := TRUE
      ELSE IsWhitespace[i] := FALSE
      END
    END;
    IsWhitespace[ORD(EOT)] := FALSE;
    NEW(Node, 255); NextNode := ORD("~") + 1;
    FOR i := firstNode TO NextNode DO
      Node[i].Ch := CHR(i); Node[i].Tok := undef;
      Node[i].Next := nil; Node[i].Sub := nil
    END;
    INC(NextNode);
    COPY("endTok", NameTab[0]);
    COPY("undefTok", NameTab[1]);
    COPY("sepTok", NameTab[2]);
    Enter(whitespace, IO.eol);
    Enter(comment, "(*");
$  END BuildTree;

BEGIN
  BuildTree
END $.
$
```

### 5.3.3 eELL1Gen.Mod

```
MODULE eELL1Gen;    (* SteWe 11/96  -  Ver 1.5 *)
(*  every line concerning RegNonts is marked with "(* R *)"  *)

IMPORT Sets := eSets, IO := eIO, EAG := eEAG, EvalGen := eSLEAGGen,
       EmitGen := eEmitGen, Shift := eShift;

CONST
  nil = 0;
  endTok = 0; undefTok = 1; sepTok = 2; firstUserTok = 3;  (* R *)
  nElemsPerSET = MAX(SET) + 1;
  firstEdge = 1;
  firstGenSet = 1;
  firstGenSetT = 1;

TYPE
  OpenNont = POINTER TO ARRAY OF RECORD
               First, Follow, IniFollow : Sets.OpenSet;
               DefaultAlt : EAG.Alt;
               Edge : INTEGER;
               AltRec, OptRec, AltExp, OptExp : INTEGER;
               FirstIndex, FollowIndex : INTEGER;
               Anonym : BOOLEAN;
             END;
  OpenAlt = POINTER TO ARRAY OF RECORD
              Dir : Sets.OpenSet
            END;
  OpenFactor = POINTER TO ARRAY OF RECORD
                 Rec : INTEGER;
               END;
  OpenEdge = POINTER TO ARRAY OF RECORD Dest, Next : INTEGER END;
  OpenGenSet = POINTER TO ARRAY OF Sets.OpenSet;
  OpenGenSetT = POINTER TO ARRAY OF Sets.OpenSet;

VAR
  Nont : OpenNont;
  Alt : OpenAlt;
  Factor : OpenFactor;
  Edge : OpenEdge; NextEdge : INTEGER;
  GenSet : OpenGenSet; NextGenSet : INTEGER;
  GenSetT : OpenGenSetT; NextGenSetT : INTEGER;
  TestNonts, GenNonts, RegNonts, ConflictNonts : Sets.OpenSet;  (* R *)
  nToks : INTEGER;
  Error, Warning, ShowMod, Compiled, UseReg : BOOLEAN;  (* R *)

PROCEDURE Expand;
  VAR
    Edge1 : OpenEdge;
    GenSet1 : OpenGenSet;
    GenSetT1 : OpenGenSetT;
    i : LONGINT;

  PROCEDURE ExpLen(ArrayLen : LONGINT) : LONGINT;
  BEGIN
    IF ArrayLen <= MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen ELSE HALT(99) END
  END ExpLen;

BEGIN
  IF NextEdge >= LEN(Edge^) THEN NEW(Edge1, ExpLen(LEN(Edge^)));
    FOR i := firstEdge TO LEN(Edge^) - 1 DO Edge1[i] := Edge[i] END;
    Edge := Edge1
  END;
  IF NextGenSet >= LEN(GenSet^) THEN NEW(GenSet1, ExpLen(LEN(GenSet^)));
    FOR i := firstGenSet TO LEN(GenSet^) - 1 DO GenSet1[i] := GenSet[i] END;
    GenSet := GenSet1
  END;
  IF NextGenSetT >= LEN(GenSetT^) THEN NEW(GenSetT1, ExpLen(LEN(GenSetT^)));
    FOR i := firstGenSetT TO LEN(GenSetT^) - 1 DO GenSetT1[i] := GenSetT[i] END;
    GenSetT := GenSetT1
  END;
END Expand;

PROCEDURE ComputeRegNonts;  (* R *)  (* whole procedure *)
  VAR N : INTEGER; A : EAG.Alt; F : EAG.Factor;

  PROCEDURE TraverseRegNonts(N : INTEGER);
    VAR A : EAG.Alt; F : EAG.Factor;
  BEGIN
```

```
    A := EAG.HNont[N].Def.Sub;
    REPEAT
      F := A.Sub;
      WHILE F # NIL DO
        IF (F IS EAG.Nont) & Sets.In(TestNonts, F(EAG.Nont).Sym) &
            ~ Sets.In(RegNonts, F(EAG.Nont).Sym) THEN
          Sets.Incl(RegNonts, F(EAG.Nont).Sym); TraverseRegNonts(F(EAG.Nont).Sym)
        END;
        F := F.Next
      END;
      A := A.Next
    UNTIL A = NIL
  END TraverseRegNonts;

  PROCEDURE DeleteConflictNont(N : INTEGER);
    VAR A : EAG.Alt; F : EAG.Factor;
  BEGIN
    Sets.Excl(ConflictNonts, N);
    A := EAG.HNont[N].Def.Sub;
    REPEAT
      F := A.Sub;
      WHILE F # NIL DO
        IF (F IS EAG.Nont) & Sets.In(ConflictNonts, F(EAG.Nont).Sym) THEN
          DeleteConflictNont(F(EAG.Nont).Sym)
        END;
        F := F.Next
      END;
      A := A.Next
    UNTIL A = NIL
  END DeleteConflictNont;

BEGIN  (* ComputeRegNonts *)
  Sets.Empty(RegNonts);
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) & EAG.HNont[N].IsToken & ~ Sets.In(RegNonts, N) THEN
      Sets.Incl(RegNonts, N); TraverseRegNonts(N)
    END
  END;
  Sets.Assign(ConflictNonts, RegNonts);
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(ConflictNonts, N) THEN
      A := EAG.HNont[N].Def.Sub;
      REPEAT
        F := A.Last;
        WHILE (F # NIL) & (F IS EAG.Nont) & ~ Sets.In(TestNonts, F(EAG.Nont).Sym) DO
          F := F.Prev
        END;
        IF F # NIL THEN F := F.Prev END;
        WHILE F # NIL DO
          IF (F IS EAG.Nont) & Sets.In(ConflictNonts, F(EAG.Nont).Sym) THEN
            DeleteConflictNont(F(EAG.Nont).Sym)
          END;
          F := F.Prev
        END;
        A := A.Next
      UNTIL A = NIL
    END
  END
END ComputeRegNonts;

PROCEDURE Init;
  VAR i : INTEGER;
BEGIN
  nToks := EAG.NextHTerm - EAG.firstHTerm + firstUserTok;
  IF EAG.NextHNont >= 1 THEN NEW(Nont, EAG.NextHNont) ELSE NEW(Nont, 1) END;
  FOR i := EAG.firstHNont TO EAG.NextHNont - 1 DO
    Sets.New(Nont[i].First, nToks );
    Sets.New(Nont[i].Follow, nToks);
    Sets.New(Nont[i].IniFollow, nToks);
    Nont[i].DefaultAlt := NIL;
    Nont[i].AltRec := nil; Nont[i].OptRec := nil; Nont[i].AltExp := nil; Nont[i].OptExp := nil;
    Nont[i].FirstIndex := nil; Nont[i].FollowIndex := nil;
    Nont[i].Anonym := Sets.In(EAG.All, i) & (EAG.HNont[i].Id < 0);
  END;
  IF EAG.NextHAlt >= 1 THEN NEW(Alt, EAG.NextHAlt) ELSE NEW(Alt, 1) END;
  FOR i := EAG.firstHAlt TO EAG.NextHAlt - 1 DO
    Sets.New(Alt[i].Dir, nToks)
  END;
  IF EAG.NextHFactor >= 1 THEN NEW(Factor, EAG.NextHFactor) ELSE NEW(Factor, 1) END;
  FOR i := EAG.firstHFactor TO EAG.NextHFactor - 1 DO Factor[i].Rec := nil END;
```

```
  NEW(Edge, 255); NextEdge := firstEdge;
  NEW(GenSet, 511); NextGenSet := firstGenSet;
  NEW(GenSetT, 255); NextGenSetT := firstGenSetT;
  Sets.New(TestNonts, EAG.NextHNont);
  Sets.Difference(TestNonts, EAG.All, EAG.Pred);
  Sets.New(GenNonts, EAG.NextHNont);
  Sets.Intersection(GenNonts, EAG.Prod, EAG.Reach);
  Sets.Difference(GenNonts, GenNonts, EAG.Pred);
  Error := FALSE; Warning := FALSE;
  ShowMod := IO.IsOption("m");
  UseReg := ~ IO.IsOption("p");  (* R *)
  Sets.New(RegNonts, EAG.NextHNont);  (* R *)
  Sets.New(ConflictNonts, EAG.NextHNont);  (* R *)
  IF UseReg THEN ComputeRegNonts END  (* R *)
END Init;

PROCEDURE Finit;
BEGIN
  Nont := NIL; Alt := NIL; Factor := NIL;
  Edge := NIL; GenSet := NIL; GenSetT := NIL
END Finit;

PROCEDURE WriteTok(Out : IO.TextOut; Tok : INTEGER);
BEGIN
  IF Tok = endTok THEN IO.WriteText(Out, "!end!")
  ELSIF Tok = undefTok THEN IO.WriteText(Out, "!undef!")
  ELSIF Tok = sepTok THEN IO.WriteText(Out, "!sep!")  (* R *)
  ELSE EAG.WriteHTerm(Out, Tok + EAG.firstHTerm - firstUserTok)
  END
END WriteTok;

PROCEDURE WriteTokSet(Out : IO.TextOut; Toks : Sets.OpenSet);
  VAR Tok : INTEGER;
BEGIN
  FOR Tok := 0 TO nToks - 1 DO
    IF Sets.In(Toks, Tok) THEN WriteTok(Out, Tok); IO.WriteText(Out, " ") END
  END
END WriteTokSet;

PROCEDURE NewEdge(From, To : INTEGER);
BEGIN
  IF NextEdge = LEN(Edge^) THEN Expand END;
  Edge[NextEdge].Dest := To;
  Edge[NextEdge].Next := Nont[From].Edge; Nont[From].Edge := NextEdge;
  INC(NextEdge)
END NewEdge;

PROCEDURE GrammarOk() : BOOLEAN;  (* R *)  (* whole procedure *)
  VAR N : INTEGER; A : EAG.Alt; F : EAG.Factor; Ok : BOOLEAN;
BEGIN
  Ok := TRUE;
  IF UseReg THEN
    IF Sets.In(RegNonts, EAG.StartSym) THEN
      IF EAG.HNont[EAG.StartSym].IsToken THEN
        IO.WriteText(IO.Msg, "\n  start symbol must not be a token")
      ELSE IO.WriteText(IO.Msg, "\n  start symbol must not be a subtoken")
      END;
      IO.Update(IO.Msg); Ok := FALSE
    END;
    FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
      IF Sets.In(TestNonts, N) & EAG.HNont[N].IsToken THEN
        IF Sets.In(EAG.Null, N) THEN
          IO.WriteText(IO.Msg, "\n  marked token "); EAG.WriteHNont(IO.Msg, N);
          IO.WriteText(IO.Msg, " is nullable"); IO.Update(IO.Msg); Ok := FALSE
        END;
        IF Nont[N].Anonym THEN
          IO.WriteText(IO.Msg, "\n  internal error: token in "); EAG.WriteNamedHNont(IO.Msg, N);
          IO.WriteText(IO.Msg, " is anonym"); IO.Update(IO.Msg); Ok := FALSE
        END
      END
    END;
    FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
      IF Sets.In(TestNonts, N) & ~ Sets.In(RegNonts, N) THEN
        A := EAG.HNont[N].Def.Sub;
        REPEAT
          F := A.Sub;
          WHILE F # NIL DO
            IF (F IS EAG.Nont) & Sets.In(TestNonts, F(EAG.Nont).Sym) &
                Sets.In(RegNonts, F(EAG.Nont).Sym) &
                ~ EAG.HNont[F(EAG.Nont).Sym].IsToken THEN
```

```
            IO.WriteText(IO.Msg, "\n  nonterminal ");
            IF Nont[N].Anonym THEN IO.WriteText(IO.Msg, "in ") END;
            EAG.WriteNamedHNont(IO.Msg, N);
            IO.WriteText(IO.Msg, " calls subtoken ");
            EAG.WriteHNont(IO.Msg, F(EAG.Nont).Sym); IO.Update(IO.Msg); Ok := FALSE
          END;
          F := F.Next
        END;
        A := A.Next
      UNTIL A = NIL
    END
  END
END;
RETURN Ok
END GrammarOk;

PROCEDURE ComputeDir;
  VAR
    N : INTEGER; A : EAG.Alt; F : EAG.Factor;
    State : POINTER TO ARRAY OF INTEGER;
    Stack : POINTER TO ARRAY OF INTEGER; Top : INTEGER;
    NullAlts, Toks : Sets.OpenSet;
    IsLast : BOOLEAN;

  PROCEDURE ComputeFirst(N : INTEGER);
    VAR n, E, N1 : INTEGER; Leftrecursion : BOOLEAN;
  BEGIN
    Stack[Top] := N; INC(Top); n := Top; State[N] := n;
    E := Nont[N].Edge; Leftrecursion := FALSE;
    WHILE E # nil DO
      N1 := Edge[E].Dest;
      IF N1 = N THEN Leftrecursion := TRUE END;
      IF State[N1] = 0 THEN ComputeFirst(N1) END;
      IF State[N1] < State[N] THEN State[N] := State[N1] END;
      Sets.Union(Nont[N].First, Nont[N].First, Nont[N1].First);
      E := Edge[E].Next
    END;
    IF State[N] = n THEN
      Leftrecursion := Leftrecursion OR (Top > n);
      IF Leftrecursion THEN
        Error := TRUE; IO.WriteText(IO.Msg, "\n  left recursion over nonterminals:")
      END;
      REPEAT
        DEC(Top); N1 := Stack[Top]; State[N1] := MAX(INTEGER);
        IF Leftrecursion THEN
          IO.WriteText(IO.Msg, "\n    "); EAG.WriteNamedHNont(IO.Msg, N1);
          IF Nont[N1].Anonym THEN
            IO.WriteText(IO.Msg, " (EBNF at "); IO.WritePos(IO.Msg, EAG.HNont[N1].Def.Sub.Pos);
            IO.WriteText(IO.Msg, ")")
          END
        END;
        Nont[N1].First := Nont[N].First
      UNTIL Top < n;
      IF Leftrecursion THEN IO.Update(IO.Msg) END
    END
  END ComputeFirst;

  PROCEDURE ComputeFollow(N : INTEGER);
    VAR n, E, N1 : INTEGER;
  BEGIN
    Stack[Top] := N; INC(Top); n := Top; State[N] := n;
    E := Nont[N].Edge;
    WHILE E # nil DO
      N1 := Edge[E].Dest;
      IF State[N1] = 0 THEN ComputeFollow(N1) END;
      IF State[N1] < State[N] THEN State[N] := State[N1] END;
      Sets.Union(Nont[N].Follow, Nont[N].Follow, Nont[N1].Follow);
      E := Edge[E].Next
    END;
    IF State[N] = n THEN
      REPEAT
        DEC(Top); N1 := Stack[Top]; State[N1] := MAX(INTEGER);
        Nont[N1].Follow := Nont[N].Follow
      UNTIL Top < n
    END
  END ComputeFollow;

  PROCEDURE Conflict(N : INTEGER; Pos : IO.Position; Dir, PrevDirs : Sets.OpenSet);
    VAR Toks : Sets.OpenSet;
  BEGIN
```

```
    Warning := TRUE; Sets.New(Toks, nToks);
    IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, Pos);
    IO.WriteText(IO.Msg, "  director set conflict in ");
    EAG.WriteNamedHNont(IO.Msg, N); IO.WriteText(IO.Msg, ": ");
    Sets.Intersection(Toks, Dir, PrevDirs); WriteTokSet(IO.Msg, Toks);
    Sets.Difference(Toks, Dir, PrevDirs);
    IF Sets.IsEmpty(Toks) THEN
      Error := TRUE; IO.WriteText(IO.Msg, "\n    alternative will never be chosen");
    END;
    IO.Update(IO.Msg)
  END Conflict;

BEGIN (* ComputeDir *)
  NEW(State, EAG.NextHNont);
  NEW(Stack, EAG.NextHNont); Top := 0;
  Sets.New(NullAlts, EAG.NextHAlt); Sets.New(Toks, nToks);
  NextEdge := firstEdge;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    Nont[N].Edge := nil; State[N] := 0
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) THEN
      Sets.Empty(Nont[N].First);
      A := EAG.HNont[N].Def.Sub;
      REPEAT
        F := A.Sub;
        LOOP
          IF F = NIL THEN EXIT END;
          IF F IS EAG.Term THEN
            Sets.Incl(Nont[N].First, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok); EXIT
          ELSE
            IF Sets.In(TestNonts, F(EAG.Nont).Sym) THEN
              NewEdge(N, F(EAG.Nont).Sym);
              IF ~ Sets.In(EAG.Null, F(EAG.Nont).Sym) THEN EXIT END
            END
          END;
          F := F.Next
        END;
        A := A.Next
      UNTIL A = NIL
    END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) & (State[N] = 0) THEN ComputeFirst(N) END
  END;
  NextEdge := firstEdge;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    Nont[N].Edge := nil; Sets.Empty(Nont[N].Follow)
  END;
  Sets.Incl(Nont[EAG.StartSym].Follow, endTok); Sets.Empty(NullAlts);
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) THEN
      A := EAG.HNont[N].Def.Sub;
      REPEAT
        IF EAG.HNont[N].Def IS EAG.Rep THEN Sets.Assign(Toks, Nont[N].First)
        ELSE Sets.Empty(Toks)
        END;
        F := A.Last; IsLast := TRUE;
        WHILE F # NIL DO
          IF F IS EAG.Term THEN
            Sets.Empty(Toks); Sets.Incl(Toks, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
            IsLast := FALSE
          ELSE
            IF Sets.In(TestNonts, F(EAG.Nont).Sym) THEN
              IF IsLast THEN NewEdge(F(EAG.Nont).Sym, N) END;
              Sets.Union(Nont[F(EAG.Nont).Sym].Follow, Nont[F(EAG.Nont).Sym].Follow, Toks);
              IF UseReg & ~ Sets.In(RegNonts, N) & Sets.In(RegNonts, F(EAG.Nont).Sym) THEN (* R *)
                Sets.Incl(Nont[F(EAG.Nont).Sym].Follow, sepTok)  (* R *)
              END;  (* R *)
              IF Sets.In(EAG.Null, F(EAG.Nont).Sym) THEN
                Sets.Union(Toks, Toks, Nont[F(EAG.Nont).Sym].First)
              ELSE Sets.Assign(Toks, Nont[F(EAG.Nont).Sym].First); IsLast := FALSE
              END
            END
          END;
          F := F.Prev
        END;
        IF IsLast THEN Sets.Incl(NullAlts, A.Ind) END;
        A := A.Next
      UNTIL A = NIL
```

```
    END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) THEN Sets.Assign(Nont[N].IniFollow, Nont[N].Follow) END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO State[N] := 0 END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) & (State[N] = 0) THEN ComputeFollow(N) END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(TestNonts, N) THEN
      Sets.Empty(Toks);
      A := EAG.HNont[N].Def.Sub;
      REPEAT
        IF Sets.In(NullAlts, A.Ind) THEN Sets.Assign(Alt[A.Ind].Dir, Nont[N].Follow)
        ELSE Sets.Empty(Alt[A.Ind].Dir)
        END;
        F := A.Sub;
        LOOP
          IF F = NIL THEN EXIT END;
          IF F IS EAG.Term THEN
            Sets.Incl(Alt[A.Ind].Dir, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok); EXIT
          ELSE
            IF Sets.In(TestNonts, F(EAG.Nont).Sym) THEN
              Sets.Union(Alt[A.Ind].Dir, Alt[A.Ind].Dir, Nont[F(EAG.Nont).Sym].First);
              IF ~ Sets.In(EAG.Null, F(EAG.Nont).Sym) THEN EXIT END
            END
          END;
          F := F.Next
        END;
        IF ~ Sets.Disjoint(Alt[A.Ind].Dir, Toks) THEN
          Conflict(N, A.Pos, Alt[A.Ind].Dir, Toks);
          Sets.Difference(Alt[A.Ind].Dir, Alt[A.Ind].Dir, Toks)
        END;
        Sets.Union(Toks, Toks, Alt[A.Ind].Dir);
        A := A.Next
      UNTIL A = NIL;
      IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN
        IF ~ Sets.Disjoint(Nont[N].Follow, Toks) THEN
          IF ~ UseReg OR ~ Sets.In(ConflictNonts, N) OR Sets.In(Toks, sepTok) THEN  (* R *)
            IF EAG.HNont[N].Def IS EAG.Opt THEN
              Conflict(N, EAG.HNont[N].Def(EAG.Opt).EmptyAltPos, Nont[N].Follow, Toks)
            ELSE Conflict(N, EAG.HNont[N].Def(EAG.Rep).EmptyAltPos, Nont[N].Follow, Toks)
            END
          END  (* R *)
        END
      END
    END
  END
END ComputeDir;

PROCEDURE ComputeDefaultAlts;
  VAR
    N : INTEGER; A : EAG.Alt; F : EAG.Factor;
    E, APrio : INTEGER;
    Alt : POINTER TO ARRAY OF RECORD Nont, Deg, Prio : INTEGER; Alt : EAG.Alt END;
    Stack : POINTER TO ARRAY OF RECORD Nont, APrio : INTEGER; Alt : EAG.Alt END; Top : INTEGER;
    StackPos : POINTER TO ARRAY OF INTEGER;
    DefNonts : Sets.OpenSet;

  PROCEDURE TestDeg(AInd: INTEGER);
    VAR N, i : INTEGER;
  BEGIN
    IF Alt[AInd].Deg = 0 THEN
      N := Alt[AInd].Nont; i := StackPos[N];
      IF i = MAX(INTEGER) THEN
        Stack[Top].Nont := N; Stack[Top].APrio := Alt[AInd].Prio;
        Stack[Top].Alt := Alt[AInd].Alt; StackPos[N] := Top; INC(Top)
      ELSIF (i >= 0) & (Stack[i].APrio > Alt[AInd].Prio) THEN
        Stack[i].APrio := Alt[AInd].Prio; Stack[i].Alt := Alt[AInd].Alt
      END
    END
  END TestDeg;

  PROCEDURE Pop(VAR Edge : INTEGER);
    VAR i, MinPrio, MinPos : INTEGER;
  BEGIN
    i := Top; DEC(Top); MinPrio := MAX(INTEGER);
    REPEAT DEC(i); IF Stack[i].APrio < MinPrio THEN MinPrio := Stack[i].APrio; MinPos := i END
    UNTIL (i = 0) OR (MinPrio = 1);
```

```
    Nont[Stack[MinPos].Nont].DefaultAlt := Stack[MinPos].Alt;
    Edge := Nont[Stack[MinPos].Nont].Edge;
    StackPos[Stack[Top].Nont] := MinPos; StackPos[Stack[MinPos].Nont] := -1;
    Stack[MinPos] := Stack[Top]
  END Pop;

BEGIN
  IF EAG.NextHAlt >= 1 THEN NEW(Alt, EAG.NextHAlt) END;
  IF EAG.NextHNont >= 1 THEN NEW(Stack, EAG.NextHNont) END; Top := 0;
  IF EAG.NextHNont >= 1 THEN NEW(StackPos, EAG.NextHNont) END;
  Sets.New(DefNonts, EAG.NextHNont); Sets.Assign(DefNonts, GenNonts);
  NextEdge := firstEdge;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    Nont[N].Edge := nil; Nont[N].DefaultAlt := NIL; StackPos[N] := MAX(INTEGER);
    IF Sets.In(GenNonts, N) &
      ((EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep)) THEN
      Sets.Excl(DefNonts, N)
    END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(DefNonts, N) THEN
      A := EAG.HNont[N].Def.Sub; APrio := 1;
      REPEAT
        Alt[A.Ind].Nont := N; Alt[A.Ind].Alt := A; Alt[A.Ind].Deg := 0; Alt[A.Ind].Prio := APrio;
        F := A.Sub;
        WHILE F # NIL DO
          IF (F IS EAG.Nont) & Sets.In(DefNonts, F(EAG.Nont).Sym) THEN
            INC(Alt[A.Ind].Deg); NewEdge(F(EAG.Nont).Sym, A.Ind)
          END;
          F := F.Next
        END;
        TestDeg(A.Ind);
        A := A.Next; INC(APrio)
      UNTIL A = NIL
    END
  END;
  WHILE Top > 0 DO
    Pop(E);
    WHILE E # nil DO
      DEC(Alt[Edge[E].Dest].Deg); TestDeg(Edge[E].Dest); E := Edge[E].Next
    END
  END
END ComputeDefaultAlts;

PROCEDURE ComputeSets;
  VAR N : INTEGER; Start : Sets.OpenSet;

  PROCEDURE NewGenSet(Toks : Sets.OpenSet; VAR GenSetIndex : INTEGER);
    VAR i : INTEGER;
  BEGIN
    i := firstGenSet;
    WHILE (i < NextGenSet) & (~ Sets.Equal(GenSet[i], Toks)) DO INC(i) END;
    GenSetIndex := i;
    IF i = NextGenSet THEN
      IF NextGenSet >= LEN(GenSet^) THEN Expand END;
      Sets.New(GenSet[NextGenSet], nToks); Sets.Assign(GenSet[NextGenSet], Toks);
      INC(NextGenSet)
    END
  END NewGenSet;

  PROCEDURE NewGenSetT(Toks : Sets.OpenSet; VAR GenSetTIndex : INTEGER);
    VAR i : INTEGER;
  BEGIN
    i := firstGenSetT;
    WHILE (i < NextGenSetT) & (~ Sets.Equal(GenSetT[i], Toks)) DO INC(i) END;
    GenSetTIndex := i;
    IF i = NextGenSetT THEN
      IF NextGenSetT >= LEN(GenSetT^) THEN Expand END;
      Sets.New(GenSetT[NextGenSetT], nToks); Sets.Assign(GenSetT[NextGenSetT], Toks);
      INC(NextGenSetT)
    END
  END NewGenSetT;

  PROCEDURE ComputeRecoverySets(N : INTEGER; VAR LocalRec : Sets.OpenSet);
    VAR A : EAG.Alt; F : EAG.Factor; S : Sets.OpenSet; RealAlt : BOOLEAN;
  BEGIN
    Sets.New(S, nToks);
    A := EAG.HNont[N].Def.Sub; RealAlt := A.Next # NIL;
    REPEAT
      (* this is what Grosch proposes: Sets.Assign(S, LocalRec); *)
```

```
      IF EAG.HNont[N].Def IS EAG.Rep THEN Sets.Union(S, LocalRec, Nont[N].First)
      ELSE Sets.Assign(S, LocalRec)
      END;
      F := A.Last;
      WHILE F # NIL DO
        IF F IS EAG.Term THEN
          Sets.Incl(S, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
          NewGenSet(S, Factor[F.Ind].Rec)
        ELSE
          IF Sets.In(GenNonts, F(EAG.Nont).Sym) THEN
            IF ~ Nont[F(EAG.Nont).Sym].Anonym THEN
              IF UseReg & ~ Sets.In(RegNonts, N) & Sets.In(RegNonts, F(EAG.Nont).Sym) THEN (* R *)
                Sets.Incl(S, sepTok)  (* R *)
              END;  (* R *)
              NewGenSet(S, Factor[F.Ind].Rec);
              Sets.Union(S, S, Nont[F(EAG.Nont).Sym].First)
            ELSE ComputeRecoverySets(F(EAG.Nont).Sym, S)
            END
          END
        END;
        F := F.Prev
      END;
      A := A.Next
    UNTIL A = NIL;
    Sets.Union(LocalRec, LocalRec, Nont[N].First);
    IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN
      NewGenSet(LocalRec, Nont[N].OptRec);
    END;
    IF RealAlt THEN NewGenSet(LocalRec, Nont[N].AltRec) END
  END ComputeRecoverySets;

BEGIN (* ComputeSets *)
  Sets.New(Start, nToks);
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, N) THEN
      Sets.Empty(Start); IF N = EAG.StartSym THEN Sets.Incl(Start, endTok) END;
      IF ~ Nont[N].Anonym THEN ComputeRecoverySets(N, Start) END;
      IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN
        IF ~ Nont[N].Anonym THEN NewGenSet(Nont[N].First, Nont[N].OptExp)
        ELSE
          Sets.Union(Start, Nont[N].First, Nont[N].IniFollow);
          NewGenSet(Start, Nont[N].OptExp)
        END;
        NewGenSetT(Nont[N].First, Nont[N].FirstIndex);
        NewGenSetT(Nont[N].Follow, Nont[N].FollowIndex)
      END;
      IF EAG.HNont[N].Def.Sub.Next # NIL THEN
        NewGenSet(Nont[N].First, Nont[N].AltExp)
      END
    END
  END
END ComputeSets;

PROCEDURE GenerateMod(ParsePass : BOOLEAN);
  VAR
    Mod : IO.TextOut; Fix : IO.TextIn;
    N : INTEGER;
    Tok : INTEGER; AllToks : Sets.OpenSet;
    Name : ARRAY EAG.BaseNameLen + 10 OF CHAR;
    OpenError, CompileError : BOOLEAN;
    TabTimeStamp : LONGINT;

  PROCEDURE TraverseNont(N : INTEGER; FirstNontCall : BOOLEAN; Poss : Sets.OpenSet);
    VAR ExactOneToken : BOOLEAN; TheOneToken : INTEGER;

    PROCEDURE TraverseAlts(A : EAG.Alt; FirstNontCall : BOOLEAN; Poss : Sets.OpenSet);
      VAR Tok : INTEGER; Toks : Sets.OpenSet;
          FirstTok, LoopNeeded : BOOLEAN;

      PROCEDURE TraverseFactors(F : EAG.Factor; FirstNontCall : BOOLEAN; Poss : Sets.OpenSet);
        VAR Poss1 : Sets.OpenSet; TwoCalls : BOOLEAN;
      BEGIN
        TwoCalls := FALSE;
        Sets.New(Poss1, nToks); Sets.Assign(Poss1, Poss);
        WHILE F # NIL DO
          IF F IS EAG.Term THEN
            IF Sets.In(Poss1, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok) THEN
              Sets.Excl(Poss1, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
              IF Sets.IsEmpty(Poss1) THEN
                IO.WriteText(Mod, "S.Get(Tok); IsRepairMode := FALSE;\n")
```

```
          ELSE
            IO.WriteText(Mod, "IF Tok # ");
            IO.WriteInt(Mod, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
            IO.WriteText(Mod, " THEN RecoveryTerminal(");
            IO.WriteInt(Mod, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
            IO.WriteText(Mod, ", ");
            IO.WriteInt(Mod, Factor[F.Ind].Rec - firstGenSet);
            IO.WriteText(Mod, ")\n");
            IO.WriteText(Mod, "ELSE S.Get(Tok); IsRepairMode := FALSE\n");
            IO.WriteText(Mod, "END;\n");
          END
        ELSE
          IO.WriteText(Mod, "RecoveryTerminal(");
          IO.WriteInt(Mod, F(EAG.Term).Sym - EAG.firstHTerm + firstUserTok);
          IO.WriteText(Mod, ", ");
          IO.WriteInt(Mod, Factor[F.Ind].Rec - firstGenSet);
          IO.WriteText(Mod, ");\n");
        END;
        Sets.Assign(Poss1, AllToks)
      ELSE
        IF Sets.In(GenNonts, F(EAG.Nont).Sym) THEN
        EvalGen.GenSynPred(N, F(EAG.Nont).Actual.Params);
        IF ~ Nont[F(EAG.Nont).Sym].Anonym THEN
          IF FirstNontCall THEN
            IO.WriteText(Mod, "IF RecTop >= LEN(RecStack^) THEN ParserExpand END;\n");
            FirstNontCall := FALSE
          END;
          IF TwoCalls THEN IO.WriteText(Mod, "RecStack[RecTop - 1] := ")
          ELSE IO.WriteText(Mod, "RecStack[RecTop] := ")
          END;
          IO.WriteInt(Mod, Factor[F.Ind].Rec - firstGenSet);
          IF TwoCalls THEN IO.WriteText(Mod, ";\n")
          ELSE IO.WriteText(Mod, "; INC(RecTop);\n")
          END;
          IF UseReg & ~ Sets.In(RegNonts, N) & Sets.In(RegNonts, F(EAG.Nont).Sym) THEN (* R *)
            IO.WriteText(Mod, "S.Get := S.Get3;\n")  (* R *)
          END;  (* R *)
          IO.WriteText(Mod, "P"); IO.WriteInt(Mod, F(EAG.Nont).Sym);
          EvalGen.GenActualParams(F(EAG.Nont).Actual.Params, TRUE);
          IO.WriteText(Mod, ";");
          IO.WriteText(Mod, "   (* "); EAG.WriteHNont(Mod, F(EAG.Nont).Sym);
          IO.WriteText(Mod, " *)\n");
          IF UseReg & ~ Sets.In(RegNonts, N) & Sets.In(RegNonts, F(EAG.Nont).Sym) THEN (* R *)
            IO.WriteText(Mod, "IF Tok = sepTok THEN S.Get(Tok); ");  (* R *)
            IO.WriteText(Mod, "IsRepairMode := FALSE END;\n");  (* R *)
            IO.WriteText(Mod, "S.Get := S.Get2;\n")  (* R *)
          END;  (* R *)
          IF (F.Next # NIL) & (F.Next IS EAG.Nont) &
            Sets.In(GenNonts, F.Next(EAG.Nont).Sym) &
            (~ Nont[F.Next(EAG.Nont).Sym].Anonym) THEN TwoCalls := TRUE
          ELSE TwoCalls := FALSE
          END;
          IF ~ TwoCalls THEN IO.WriteText(Mod, "DEC(RecTop);\n") END
        ELSE TraverseNont(F(EAG.Nont).Sym, FirstNontCall, Poss1)
        END;
        EvalGen.GenAnalPred(N, F(EAG.Nont).Actual.Params);
        Sets.Assign(Poss1, AllToks)
      ELSIF Sets.In(EAG.Pred, F(EAG.Nont).Sym) THEN
        EvalGen.GenSynPred(N, F(EAG.Nont).Actual.Params);
        EvalGen.GenPredCall(F(EAG.Nont).Sym, F(EAG.Nont).Actual.Params);
        EvalGen.GenAnalPred(N, F(EAG.Nont).Actual.Params)
      ELSE Warning := TRUE;
        IO.WriteText(Mod,
          "IO.WriteText(IO.Msg, \'  runtime error: call of nonproductive nonterminal!\\n\');\n");
        IO.WriteText(Mod, "IO.Update(IO.Msg); HALT(99);\n");
        IO.WriteText(IO.Msg, "\n  generated compiler containes corrupt code");
        IO.WriteText(IO.Msg, "\n    for nonproductive nonterminals!"); IO.Update(IO.Msg)
      END
    END;
    F := F.Next
  END
END TraverseFactors;

BEGIN (* TraverseAlts *)
  IF A.Next = NIL THEN
    EvalGen.InitScope(A.Scope);
    EvalGen.GenAnalPred(N, A.Formal.Params);
    TraverseFactors(A.Sub, FirstNontCall, Poss);
    IF EAG.HNont[N].Def IS EAG.Rep THEN EvalGen.GenRepAlt(N,A)
    ELSE EvalGen.GenSynPred(N, A.Formal.Params)
```

```
      END
    ELSE
      Sets.New(Toks, nToks);
      IF ~ Sets.In(EAG.Null, N) THEN Sets.Assign(Toks, Nont[N].First)
      ELSE Sets.Union(Toks, Nont[N].First, Nont[N].Follow)
      END;
      IF Sets.Included(Toks, Poss) THEN LoopNeeded := FALSE
      ELSE LoopNeeded := TRUE
      END;
      IF LoopNeeded THEN IO.WriteText(Mod, "LOOP\n") END;
      IO.WriteText(Mod, "\tCASE Tok OF\n");
      REPEAT
        IF (~ LoopNeeded) & Sets.Disjoint(Alt[A.Ind].Dir, Poss) THEN
          IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, A.Pos);
          IO.WriteText(IO.Msg, "  dead alternative in ");
          EAG.WriteNamedHNont(IO.Msg, N); IO.Update(IO.Msg); Warning := TRUE
        END;
        IO.WriteText(Mod, "\t| "); FirstTok := TRUE;
        FOR Tok := 0 TO nToks - 1 DO
          IF Sets.In(Alt[A.Ind].Dir, Tok) THEN
            IF ~ FirstTok THEN IO.WriteText(Mod, ", ") END;
            IO.WriteInt(Mod, Tok); FirstTok := FALSE
          END
        END;
        IO.WriteText(Mod, " : \n");
        EvalGen.InitScope(A.Scope);
        EvalGen.GenAnalPred(N, A.Formal.Params);
        TraverseFactors(A.Sub, FirstNontCall, Alt[A.Ind].Dir);
        IF EAG.HNont[N].Def IS EAG.Rep THEN EvalGen.GenRepAlt(N,A)
        ELSE EvalGen.GenSynPred(N, A.Formal.Params)
        END;
        IF LoopNeeded THEN IO.WriteText(Mod, "\t\tEXIT\n") END;
        A := A.Next
      UNTIL A = NIL;
      IF LoopNeeded THEN
        A := Nont[N].DefaultAlt;
        IO.WriteText(Mod, "\tELSE\n");
        IO.WriteText(Mod, "\t\tIF IsRepairMode THEN\n");
        Sets.Difference(Toks, AllToks, Toks);
        EvalGen.InitScope(A.Scope);
        EvalGen.GenAnalPred(N, A.Formal.Params);
        TraverseFactors(A.Sub, FirstNontCall, Toks);
        IF EAG.HNont[N].Def IS EAG.Rep THEN EvalGen.GenRepAlt(N, A)
        ELSE EvalGen.GenSynPred(N, A.Formal.Params)
        END;
        IO.WriteText(Mod, "\t\t\tEXIT END;\n");
        IO.WriteText(Mod, "\t\tErrorRecovery(");
        IO.WriteInt(Mod, Nont[N].AltExp - firstGenSet);
        IO.WriteText(Mod, ", ");
        IO.WriteInt(Mod, Nont[N].AltRec - firstGenSet);
        IO.WriteText(Mod, ")\n");
      END;
      IO.WriteText(Mod, "\tEND;\n");
      IF LoopNeeded THEN IO.WriteText(Mod, "END;\n") END
    END
  END TraverseAlts;

  PROCEDURE TestOneToken(Toks : Sets.OpenSet; VAR ExactOneToken : BOOLEAN;
                         VAR TheOneToken : INTEGER);
    VAR Tok : INTEGER;
  BEGIN
    Tok := 0; ExactOneToken := FALSE;
    WHILE Tok < nToks DO
      IF Sets.In(Toks, Tok) THEN
        IF ExactOneToken THEN ExactOneToken := FALSE; RETURN
        ELSE ExactOneToken := TRUE; TheOneToken := Tok
        END
      END;
      INC(Tok);
    END
  END TestOneToken;

BEGIN (* TraverseNont *)
  IF EAG.HNont[N].Def IS EAG.Opt THEN
    IF Sets.Included(Nont[N].Follow, Poss) & Sets.Disjoint(Nont[N].First, Poss) THEN
      IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, EAG.HNont[N].Def.Sub.Pos);
      IO.WriteText(IO.Msg, "  dead [ ] - brackets in ");
      EAG.WriteNamedHNont(IO.Msg, N); IO.Update(IO.Msg); Warning := TRUE
    ELSIF Sets.Included(Nont[N].First, Poss) THEN
      IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, EAG.HNont[N].Def.Sub.Pos);
```

```
      IO.WriteText(IO.Msg, "  useless [ ] - brackets in ");
      EAG.WriteNamedHNont(IO.Msg, N); IO.Update(IO.Msg); Warning := TRUE
    END;
    IO.WriteText(Mod, "LOOP\n");
    IO.WriteText(Mod, "\tIF ");
    TestOneToken(Nont[N].First, ExactOneToken, TheOneToken);
    IF ExactOneToken THEN
      IO.WriteText(Mod, "Tok = "); IO.WriteInt(Mod, TheOneToken)
    ELSE
      IO.WriteInt(Mod, (Nont[N].FirstIndex - firstGenSetT) MOD nElemsPerSET);
      IO.WriteText(Mod, " IN SetT[");
      IO.WriteInt(Mod, (Nont[N].FirstIndex - firstGenSetT) DIV nElemsPerSET);
      IO.WriteText(Mod, "][Tok]");
    END;
    IO.WriteText(Mod, " THEN\n");
    TraverseAlts(EAG.HNont[N].Def.Sub, FirstNontCall, Nont[N].First);
    IO.WriteText(Mod, "\t\tEXIT\n");
    IO.WriteText(Mod, "\tELSIF (");
    TestOneToken(Nont[N].Follow, ExactOneToken, TheOneToken);
    IF ExactOneToken THEN
      IO.WriteText(Mod, "Tok = "); IO.WriteInt(Mod, TheOneToken)
    ELSE
      IO.WriteInt(Mod, (Nont[N].FollowIndex - firstGenSetT) MOD nElemsPerSET);
      IO.WriteText(Mod, " IN SetT[");
      IO.WriteInt(Mod, (Nont[N].FollowIndex - firstGenSetT) DIV nElemsPerSET);
      IO.WriteText(Mod, "][Tok]");
    END;
    IO.WriteText(Mod, ") OR IsRepairMode THEN\n");
    EvalGen.InitScope(EAG.HNont[N].Def(EAG.Opt).Scope);
    EvalGen.GenAnalPred(N, EAG.HNont[N].Def(EAG.Opt).Formal.Params);
    EvalGen.GenSynPred(N, EAG.HNont[N].Def(EAG.Opt).Formal.Params);
    IO.WriteText(Mod, "\t\tEXIT\n");
    IO.WriteText(Mod, "\tEND;\n");
    IO.WriteText(Mod, "\tErrorRecovery(");
    IO.WriteInt(Mod, Nont[N].OptExp - firstGenSet);
    IO.WriteText(Mod, ", ");
    IO.WriteInt(Mod, Nont[N].OptRec - firstGenSet);
    IO.WriteText(Mod, ")\n");
    IO.WriteText(Mod, "END;\n");
  ELSIF EAG.HNont[N].Def IS EAG.Rep THEN
    IF Sets.Included(Nont[N].Follow, Poss) & Sets.Disjoint(Nont[N].First, Poss) THEN
      IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, EAG.HNont[N].Def.Sub.Pos);
      IO.WriteText(IO.Msg, "  dead { } - braces in ");
      EAG.WriteNamedHNont(IO.Msg, N); IO.Update(IO.Msg); Warning := TRUE
    END;
    EvalGen.GenRepStart(N);
    IO.WriteText(Mod, "LOOP\n");
    IO.WriteText(Mod, "\tIF ");
    TestOneToken(Nont[N].First, ExactOneToken, TheOneToken);
    IF ExactOneToken THEN
      IO.WriteText(Mod, "Tok = "); IO.WriteInt(Mod, TheOneToken)
    ELSE
      IO.WriteInt(Mod, (Nont[N].FirstIndex - firstGenSetT) MOD nElemsPerSET);
      IO.WriteText(Mod, " IN SetT[");
      IO.WriteInt(Mod, (Nont[N].FirstIndex - firstGenSetT) DIV nElemsPerSET);
      IO.WriteText(Mod, "][Tok]");
    END;
    IO.WriteText(Mod, " THEN\n");
    TraverseAlts(EAG.HNont[N].Def.Sub, FirstNontCall, Nont[N].First);
    IO.WriteText(Mod, "\tELSIF (");
    TestOneToken(Nont[N].Follow, ExactOneToken, TheOneToken);
    IF ExactOneToken THEN
      IO.WriteText(Mod, "Tok = "); IO.WriteInt(Mod, TheOneToken)
    ELSE
      IO.WriteInt(Mod, (Nont[N].FollowIndex - firstGenSetT) MOD nElemsPerSET);
      IO.WriteText(Mod, " IN SetT[");
      IO.WriteInt(Mod, (Nont[N].FollowIndex - firstGenSetT) DIV nElemsPerSET);
      IO.WriteText(Mod, "][Tok]");
    END;
    IO.WriteText(Mod, ") OR IsRepairMode THEN EXIT\n");
    IO.WriteText(Mod, "\tELSE ErrorRecovery(");
    IO.WriteInt(Mod, Nont[N].OptExp - firstGenSet);
    IO.WriteText(Mod, ", ");
    IO.WriteInt(Mod, Nont[N].OptRec - firstGenSet);
    IO.WriteText(Mod, ")\n");
    IO.WriteText(Mod, "\tEND\n");
    IO.WriteText(Mod, "END;\n");
    EvalGen.GenRepEnd(N)
  ELSE TraverseAlts(EAG.HNont[N].Def.Sub, FirstNontCall, Poss);
  END
```

```
  END TraverseNont;

  PROCEDURE WriteTab(Name : ARRAY OF CHAR);
    CONST magicNumber = 314C6C45H;
    VAR Tab : IO.File; i, j, m, Tok : INTEGER; s : SET;
  BEGIN
    IO.CreateFile(Tab, Name);
    IO.PutLInt(Tab, magicNumber);
    IO.PutLInt(Tab, TabTimeStamp);
    IO.PutLInt(Tab, MAX(SET));
    IO.PutSet(Tab, {0,2,3,6,9,13,18,19,20,24,25,27,28,31});
    FOR i := firstGenSetT TO NextGenSetT - 1 BY nElemsPerSET DO
      IF nElemsPerSET <= NextGenSetT - i THEN m := nElemsPerSET
      ELSE m := NextGenSetT - i
      END;
      FOR Tok := 0 TO nToks - 1 DO
        s := {};
        FOR j := 0 TO m - 1 DO
          IF Sets.In(GenSetT[i + j], Tok) THEN INCL(s, j) END
        END;
        IO.PutSet(Tab, s)
      END
    END;
    FOR i := firstGenSet TO NextGenSet - 1 DO
      FOR j := 0 TO Sets.nSetsUsed(GenSet[i]) - 1 DO
        IO.PutSet(Tab, Sets.ConvertToSET(GenSet[i], j))
      END
    END;
    IO.PutLInt(Tab, magicNumber);
    IO.CloseFile(Tab)
  END WriteTab;

  PROCEDURE InclFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = OX THEN
        IO.WriteText(IO.Msg, "\n  error: unexpected end of eELL1Gen.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Write(Mod, c); IO.Read(Fix, c)
    END
  END InclFix;

  PROCEDURE Append(VAR Dest : ARRAY OF CHAR; Src, Suf : ARRAY OF CHAR);
    VAR i, j : INTEGER;
  BEGIN
    i := 0; j := 0;
    WHILE (Src[i] # OX) & (i < LEN(Dest) - 1) DO Dest[i] := Src[i]; INC(i) END;
    WHILE (Suf[j] # OX) & (i < LEN(Dest) - 1) DO Dest[i] := Suf[j]; INC(i); INC(j) END;
    Dest[i] := OX
  END Append;

BEGIN (* GenerateMod(ParsePass : BOOLEAN) *)
  Sets.New(AllToks, nToks);
  IO.OpenIn(Fix, "eELL1Gen.Fix", OpenError);
  IF OpenError THEN
    IO.WriteText(IO.Msg, "\n  error: could not open eELL1Gen.Fix\n"); IO.Update(IO.Msg); HALT(99)
  END;
  IO.CreateModOut(Mod, EAG.BaseName);
  IF ParsePass THEN EvalGen.InitGen(Mod, EvalGen.parsePass)
  ELSE EvalGen.InitGen(Mod, EvalGen.onePass)
  END;
  InclFix("$"); IO.WriteText(Mod, EAG.BaseName);
  InclFix("$"); Append(Name, EAG.BaseName, "Scan"); IO.WriteText(Mod, Name);
  IF ParsePass THEN
    IO.WriteText(Mod, ", Eval := ");
    IO.WriteText(Mod, EAG.BaseName); IO.WriteText(Mod, "Eval");
  END;
  InclFix("$"); IO.WriteInt(Mod, nToks);
  InclFix("$"); IO.WriteInt(Mod, Sets.nSetsUsed(AllToks));
  InclFix("$"); IO.WriteInt(Mod, (NextGenSetT - firstGenSetT - 1) DIV nElemsPerSET + 1);
  InclFix("$"); IO.WriteInt(Mod, NextGenSet - firstGenSet);
  InclFix("$");
  EvalGen.GenDeclarations;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, N) THEN
      IF ~ Nont[N].Anonym THEN
        IO.WriteText(Mod, "PROCEDURE^ P"); IO.WriteInt(Mod, N);
```

```
        EvalGen.GenFormalParams(N, TRUE); IO.WriteText(Mod, ";");
        IO.WriteText(Mod, "   (* "); EAG.WriteHNont(Mod, N); IO.WriteText(Mod, " *)\n");
      END
    END
  END;
  EvalGen.GenPredProcs;
  InclFix("$"); TabTimeStamp := IO.TimeStamp(); IO.WriteInt(Mod, TabTimeStamp);
  InclFix("$");
  Sets.Empty(AllToks);
  FOR Tok := 0 TO nToks - 1 DO Sets.Incl(AllToks, Tok) END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, N) THEN
      IF ~ Nont[N].Anonym THEN
        EvalGen.ComputeVarNames(N, TRUE);
        IO.WriteText(Mod, "PROCEDURE P"); IO.WriteInt(Mod, N);
        EvalGen.GenFormalParams(N, TRUE); IO.WriteText(Mod, ";");
        IO.WriteText(Mod, "   (* "); EAG.WriteHNont(Mod, N); IO.WriteText(Mod, " *)\n");
        EvalGen.GenVarDecl(N);
        IO.WriteText(Mod, "BEGIN\n");
        TraverseNont(N, TRUE, AllToks);
        IO.WriteText(Mod, "END P"); IO.WriteInt(Mod, N); IO.WriteText(Mod, ";\n\n")
      END
    END
  END;
  IF ~ ParsePass THEN EmitGen.GenEmitProc(Mod) END;
  InclFix("$"); IF ParsePass THEN IO.WriteText(Mod, "& Eval.EvalInitSucceeds()") END;
  InclFix("$"); IO.WriteText(Mod, EAG.BaseName);
  InclFix("$"); IO.WriteText(Mod, "P"); IO.WriteInt(Mod, EAG.StartSym);
  InclFix("$");
  IF ParsePass THEN
    IO.WriteText(Mod, "Eval.TraverseSyntaxTree(Heap, PosHeap, ErrorCounter, V1, arityConst);\n");
    IO.WriteText(Mod, '\t\tIF IO.IsOption("i") THEN \n');
    IO.WriteText(Mod, '\t\t\tIO.WriteText(IO.Msg, "\\tsyntax tree uses twice ");\n');
    IO.WriteText(Mod, '\t\t\tIO.WriteInt(IO.Msg, NextHeap); IO.WriteLn(IO.Msg); IO.Update(IO.Msg);\n');
    IO.WriteText(Mod, '\t\tEND;')
  ELSE
    IO.WriteText(Mod, 'IF ErrorCounter > 0 THEN\n');
    IO.WriteText(Mod, '\t\t\tIO.WriteText(IO.Msg, "  "); IO.WriteInt(IO.Msg, ErrorCounter);\n');
    IO.WriteText(Mod, '\t\t\tIO.WriteText(IO.Msg, " errors detected\\n"); IO.Update(IO.Msg);\n');
    IO.WriteText(Mod, '\t\tELSE\n');
    EmitGen.GenEmitCall(Mod);
    IO.WriteText(Mod, '\t\tEND;\n');
    EmitGen.GenShowHeap(Mod)
  END;
  InclFix("$"); IO.WriteText(Mod, EAG.BaseName);
  InclFix("$"); Append(Name, EAG.BaseName, ".Tab"); IO.WriteText(Mod, Name);
  InclFix("$"); IO.WriteText(Mod, EAG.BaseName);
  InclFix("$");
  Append(Name, EAG.BaseName, ".Tab"); WriteTab(Name);
  IO.Update(Mod);
  IF ShowMod THEN IO.Show(Mod)
  ELSE IO.Compile(Mod, CompileError); Compiled := TRUE;
    IF CompileError THEN IO.Show(Mod) END
  END;
  IO.CloseIn(Fix); IO.CloseOut(Mod); EvalGen.FinitGen
END GenerateMod;

PROCEDURE Test*;
BEGIN
  IO.WriteText(IO.Msg, "ELL(1) testing    "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.Update(IO.Msg);
  IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
    EXCL(EAG.History, EAG.parsable);
    Init;
    IF GrammarOk() THEN  (* R *)
      ComputeDir;
      IF ~ (Error OR Warning) THEN
        IO.WriteText(IO.Msg, "   ok");
        INCL(EAG.History, EAG.parsable)
      END
    END;  (* R *)
    Finit
  END;
  IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END Test;

PROCEDURE Generate*;
BEGIN
  IO.WriteText(IO.Msg, "ELL(1) writing   "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteText(IO.Msg, "   "); IO.Update(IO.Msg); Compiled := FALSE;
```

```
  IF EAG.Performed({EAG.analysed, EAG.predicates, EAG.isSLEAG}) THEN
    EXCL(EAG.History, EAG.parsable);
    Init;
    IF GrammarOk() THEN  (* R *)
      ComputeDir;
      IF ~ Error THEN
        ComputeDefaultAlts;
        ComputeSets;
        GenerateMod(FALSE);
        INCL(EAG.History, EAG.parsable)
      END
    END;  (* R *)
    Finit
  END;
  IF ~ Compiled THEN IO.WriteLn(IO.Msg) END; IO.Update(IO.Msg)
END Generate;

PROCEDURE GenerateParser*;
BEGIN
  IO.WriteText(IO.Msg, "ELL(1) writing parser of "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteText(IO.Msg, "   "); IO.Update(IO.Msg); Compiled := FALSE;
  IF EAG.Performed({EAG.analysed, EAG.predicates, EAG.hasEvaluator}) THEN
    EXCL(EAG.History, EAG.parsable);
    Init;
    IF GrammarOk() THEN  (* R *)
      EAG.History := {}; Shift.Shift(0);  (* dummy parameter *)
      ComputeDir;
      IF ~ Error THEN
        ComputeDefaultAlts;
        ComputeSets;
        GenerateMod(TRUE)
      END;
    END;  (* R *)
    Finit
  END;
  IF ~ Compiled THEN IO.WriteLn(IO.Msg) END; IO.Update(IO.Msg)
END GenerateParser;

END eELL1Gen.
```

### 5.3.4 eELL1Gen.Fix

```
MODULE $;    (* eELL1Gen.Fix, SteWe 08/96  -  Ver 1.2 *)

IMPORT IO := eIO, S := $;

CONST
  nToks = $;
  tokSetLen = $;
  nSetT = $;
  nSet = $;
  M = MAX(SET) + 1;
  endTok = 0; sepTok = 2;  (* R *)
  firstRecStack = 1;

TYPE
  OpenRecStack = POINTER TO ARRAY OF INTEGER;
  TokSet = ARRAY tokSetLen OF SET;

VAR
  Tok : INTEGER;
  RecStack : OpenRecStack; RecTop : INTEGER;
  SetT : ARRAY nSetT + 1, nToks OF SET;
  Set : ARRAY nSet + 1 OF TokSet;
  ErrorCounter : LONGINT;
  IsRepairMode : BOOLEAN;
  LongErrorMsgs : BOOLEAN;
  ParserTabIsLoaded : BOOLEAN;
  Out: IO.TextOut;

$

PROCEDURE ParserExpand;
  VAR
    RecStack1 : OpenRecStack;
    i : LONGINT;

  PROCEDURE ExpLen(ArrayLen : LONGINT) : LONGINT;
  BEGIN
    IF ArrayLen <= MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen ELSE HALT(99) END
  END ExpLen;

BEGIN
  IF RecTop >= LEN(RecStack^) THEN NEW(RecStack1, ExpLen(LEN(RecStack^)));
    FOR i := firstRecStack TO LEN(RecStack^) - 1 DO RecStack1[i] := RecStack[i] END;
    RecStack := RecStack1
  END;
END ParserExpand;

PROCEDURE ReadParserTab(Name : ARRAY OF CHAR);
  CONST magicNumber  = 827092037;
        tabTimeStamp = $;
  VAR
    Tab : IO.File; OpenError : BOOLEAN;
    i, j : INTEGER; l : LONGINT; s : SET;

  PROCEDURE LoadError(Msg : ARRAY OF CHAR);
  BEGIN
    IO.WriteText(IO.Msg, "  loading the parser table "); IO.WriteString(IO.Msg, Name);
    IO.WriteText(IO.Msg, " failed\n"); IO.WriteText(IO.Msg, "  ");
    IO.WriteText(IO.Msg, Msg); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END LoadError;

BEGIN
  IO.OpenFile(Tab, Name, OpenError);
  IF OpenError THEN LoadError("it could not be opened"); RETURN END;
  IO.GetLInt(Tab, l);
  IF l # magicNumber THEN LoadError("no or corrupt parser table"); RETURN END;
  IO.GetLInt(Tab, l);
  IF l # tabTimeStamp THEN LoadError("wrong time stamp"); RETURN END;
  IO.GetLInt(Tab, l);
  IF l # MAX(SET) THEN LoadError("incompatible MAX(SET) in table"); RETURN END;
  IO.GetSet(Tab, s);
  IF s # {0,2,3,6,9,13,18,19,20,24,25,27,28,31} THEN
    LoadError("incompatible SET format in table"); RETURN
  END;
  FOR i := 0 TO nSetT - 1 DO
    FOR j := 0 TO nToks - 1 DO IO.GetSet(Tab, SetT[i][j]) END
  END;
```

```
  FOR i := 0 TO nSet - 1 DO
    FOR j := 0 TO tokSetLen - 1 DO IO.GetSet(Tab, Set[i][j]) END
  END;
  IO.GetLInt(Tab, l);
  IF l # magicNumber THEN LoadError("corrupt parser table"); RETURN END;
  ParserTabIsLoaded := TRUE
END ReadParserTab;

PROCEDURE ParserInit;
BEGIN
  RecTop := firstRecStack;
  ErrorCounter := 0;
  IsRepairMode := FALSE;
  LongErrorMsgs := IO.IsOption("v")
END ParserInit;

PROCEDURE WriteTokSet(Toks : TokSet);
  VAR Tok1 : INTEGER;
BEGIN
  FOR Tok1 := 0 TO nToks - 1 DO
    IF (Tok1 MOD M) IN Toks[Tok1 DIV M] THEN
      S.WriteRepr(IO.Msg, Tok1); IO.WriteText(IO.Msg, " ")
    END
  END
END WriteTokSet;

PROCEDURE ErrorMessageTok(Pos : IO.Position; Tok1 : INTEGER);
BEGIN
  IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, Pos);
  IO.WriteText(IO.Msg, "  syntax error, expected: ");
  S.WriteRepr(IO.Msg, Tok1); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END ErrorMessageTok;

PROCEDURE ErrorMessageTokSet(Pos : IO.Position; VAR Toks : TokSet);
BEGIN
  IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, Pos);
  IO.WriteText(IO.Msg, "  syntax error, expected: ");
  WriteTokSet(Toks); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END ErrorMessageTokSet;

PROCEDURE RestartMessage(Pos : IO.Position);
BEGIN
  IF LongErrorMsgs THEN
    IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, Pos);
    IO.WriteText(IO.Msg, "      restart point\n"); IO.Update(IO.Msg)
  END
END RestartMessage;

PROCEDURE RepairMessage(Pos : IO.Position; Tok1 : INTEGER);
BEGIN
  IF LongErrorMsgs THEN
    IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, Pos);
    IO.WriteText(IO.Msg, "      symbol inserted: ");
    S.WriteRepr(IO.Msg, Tok1); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END
END RepairMessage;

PROCEDURE SkipTokens(Recover : INTEGER);
  VAR GlobalRecoverySet : TokSet; i, j : INTEGER;
BEGIN
  GlobalRecoverySet := Set[Recover];
  FOR i := firstRecStack TO RecTop - 1 DO
    FOR j := 0 TO tokSetLen - 1 DO
      GlobalRecoverySet[j] := GlobalRecoverySet[j] + Set[RecStack[i]][j]
    END
  END;
  WHILE ~ ((Tok MOD M) IN GlobalRecoverySet[Tok DIV M]) DO S.Get(Tok) END;
  RestartMessage(S.Pos);
  IsRepairMode := TRUE
END SkipTokens;

PROCEDURE ErrorRecovery(Expected, Recover : INTEGER);
BEGIN
  IF ~ IsRepairMode THEN
    INC(ErrorCounter);
    ErrorMessageTokSet(S.Pos, Set[Expected]);
    SkipTokens(Recover)
  END
END ErrorRecovery;
```

```
PROCEDURE RecoveryTerminal(ExpectedTok, Recover : INTEGER);
BEGIN
  IF ~ IsRepairMode THEN
    INC(ErrorCounter);
    ErrorMessageTok(S.Pos, ExpectedTok);
    SkipTokens(Recover)
  END;
  IF Tok # ExpectedTok THEN RepairMessage(S.Pos, ExpectedTok)
  ELSE IF Tok # endTok THEN S.Get(Tok) END; IsRepairMode := FALSE
  END
END RecoveryTerminal;

$

PROCEDURE Compile*;
  VAR V1 : HeapType;
BEGIN
  IF ParserTabIsLoaded & EvalInitSucceeds() $ THEN
    IO.WriteText(IO.Msg, "$ compiler: compiling...\n"); IO.Update(IO.Msg);
    ParserInit; S.Init; S.Get(Tok);
    $(V1);
    $
  ELSIF ~ ParserTabIsLoaded THEN
    IO.WriteText(IO.Msg, "parser table is not loaded\n"); IO.Update(IO.Msg)
  END
END Compile;

BEGIN
  IO.WriteText(IO.Msg, "$ compiler (generated with Epsilon)\n"); IO.Update(IO.Msg);
  NEW(RecStack, 500); ParserTabIsLoaded := FALSE; ReadParserTab("$");
  Reset
END $.
$
```

# Kapitel 6

# Die Generierung von Evaluationscode

## 6.1 Der Evaluatorgenerator

Im generierten Compiler realisiert der Evaluator die semantische Analyse sowie evtl. die Codeerzeugung. Konzeptionell wird dazu der Ableitungsbaum zur Grundgrammatik mit zugehörigen Sätzen der Meta-Grammatik „dekoriert". Dabei werden die in der Hyper-Grammatik angegebenen Prädikate ausgewertet.Existiert keine passende Dekoration, so sollen entsprechende Kontextfehler gemeldet werden. Weiterhin sollen Folgefehler bei der Fortsetzung der Evaluation durch Reparaturen unterdrückt werden.

Es wird gefordert, daß sich die Bestimmung der Dekoration in primitive Schritte zerlegen läßt, deren Anordung durch den Generator festgelegt werden kann. Ein Verstoß gegen diese Bedingung soll vom Generator erkannt und gemeldet werden.

Nach dem Vorbild effizienter Compiler, die als Erweiterung rekursiver Abstiegsparser realisiert sind, soll die Generierung echter Ein-Pass-Compiler möglich sein. Die Affixpositionen der Hyper-Regeln werden dabei zu Prozedurparametern. Während der Ableitungsbaum implizit durch die Aufrufhierarchie repräsentiert wird, spiegeln die berechneten Parameterwerte dessen Dekoration wider. Damit die Affixberechnung mit der syntaktischen Analyse verschränkt werden kann, muß die Grammatik gewisse Bedingungen erfüllen. Insbesondere muß die transformierte Hyper-Grammatik (ohne EBNF-Operatoren) eine LEAG sein, d.h. ihre Regeln müssen alle linksdefinierend sein. Eine weitere Einschränkung ergibt sich aus der geforderten Umsetzung des EBNF-Operators „Wiederholung" in eine Schleife, so daß keine Berechnungen erforderlich sein dürfen, die im Falle einer Implementierung mit Rekursion beim Wiederaufstieg ausgeführt werden müßten. Eine formale Fassung dieser beiden Bedingungen definiert die Klasse der S(trong)LEAGen, und ein entsprechendes SLEAG-Verfahren kann zur Generierung einer verschränkt ablaufenden „Parser-Evaluator-Kombination" genutzt werden.

Andererseits soll es aber auch möglich sein, einen separaten Evaluator einzusetzen, wozu der Ableitungsbaum tatsächlich aufgebaut werden muß. Hier wird ausgenutzt, daß sich der Aufbau eines Ableitungsbaums gemäß der transformierten Grundgrammatik selbst durch eine (generierbare) SLEAG beschreiben läßt, so daß für diese Aufgabe nach der obigen Beschreibung ein „Ein-Pass-Compiler" generiert werden kann.

Als Beispiel eines separaten Evaluators soll das Single-Sweep-Verfahren implementiert werden. Die zugehörige Grammatikklasse ist dadurch charakterisiert, daß es für jede Hyper-Regel eine

Permutation der rechten Seite gibt, mit der die Regel linksdefinierend wird. Insbesondere ist dies für jede LEAG erfüllt. Die größere Mächtigkeit sowie die Tatsache, daß Affixpositionen weiter effizient als Prozedurparameter realisiert werden können, lassen einen Generator nach dem Single-Sweep-Verfahren als einfach zu erreichende und praktisch nützliche Alternative erscheinen.

### 6.1.1 Die Evaluation

Hier wird zuerst die direkte Evaluation einfacher Regeln (ohne EBNF-Operatoren) nach dem LEAG-Verfahren beschrieben.

In der Signatur eines Hyper-Nichtterminals werden Ein- („-“) und Ausgabeparameter („+“) unterschieden. Die Werte der Eingabeparameter der linken Regelseite sowie die Werte der Ausgabeparameter auf der rechten Regelseite werden im Regelkontext bestimmt. Die zugehörigen Affixpositionen werden definierend genannt. Eine echte Affixform auf definierender Position erfordert somit eine Analyse des angelieferten Wertes. Zudem müssen Vergleiche für mehrfach auf definierender Position vorkommende Variablen durchgeführt werden.

Umgekehrt werden die Werte der Ausgabeparameter der linken Regelseite sowie die Werte der Eingabeparameter auf der rechten Regelseite innerhalb der Regel bestimmt. Die zugehörigen Affixpositionen werden dann applizierend genannt, und eine echte Affixform auf applizierender Position erfordert nun eine Synthese des bereitzustellenden Wertes.

Eine Regel ist linksdefinierend, wenn es für jede Variable auf einer applizierenden Position eines Hyper-Nichtterminalvorkommens der rechten Seite links davon ein (anderes) Hyper-Nichtterminalvorkommen mit dieser Variablen auf definierender Position gibt.

Die grundlegende Idee für die Implementierung ist, Evaluatorcode in Prozeduren einzufügen, die den Ableitungsbaum entweder gedacht oder tatsächlich traversieren. Die Prozedurparameter stellen dabei die Affixpositionen der Hyper-Nichtterminale dar, deren Werte Sätze zu den Wertebereichssymbolen repräsentieren.

Am Anfang einer Prozedur für die Affixberechnung eines Grundnichtterminals wird die passende Alternative ermittelt. Zu Beginn einer Alternative werden die Eingabeparameter der linken Seite analysiert und evtl. Vergleiche durchgeführt. Jedes Vorkommen eines Grundnichtterminals auf der rechten Seite führt zu einem Prozeduraufruf, vor dem die zugehörigen Eingabeparameter synthetisiert werden und nach dem die zugehörigen Ausgabeparameter analysiert werden sowie evtl. Vergleiche durchgeführt werden. Am Ende der Alternative werden vor dem Verlassen der Prozedur die Ausgabeparameter der linken Seite entweder synthetisiert oder in besonderen Fällen durch Übertragung anderer Parameterwerte bestimmt. Dies spiegelt folgendes Prozedurschema wider:

```
PROCEDURE N (formale Parameter);
...
  (* Beginn einer Alternative *)
  Analyse der Eingabeparameter der linken Seite (evtl. Vergleiche)
  ...
  Synthese der Eingabeparameter zu N'
  N' (aktuelle Parameter);
  Analyse der Ausgabeparameter zu N' (evtl. Vergleiche)
  ...
  Synthese der Ausgabeparameter der linken Seite (evtl. Transfer)
  (* Ende der Alternative *)
...
END N;
```

Die formalen Parameter dieser Prozedur werden durch die Signatur des jeweiligen Hyper-Nichtterminals bestimmt. Die Eingabeparameter werden zu Wertparametern, die Ausgabeparameter zu Referenzparametern.

Als Parameterwerte werden Sätze zum Wertebereichssymbol durch deren Ableitungsbäume repräsentiert, damit die erforderliche Syntaxanalyse nicht die Laufzeit des generierten Compilers belasten muß. Dadurch lassen sich Analysen und Synthesen effizient durchführen, Vergleiche können einfach strukturell realisiert werden. Jedoch muß dazu die Repräsentation im allgemeinen eindeutig sein, was durch die Wohlgeformtheitsbedingungen sichergestellt wird.

Aus Gründen der Speichereffizienz und in Hinblick auf eine eigene Freispeicherverwaltung wird eine hochsprachliche Nachbildung des Speichers, genannt `Heap`, verwendet. Dort werden die Knoten von Ableitungsbäumen in aufeinanderfolgenden Einträgen abgelegt: `Heap[V]` ist ein Knotenbezeichner, `Heap[V+i]` verweist auf den i-ten Unterbaum (ein Beispiel ist in Abbildung 6.1 angegeben). Der Knotenbezeichner ist konzeptionell ein Paar aus Alternativennummer und Stelligkeit, d.h. der Anzahl der Unterbäume. Um Speicherplatz zu sparen, codieren wir diese Paare durch Zahlen. Die Konstante `arityConst` wird wie folgt für die Rekonstruktion der Komponenten verwendet :

$$\begin{aligned} \textit{Stelligkeit} &= \textit{Knotenbezeichner}\ \texttt{DIV arityConst}, \\ \textit{Alternativennummer} &= \textit{Knotenbezeichner}\ \texttt{MOD arityConst}. \end{aligned}$$

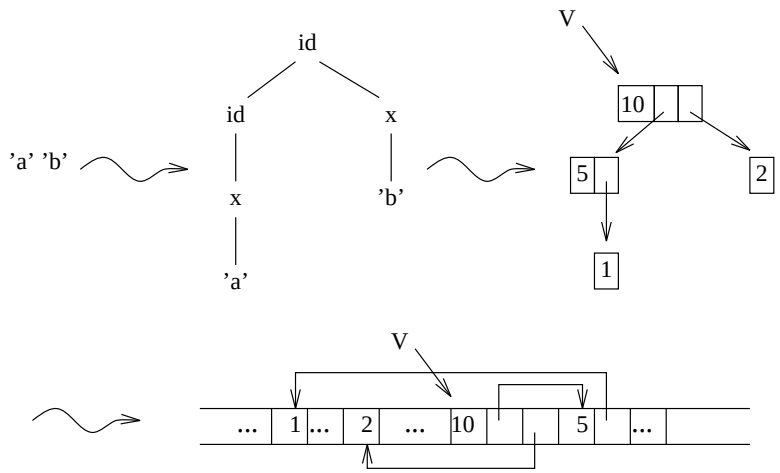



Abbildung 6.1: Speicherung der Repräsentation des Satzes 'a' 'b' zum Wertebereichssymbol id

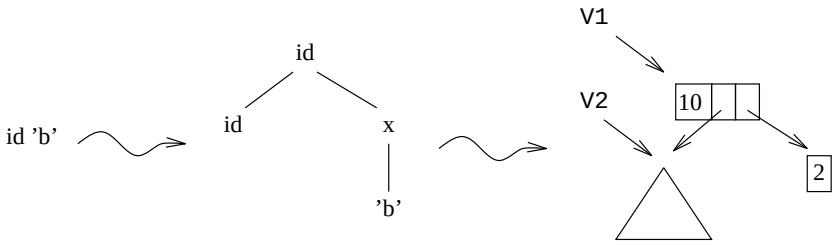
In Eta wird die Durchführung von Analysen und Synthesen weiter unterteilt. Formal werden dazu Analyse- und Syntheseprädikate für jede Meta-Alternative (sowie Equal- und Transferprädikate für jedes Meta-Nichtterminal) eingeführt. Diese primitiven Prädikate werden als kurze Prozeduren implementiert. Der Evaluatorcode besteht dann wesentlich aus Folgen von Prozeduraufrufen. Dies ist insofern verbesserungswürdig, als es zu Effizienzeinbußen aufgrund der Vielzahl von Prozeduraufrufen samt Parameterübergaben führt. Hier bietet sich die Möglichkeit, die Laufzeiteffizienz erheblich zu optimieren, indem wir sozusagen die Prozedurrümpfe direkt einsetzen und die Zuweisungen für Parameterübergaben durch Variablenumbenennungen weitestgehend eliminieren. Im allgemeinen wird dadurch jedoch der generierte Evaluator vergrößert.

#### 6.1.1.1 Synthesen

Bei einer Synthese wird aus vorhandenen Unterbäumen gemäß einer Affixform ein neuer Ableitungsbaum erstellt. In der Affixform vorkommende Variablen sind dabei Platzhalter für bereits

synthetisierte Unterbäume. Aus diesen wird durch Anlegen von Knoten schrittweise ein neuer Baum konstruiert. Dabei ist es nicht notwendig, die Unterbäume zu kopieren, da sie nicht verändert werden können.

Da zur Generierungszeit die Größe der anzulegenden Knoten und deren Anzahl bekannt sind, kann der benötigte Speicherplatz auf einmal für alle Knoten angefordert werden. Während einer preorder-Traversierung des Ableitungsbaums zur Affixform werden die erforderlichen Zuweisungen ermittelt, wobei die Indizierung der Speicherzellen relativ zu einem festen Offset erfolgen kann (in Abbildung 6.2 ist ein Beispiel ausgeführt).




```
IF Next >= LEN(Heap^) - 4 THEN EvalExpand END;
V1 := Next; Heap[Next] := 10; Heap[Next + 1] := V2;
Heap[Next + 2] := Next + 3; Heap[Next + 3] := 2;
INC(Next, 4);
```

Abbildung 6.2: Synthese von id 'b' zum Wertebereichssymbol id

#### 6.1.1.2 Transfer

Dies ist ein technisches Problem, das sich dadurch ergibt, daß in der Prozedur eines Nichtterminals für die formalen Parameter unterschiedliche Bezeichner erforderlich sind. Tritt wiederholt eine Variable als triviale Affixform auf den Positionen der linken Seite auf, so muß für applizierende Positionen durch einfache Zuweisungen derselbe Wert geliefert werden.

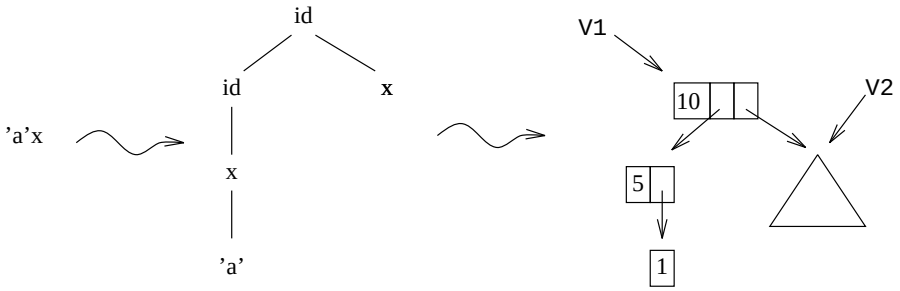
#### 6.1.1.3 Analysen

Bei einer Analyse wird ein Ableitungsbaum auf die durch die Affixform vorgegebene Struktur überprüft. Im Erfolgsfall werden die Unterbäume zu den in der Affixform vorkommenden Variablen ermittelt.

Die erforderliche Überprüfung der Knotenbezeichner wird während einer preorder-Traversierung des Ableitungsbaums zur Affixform durchgeführt. Die zugehörige Traversierung des zu analysierenden Baums muß zur Laufzeit des generierten Compilers erfolgen. Dabei kann der Zugriff auf die Unterbäume durch geschachtelte Indizierungen codiert werden, die den jeweiligen Weg von der Wurzel aus darstellen.

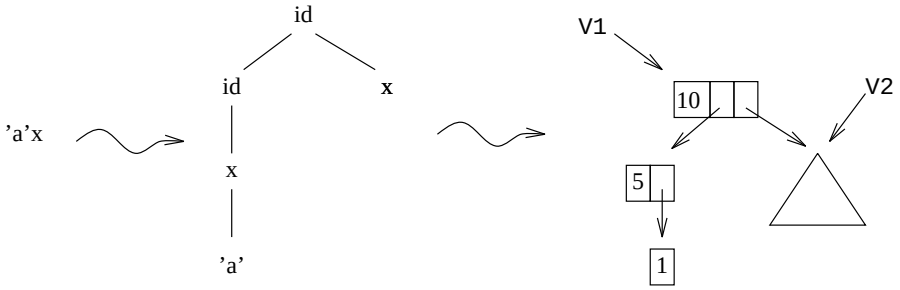
Wie aus Abbildung 6.3 ersichtlich ist, können diese Ausdrücke schnell anwachsen.[1] Dies kann durch die übliche Verwendung temporärer Variablen zur Elimination gemeinsamer Teilausdrücke verhindert werden (siehe Abbildung 6.4), wodurch zusätzlich die Laufzeit optimiert wird.

[1] Insbesondere versagte der von uns verwendete Oberon-Compiler ab Schachtelungstiefe 5




```
IF Heap[V1] # 10 THEN Error END;
IF Heap[Heap[V1 + 1]] # 5 THEN Error END;
IF Heap[Heap[Heap[V1 + 1] + 1]] # 1 THEN Error END;
V2 := Heap[V1 + 2];
```

Abbildung 6.3: Analyse von 'a' x zum Wertebereichssymbol id




```
IF Heap[V1] # 10 THEN Error END; V3 := Heap[V1 + 1];
IF Heap[V3] # 5 THEN Error END;
IF Heap[Heap[V3 + 1]] # 1 THEN Error END;
V2 := Heap[V1 + 2];
```

Abbildung 6.4: Analyse von 'a' x zum Wertebereichssymbol id mit Hilfe temporärer Variablen

#### 6.1.1.4 Vergleiche

Steht eine Variable *M* mehrfach auf definierender Position, so müssen die zugehörigen Werte aufgrund der Forderung nach konsistenter Ersetzung gleich sein. Dagegen müssen die Werte der Variablen *M* und *#M* verschieden sein.

Die Überprüfung wird einfach durch den Vergleich der Ableitungsbäume realisiert. Die boolesche Funktionsprozedur `Equal` erfüllt diese Aufgabe durch eine rekursive Traversierung beider Ableitungsbäume mit erfolgreichem Abbruch bei gleichen Verweisen. Um zu überprüfen, ob zwei Bäume verschieden sind, wird das Resultat negiert. Ein Aufruf dieser Prozedur erfolgt während der Analyse, direkt nach der Bestimmung des zugehörigen Unterbaums.

#### 6.1.1.5 Prädikate

Einige Hyper-Nichtterminale können nur zum leeren Wort abgeleitet werden. Sie tragen nichts zur kontextfreien Definition der Quellsprache bei. Diese sogenannten Prädikate müssen aufgrund der für sie absichtlich zugelassenen Mehrdeutigkeit besonders behandelt werden.

Bei der Evaluation eines Prädikats muß anhand der Parametrisierung eine Ableitung des leeren Worts im allgemeinen durch Backtracking gefunden werden. Das Scheitern einer Analyse bzw. eines Vergleichs zeigt hier keinen Fehler sondern eine Sackgasse an.

Ein Prädikat wird als boolesche Funktionsprozedur implementiert. Im Gegensatz zum Prozedurschema für Grundnichtterminale ist hier die Auswahl der passenden Alternative wesentlich. Dazu werden alle Alternativen der Reihe nach ausprobiert:

```
PROCEDURE P (formale Parameter): BOOLEAN;
...
BEGIN
  failed := TRUE;
  Code zu Überprüfung der ersten Alternative
  IF failed THEN
    Code zu Überprüfung der zweiten Alternative
    ...
  END;
  RETURN ~ failed
END P;
```

Für eine Alternative steuern die Analysen und Vergleiche sowie die Prozeduraufrufe zu den zugehörigen Prädikaten den weiteren Kontrollfluß. Dies wird durch geschachtelte `IF`-Anweisungen angemessen formuliert. Dabei müssen die im folgenden Schema nicht ausformulierten Analysen, insbesondere die Überprüfung des einzelnen Knotenbezeichners, ebenfalls in geschachtelte `IF`-Anweisungen umgesetzt werden.

```
(* Beginn einer Alternative *)
IF Analyse der Eingabeparameter der linken Seite (evtl. Vergleiche)
   erfolgreich THEN
  ...
  Synthese der Eingabeparameter zu P'
  IF P'(aktuelle Parameter) THEN
    IF Analyse der Ausgabeparameter zu P' (evtl. Vergleiche)
       erfolgreich THEN
      ...
      Synthese der Ausgabeparameter der linken Seite (evtl. Transfer)
      failed := FALSE;
      ...
    END
  END;
  ...
END;
(* Ende der Alternative *)
```

Erst der Aufruf einer Prädikatprozedur aus einer Prozedur für ein Grundnichtterminal führt beim Scheitern zu einer Fehlermeldung.

```
Synthese der Eingabeparameter zu P
IF ~ P (aktuelle Parameter) THEN Fehlerbehandlung END;
Analyse der Ausgabeparameter zu P (evtl. Vergleiche)
```

#### 6.1.1.6 Fehlerbehandlung

Wie oben angesprochen, können während der Evaluation Fehler auftreten: Analysen, Vergleiche und Prädikate können scheitern. Dies zeigt Kontextfehler an und verhindert eine korrekte Übersetzung. Ein Abbruch des generierten Compilers stellt jedoch ein unbefriedigendes Verhalten dar. Der Programmablauf soll fortgeführt werden, um weitere Kontextfehler finden und melden zu können. Dabei sollen Folgefehlermeldungen unterdrückt werden.

Anders als bei scheiternden Vergleichen kann bei scheiternden Analysen der Programmablauf nicht ohne weiteres fortgeführt werden, da im allgemeinen bei der Überprüfung der Unterbäume temporäre Variablen Werte erhalten. Auch bei scheiternden Prädikaten müssen die Ausgabeparameter auf definierte Werte gesetzt werden.

Als Lösung bietet es sich an, einen speziellen Fehlerknoten anzulegen, der die maximale Stelligkeit aufweist. Für jeden Unterbaum wird durch eine Schlinge wieder auf den Fehlerknoten verwiesen. Als Fehlerwert wird ein Verweis auf diesen Knoten verwendet; Folgefehler können dann anhand dieses Wertes erkannt werden. In der Implementierung bilden die ersten Einträge des Heaps den Fehlerknoten. Der Fehlerwert wird durch die Konstante `errVal` bezeichnet und verweist auf den ersten Eintrag dieses Knotens.

Scheitert in einer Analyse die Überprüfung eines Knotenbezeichners, wird der Verweis auf den Knoten durch den Fehlerwert ersetzt. Wird dann die Analyse einfach fortgeführt, so ergibt sich für alle Unterbäume ebenfalls der Fehlerwert; insbesondere werden die zugehörigen Variablen auf diesen Wert gesetzt. Dabei wird kein Folgefehler gemeldet, wenn bereits der Fehlerwert untersucht wird:

```
IF V # errVal THEN Fehlermeldung; V := errVal END;
```

Scheitert ein Prädikat, dann werden die Ausgabeparameter auf den Fehlerwert gesetzt. Die Unterdrückung von Fehlermeldungen, falls eines der Eingabeparameter den Fehlerwert aufweist, erfolgt für Prädikataufrufe aus Prozeduren der Grundnichtterminale.

```
IF alle Eingabeparameter ungleich errVal THEN Fehlermeldung END;
```

Scheitert der Vergleich zweier Bäume, wird die Fehlermeldung unterdrückt, falls für mindestens einen der Fehlerwert vorliegt:

```
IF (V1 # errVal) & (V2 # errVal) THEN Fehlermeldung END;
```

Um die Größe des generierten Codes zu verringern, werden die Fehlerbehandlungen in den Prozeduren `AnalyseError`, `Eq`, `UnEq` sowie `CheckP` für ein Prädikat $P$ zusammengefaßt.

Die Texte der Fehlermeldungen werden vom Generator automatisch aus den in der Spezifikation vorkommenden Bezeichnern erstellt. Zusammen mit der Fehlermeldung wird eine Position im Eingabetext ausgegeben. Analysefehlermeldungen zum Hyper-Nichtterminal $N$ haben die Form „analysis in $N$ failed“. Für scheiternde Vergleiche wird der Name der Variablen $M$ mit dem des zugehörigen Hyper-Nichtterminals in „$M$ failed in $N$“ verbunden. Scheitert ein Prädikat $P$, wird die Fehlermeldung „predicate $P$ failed“ ausgegeben.

#### 6.1.1.7 EBNF-Operatoren

Die Semantik der ENBF-Operatoren ist über die in Abschnitt 2.2 beschriebene Transformation definiert. Für die daraus resultierenden Regeln ohne EBNF-Operatoren läßt sich das bisher beschriebene LEAG-Verfahren anwenden, falls diese linksdefinierend sind.

Eine optimierte Implementierung der EBNF-Operatoren in Prädikatregeln, insbesondere eine Umsetzung der Wiederholung in eine Iteration, wurde verworfen. Da die Iteration ein Backtracking verhindert, würde sich der Verzicht auf Rekursion in einer veränderten Semantik bemerkbar machen.

Aufgrund der Einbettung des Codes für anonyme Grundnichtterminale ergeben sich keine Änderungen bei den Evaluationscodeschemata. Bei der Generierung ist allerdings darauf zu achten, daß sich die Namen der für die Evaluation benötigten temporären Variablen nicht mit denen der umgebenden Prozedur überschneiden. Die durch die Einbettung des Codes entfallenden formalen Prozedurparameter werden durch temporäre Variablen dargestellt.

Einschränkungen ergeben sich durch die optimierende Behandlung des EBNF-Operators Wiederholung durch den Parsergenerator, der diesen Operator nicht, wie im nachfolgenden Beispiel beschrieben, als rekursive Prozedur, sondern durch eine Schleife implementiert.

```
(* Wiederholung *)
{<Parameter für den Rumpf> ...  <Rekursionsparameter>
}<Parameter für den Abbruch>

(* transformierte Wiederholung *)
A <Parameter für den Rumpf>: ...  A <Rekursionsparameter>.
A <Parameter für den Abbruch>: .

(* rekursive Implementierung *)
PROCEDURE A (formale Parameter);
...
  (* Begin des Rumpfes *)
  Analyse der Eingabeparameter für den Rumpf (evtl. Vergleiche)
  ...
  Synthese der Rekursionsparameter
  A (aktuelle Parameter)
  Analyse der Rekursionsparameter (evtl.  Vergleiche)
  Synthese der Ausgabeparameter für den Rumpf (evtl. Transfer)
  (* Ende des Rumpfes *)
  ...
  (* Abbruch *)
  Analyse der Eingabeparameter für den Abbruch (evtl. Vergleiche)
  Synthese der Ausgabeparameter für den Abbruch (evtl. Transfer)
  ...
END A;
```

Da die Rekursion durch eine Schleife implementiert wird, entfällt die Möglichkeit der Berechnung beim Wiederaufstieg. Die Werte der Rekursionsparameter sind daher in einem Schleifendurchlauf noch nicht bekannt und können erst im jeweils nachfolgenden Schritt bestimmt werden. Diese Parameter stehen somit nicht für Analysen, Vergleiche und Synthesen, die gemäß dem LEAG-Verfahren durchgeführt werden, zur Verfügung.

Durch Wohlgeformtheitsbedingungen werden Spezifikationen daher so eingeschränkt, daß weder Analysen für Rekursionsparameter noch Vergleiche erforderlich sind und eine effiziente Implementierbarkeit der Synthesen der Ausgabeparameter garantiert wird.

Eine EAG ist eine S(trong-)LEAG, wenn

- ihre transformierten Regeln linksdefinierend sind (LEAG),

- auf einer definierenden Position der Rekursionsparameter nur eine Variable des Wertebereichssymbols steht, so daß keine Analysen erforderlich sind,
- für jede in Rekursionsparametern auf definierender Position vorkommende Variable *M* weder *M* noch *#M* auf einer weiteren definierenden Position in der Alternative vorkommt, damit keine Vergleiche erforderlich sind,
- keine in Rekursionsparametern auf definierender Position vorkommende Variable mehrfach appliziert wird (einfache Synthese).

In der iterativen Implementierung stellen die Rekursionsparameter einer Wiederholung der Parameter der linken Seite im nachfolgenden Iterationsschritt dar. Die Iteration endet mit der Evaluation der Ausgabeparameter für den leeren Fall.

In einem Iterationsschritt sind Verweise auf die Bäume der Rekursionsparameter noch nicht bekannt, da sie erst im nachfolgenden Iterationsschritt bestimmt werden, und können nicht eingetragen werden. Behoben wird dieses Problem dadurch, daß sich diese Einträge des Baums gemerkt und im nachfolgenden Iterationsschritt die Verweise auf die Unterbäume nachgetragen werden. Um sich nicht in jedem Iterationsschritt mehrere Einträge merken zu müssen, wird gefordert, daß die Variablen der definierenden Rekursionsparameter nicht mehrfach appliziert werden.

In der Implementierung erfolgt der Aufbau eines Ableitungsbaumes durch „indirekte Heapzugriffe“. Eine temporäre Variable, die sogenannte „Wurzelvariable“, markiert eine Stelle im Heap, die auf die Wurzel des synthetisierten Baums verweist. Eine zweite sogenannte „Laufvariable“ beschreibt den Heapeintrag des bereits synthetisierten Baums, in den im folgenden Iterationsschritt ein weiterer Teil des Baums synthetisiert wird. Vor der Iteration erfolgt die Initialisierung beider Variablen. Im Anschluß an die Iteration stellt die Wurzelvariable den Ausgabeparameter dar.

Folgendes Beispiel zeigt den für eine Wiederholung generierten Oberon-Quellcode:

```
N = 'i' N | .
{<+ 'i' N: N> 'a' <N>}<+ : N>

V1 := NextHeap; INC(NextHeap); V2 := V1; (* N *)
WHILE Tok = 'a' DO Get(Tok);
  Heap[V2] := NextHeap; Heap[NextHeap] := „N = 'i' N.“;
  V2 := NextHeap + 1; INC(NextHeap, 2)
END;
Heap[V2] := NextHeap; Heap[NextHeap] := „N = .“; INC(NextHeap);
V1 := Heap[V1];
```

Wie in dem Beispiel zu erkennen ist, wird erst mit „`Heap[V2] := NextHeap;`“ der Verweis auf den Unterbaum eintragen und sich mit „`V2 := NextHeap + 1;`“ in der nachfolgenden Synthese der noch undefinierte Eintrag im Baum gemerkt. Dabei wird die Laufvariable `V2` mit einem neuen Wert überschrieben.

Das Nachtragen des Verweises erfolgt getrennt von Synthesen für alle Laufvariablen, damit der Wert einer Laufvariablen nicht vorzeitig in einer Synthese überschrieben wird, wie das andernfalls in folgendem Beispiel der Fall wäre.

```
N = 'i' N | .
{<+ 'i' N2: N, + 'i' N1: N> 'a' <N1, N2>}<+ : N, + : N>

V1 := NextHeap; INC(NextHeap); V2 := V1; (* N1 *)
V3 := NextHeap; INC(NextHeap); V4 := V3; (* N2 *)
```

```
WHILE Tok = 'a' DO Get(Tok);
  Heap[V2] := NextHeap; Heap[V4] := NextHeap + 2;
  Heap[NextHeap] := „N = 'i' N."; V4 := NextHeap + 1;
  Heap[NextHeap + 3] := „N = 'i' N."; V2 := NextHeap + 3;
  INC(NextHeap, 4)
END;
...
```

Weiterhin kann es vorkommen, daß die Variable auf definierender Position eines Rekursionsparameters in einer Alternative nicht appliziert wird, in anderen dagegen schon. Zur Laufzeit besitzt dann nach Auswertung dieser Alternative die entsprechende Laufvariable keinen definierten Wert; in einem nachfolgenden Iterationsschritt darf in diesem Fall kein Verweis auf auf einen Unterbaum eingetragen werden.

Als Lösung wird in dem Code einer Alternativen, in der diese Variable nicht appliziert wird, der Laufvariablen der Wert `undef` zugewiesen. Vor dem Einhängen eines Unterbaums muß daher in allen Alternativen die Laufvariable auf diesen Wert hin überprüft werden. Da dies sehr aufwendig ist, entfällt diese Behandlung, wenn in allen Alternativen die entsprechende Variable appliziert wird, was sich durch den Generator überprüfen läßt. Das Codeschema für diesen Fall wird durch das nachfolgende Beispiel angedeutet. Zu beachten ist hier, daß das Einhängen wie oben beschrieben unabhängig von der Synthese stattfindet.

```
N = 'i' N | .
{<+ 'i' N: N> 'a' <N> | <+ : N> 'b' <N>}<+ : N>

V1 := NextHeap; INC(NextHeap); V2 := V1; (* N *)
WHILE ...  DO Get(Tok);
  IF Tok = 'a' THEN
    IF V2 # undef THEN
      Heap[V2] := NextHeap
    END;
    IF V2 # undef THEN
      Heap[NextHeap] := „N = 'i' N."; V2 := NextHeap + 1;
      INC(NextHeap, 2)
    END
  ELSIF Tok = 'b' THEN
    IF V2 # undef THEN
      Heap[V2] := NextHeap
    END;
    IF V2 # undef THEN
      Heap[NextHeap] := „N = ."; INC(NextHeap)
    END
  END
END;
...
```

#### 6.1.1.8 Optimierung terminaler Affixformen

Terminale Affixformen sind konstant in dem Sinne, daß sie im Evaluator durch immer gleiche Bäume dargestellt werden. Eine Synthese muß prinzipiell nur einmal erfolgen; später genügt es, einen Verweis auf den Baum zu liefern. Ebenso muß für jede abschließende Regel (ohne Nichtterminale) ein Blatt nur einmal im Heap vorliegen. Durch eine solche Konstantenfaltung kann der Speicherbedarf gesenkt werden, und natürlich entsteht ein kürzeres und schnelleres Programm.

Die Generierung einer derart optimierten Evaluation erfolgt optional, da sie aufgrund der dadurch entstehenden Kollabierung nicht zur Erstellung von Ableitungsbäumen geeignet ist.

In der Implementierung werden initial alle Blätter und die Bäume terminaler Affixformen in einem reservierten Teil des Heaps vorsynthetisiert. Die Verweise sind dadurch bekannt und können in Synthesen benutzt werden. Das Ende dieses reservierten Bereichs wird, wie in Abbildung 6.5 dargestellt, von der Konstanten `predefined` markiert. Das Beispiel einer Synthese aus Abbildung 6.2

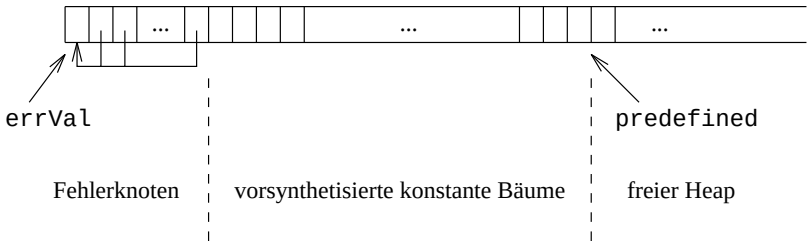



Abbildung 6.5: Erläuterung der Speicherstruktur in den generierten Evaluatoren

verändert sich dadurch zu:

```
(* Synthese von id 'b' zum Wertebereichssymbol id *)
IF Next >= LEN(Heap^) - 3 THEN EvalExpand END;
V1 := Next; Heap[Next] := 10; Heap[Next + 1] := V2;
Heap[Next + 2] := 42; (* Verweis auf Blatt *)
INC(Next, 3);
```

In Analysen können terminale Affixformen im allgemeinen nicht genutzt werden, da die zu analysierenden Bäume auf vielfältige Art und Weise entstanden sein können. Da Blätter jedoch genau einmal im Heap vorliegen, kann die Überprüfung des Knotenbezeichners durch eine Überprüfung des Verweises abgekürzt werden. Das Beispiel aus Abbildung 6.4 verändert sich dadurch zu:

```
(* Analyse des Satzes 'a' x zum Wertebereichssymbol id *)
IF Heap[V1] # 10 THEN Fehlerbehandlung END; V3 := Heap[V1 + 1];
IF Heap[V3] # 5 THEN Fehlerbehandlung END;
IF Heap[V3 + 1] # 43 (* Verweis auf Blatt *) THEN Fehlerbehandlung END;
V2 := Heap[V1 + 2];
```

#### 6.1.1.9 Freispeicherverwaltung

Da Speicherplatz für Knoten zwar bei Synthesen explizit angefordert, aber nie explizit freigegeben wird, und da potentiell ein großer Teil des Heaps zur Darstellung von Zwischenergebnissen mit temporärer Bedeutung verwendet wird, soll der Evaluator wahlweise um eine Freispeicherverwaltung ergänzt werden können.

Im generierten Evaluator kann ein Referenzzähler-Verfahren verwendet werden. Hierbei werden die in temporären Variablen vorliegenden Verweise auf die Wurzeln von Bäumen zur Laufzeit mitprotokolliert, was eine sofortige Wiederbenutzung nicht mehr referenzierter Knoten ermöglicht.

Die Referenz auf einen Baum wird durch die Erhöhung des Zählers des Wurzelknotens ausgedrückt. Durch Kollabierung entstehen, wie in Abbildung 6.6 erläutert, mehrfache Referenzen auf Teilbäume.

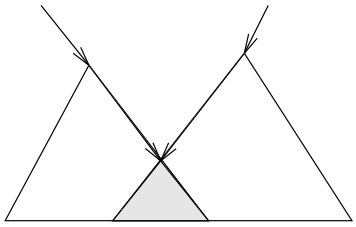


Abbildung 6.6: Erläuterung einer mehrfachen Referenz auf Teilbäume

Verliert die letzte Referenz ihre Gültigkeit, erfolgt eine Freigabe des Wurzelknotens sowie aller Knoten seiner Unterbäume, auf die keine weitere Referenz existiert, indem sie in Freispeicherlisten eingetragen werden. Angeforderter Heapspeicher wird dann vorrangig diesen Listen entnommen.

Neue Referenzen entstehen bei der Auswertung von applizierenden Affixpositionen, und zwar auf die durch Synthesen erstellten Bäume und auf die durch Variablen in der Affixform beschriebenen Unterbäume.

Erst mit dem Abschluß der Evaluation einer Alternativen verliert die Referenz auf einen Baum, der den Parameterwert einer definierenden Position darstellt, ihre Gültigkeit, da Unterbäume eventuell noch für Synthesen oder Vergleiche benutzt werden.

Für den Fehlerknoten und für vorsynthetisierte Bäume terminaler Affixformen funktioniert das Verfahren entsprechend. Beim Setzen des Fehlerwertes verliert der fehlerhafte (Unter-)Baum seine Referenz, der Fehlerknoten erhält dafür eine weitere, um eine Freigabe ohne Sonderbehandlung zu verhindern. Durch eine Synthese entsteht eine zusätzliche Referenz auf einen vorsynthetisierten Baum.

Für die Evaluation von Grundnichtterminalen ergibt sich dadurch eine Optimierungsmöglichkeit, daß, im Gegensatz zu Prädikaten, sicher eine Bestimmung aller Parameterwerte einer Alternative vorgenommen wird. In diesem Fall kann vom Generator bestimmt werden, wie oft eine Variable appliziert wird; die Erhöhung des Referenzzählers des entsprechenden Unterbaums wird dann zusammengefaßt. Dadurch kann eine vorzeitige Löschung der Referenz direkt nach Analysen erfolgen. Kommt allerdings eine Variable *M* oder *#M* in einer Alternative auf einer definierenden Position „rechts“ von allen Positionen vor, auf denen sie appliziert wird, kann aufgrund einer verfrühten Freigabe des Baums ein fehlerhafter Vergleich erfolgen. Dies wird durch eine zusätzliche Erhöhung des Referenzzählers verhindert. Im Anschluß an den letzten Vergleich ist in diesem Fall ein explizites Löschen der Referenz erforderlich.

In der Implementierung wird der Knotenbezeichner konzeptionell um den Referenzzähler zu einem Tripel erweitert. Die zusätzliche Konstante `refConst` wird wie folgt für die Rekonstruktion der Komponenten verwendet:

```
    Referenzzähler  =  Knotenbezeichner DIV refConst,
       Stelligkeit  =  (Knotenbezeichner MOD refConst) DIV arityConst,
Alternativennummer  =  Knotenbezeichner MOD arityConst.
```

In dem Referenzzähler wird lediglich die Anzahl der zusätzlichen Verweise auf einen Knoten gezählt.

Für jede Knotenstelligkeit wird eine eigene Freispeicherliste angelegt. Freigegebene Knoten werden gemäß ihrer Stelligkeit in diesen Listen verwaltet. Die Einträge des Feldes `FreeList` verweisen auf den jeweils ersten Knoten einer solchen Liste. Die Verkettung ist, wie in Abbildung 6.7 dargestellt, in dem jeweils ersten Eintrag der freigegebenen Knoten realisiert.

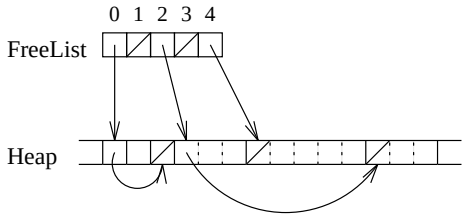



Abbildung 6.7: Beispiel einer Freispeicherliste

In Synthesen erfolgt nun die Anforderung von Speicherplatz knotenweise durch die Prozedur `GetHeap`. Das Löschen einer Referenz sowie eine eventuelle Freigabe des Speicherplatzes erfolgt durch die Prozedur `FreeHeap`.

Durch Anwendung des Referenzzählerverfahrens und unter der Voraussetzung, daß keine Optimierung terminaler Affixformen erfolgt und die Synthese in einer Prädikatprozedur vorkommt, verändert sich das Beispiel der Synthese sich aus Abbildung 6.2 zu:

```
(* Synthese von id 'b' zum Wertebereichssymbol id *)
GetHeap(V1, 2); Heap[V1] := 10; Heap[V1 + 1] := V2;
INC(Heap[V2], refConst);
GetHeap(V3, 1); Heap[V1 + 2] := V3; Heap[V3] := 2;
```

Das Beispiel aus Abbildung 6.4 verändert sich unter der Voraussetzung, daß keine Optimierung terminaler Affixformen erfolgt und die Analyse in der Prozedur für ein Grundnichtterminal vorkommt, zu:

```
(* Analyse des Satzes 'a' x zum Wertebereichssymbol id *)
IF Heap[V1] # 10 THEN Fehlerbehandlung END; V3 := Heap[V1 + 1];
IF Heap[V3] # 5 THEN Fehlerbehandlung END;
IF Heap[Heap[V3] + 1] # 1 THEN Fehlerbehandlung END;
V2 := Heap[V1 + 2]; INC(Heap[V2], 3 * refconst);
FreeHeap(V1);
```

#### 6.1.1.10 Speicherung von Positionsangaben

Für Fehlermeldungen in einem separaten Evaluator wird der von dem „Ein-Pass-Compiler“ aufgebaute Ableitungsbaum mit Positionsangaben angereichert (siehe Kapitel 7). Dazu wird im Ableitungsbaum für jede mögliche Fehlerstelle eine Position gespeichert. Zum Aufbau werden Anweisungen in den Evaluationscode des Ein-Pass-Compilers eingefügt.

In der Implementierung werden die Positionen parallel zum Heap in dem Feld `PosHeap` dargestellt. Für den korrekten Eintrag der Positionsangaben werden diese in dem Keller `PosStack` zwischengespeichert. Durch Aufruf der Prozedur `PushPos` wird nach der Evaluation einer definierenden Position eine Position auf dem Keller abgelegt; vor der Synthese des Ausgabeparameters einer Prozedur werden diese Einträge durch Aufruf der Prozedur `PopPos` vom Keller in das Feld `PosHeap` übertragen.

### 6.1.2 Der Generator

Der SLEAG-Evaluatorgenerator ermöglicht die Generierung einer Evaluation entweder nach dem LEAG-Verfahren für transformierte EBNF-Regeln oder nach dem SLEAG-Verfahren. Er gliedert

sich in einen Testteil, der Prozeduren für die Überprüfung der Linksdefiniertheit und der SLEAG-Bedingungen für Wiederholungen enthält, und eine Sammlung von Prozeduren, die gemäß dem Prozedurschema Programmfragmente für globale Deklarationen, Prädikate, Deklarationen von formalen Prozedurparametern und lokalen Variablen, Prädikataufrufe sowie Analyse- und Synthesecode generieren. Diese werden anderweitig benutzt, um Evaluatorcode in umgebenden Code einzustreuen.

Sowohl der Test als auch die Generierung orientieren sich an dem Prozedurschema und erfolgen daher nichtterminalweise. Dabei ist zu beachten, daß bei dem LEAG-Verfahren auch für anonyme Nichtterminale eigene Prozeduren erstellt werden. Die Prädikatcodierung erfolgt ausschließlich nach dem LEAG-Verfahren.

Im generierten Code kann wahlweise eine Optimierung terminaler Affixformen erfolgen. Ebenso kann dieser optional um eine Freispeicherverwaltung ergänzt werden. Aufgrund der vielen Fallunterscheidungen ergibt sich die besondere Größe des Moduls `SLEAGGen.Mod`.

#### 6.1.2.1 Überprüfung der Evaluierbarkeit

Die Überprüfung der Auswertbarkeit erfolgt unabhängig von der eigentlichen Generierung.

```
PROCEDURE Test*;
PROCEDURE InitTest*;
PROCEDURE IsLEAG* (N: INTEGER; EmitErr: BOOLEAN): BOOLEAN;
PROCEDURE IsSLEAG* (N: INTEGER; EmitErr: BOOLEAN): BOOLEAN;
PROCEDURE FinitTest*;
```

Nach einer Initialisierung wird die regellokale Bedingung der Linksdefiniertheit für ein Nichtterminal durch Aufruf von `IsLEAG` überprüft. `IsSLEAG` überprüft die SLEAG-Bedingung für ein Nichtterminal und die durch die Transformation seiner Regeln entstandenen anonymen Nichtterminale. Über den Schalter `EmitErr` wird die Ausgabe von Fehlermeldungen gesteuert.

Abkürzend kann mit dem Kommando `Test` die SLEAG-Bedingung für die gesamte Grammatik überprüft werden.

Die Überprüfung der Linksdefiniertheit erfolgt durch eine die Evaluation vorwegnehmende Traversierung der transformierten Regeln. Zu Beginn werden dazu die Variablen einer Regel als undefiniert markiert. Kommt eine Variable im Laufe der Traversierung auf definierender Position vor, wird sie als definiert markiert. Eine Regel ist linksdefinierend, wenn keine undefinierte Variable auf applizierender Position vorkommt.

#### 6.1.2.2 Schnittstelle

Die Generierung eines Auswertungsverfahrens teilt sich auf in die Generierung globaler Teile und der Generierung prozedurlokaler Teile.

```
CONST
  parsePass = 0; onePass = 1; sSweepPass = 2;

PROCEDURE InitGen (MOut: eIO.TextOut; Treatment: INTEGER);
PROCEDURE GenDeclarations;
PROCEDURE GenPredProcs;
PROCEDURE FinitGen;
```

Mit der Initialisierung wird das generierte Auswertungsverfahren über den Parameter `Treatment`, der die Werte `parsePass`, `onePass` und `sSweepPass` annehmen kann, verbindlich festgelegt. Zusätzlich wird der Text übergeben, in den wechselseitig generiert wird. Es erfolgt die Berechnung der Knotenbezeichner und die Bestimmung terminaler Affixformen. Die für letztere synthetisierten Bäume bilden mit dem Fehlerknoten zusammen die initialen Einträge des Heaps.

Die Prozedur `GenGlobalDef` fügt globale Datenstrukturen und Prozeduren in das zu generierende Modul ein und erstellt die Evaluatortabelle. Die globalen Datenstrukturen wie beispielsweise die des Heaps und die globalen Prozeduren wie beispielsweise die Fehlerausgabeprozeduren werden aus einer Textdatei in das zu generierende Modul kopiert. Dabei werden Markierungen „$“ genutzt, um spezifische Teile einzufügen bzw. um nicht benötigte Teile der Textdatei zu überlesen. Die Evaluatortabelle beinhaltet die vorinitialisierten Heapeinträge und wird als permanante Datei angelegt.

Die Prozedur `GenPredProcs` codiert die Prädikate. Für jedes Prädikat werden zwei Prozeduren erzeugt: Eine wertet das Prädikat aus, in der zweiten erfolgt die in Abschnitt 6.1.1.6 beschriebene Fehlerbehandlung. Da sich Prädikate verschränkt aufrufen können, werden FORWARD-Deklarationen erzeugt.

```
PROCEDURE ComputeVarNames (N: INTEGER; Embed: BOOLEAN);
PROCEDURE InitScope (Scope: eEAG.ScopeDesc);
PROCEDURE GenFormalParams (N: INTEGER; ParNeeded: BOOLEAN);
PROCEDURE GenVarDecl (N: INTEGER);
PROCEDURE GenAnalPred (N, P: INTEGER);
PROCEDURE GenSynPred (N, P: INTEGER);
PROCEDURE GenPredCall (N, ActualParams: INTEGER);
PROCEDURE GenActualParams (P: INTEGER; ParNeeded: BOOLEAN);
PROCEDURE GenRepStart (N: INTEGER);
PROCEDURE GenRepAlt (N: INTEGER; A: eEAG.Alt);
PROCEDURE GenRepEnd (N: INTEGER);
PROCEDURE PosNeeded (P: INTEGER): BOOLEAN;
```

Um für die Deklaration temporärer Variablen deren Anzahl und Namen zu kennen, erfolgt die Variablenvergabe durch die Prozedur `ComputeVarNames` vor der Generierung prozedurlokaler Programmfragmente. Vor der Generierung werden die Variablen einer Alternative durch Aufruf von `InitScope` als undefiniert markiert. `GenAnalPred` faßt die Generierung von Analysen und Vergleichen zusammen, `GenSynPred` die von Synthesen und Transfers. Die Generierung von Evaluationscode für Schleifen erfordert die in Abschnitt 6.1.1.7 beschriebene Sonderbehandlung und besteht aus einer Initialisierung, aus der Behandlung der Affixpositionen der Rekusionsparameter sowie der applizierenden Positionen der linken Seite und abschließend aus der Evaluation für den Schleifenabbruch.

Da in der Sprache Oberon die formalen Parameter einer Prozedur und ihrer FORWARD-Deklaration identisch sein müssen, werden deren Namen nach einem festen Schema vergeben, so daß für deren Generierung keine vollständige Variablenvergabe notwendig ist.

#### 6.1.2.3 Vergabe von Variablennamen

Affixpositionen und Variablen werden in den Prozeduren des generierten Programms durch temporäre Variablen dargestellt. Die Affixpositionen der linken Regelseite eines Nichtterminals werden dabei durch jeweils gleiche Oberon-Variablen repräsentiert, i.a. durch die formalen Prozedurparameter. Kommt eine Variable als triviale Affixform auf applizierender Position vor, so werden die Variable und die zugehörige Position durch dieselbe temporäre Variable dargestellt. Dies gelingt nicht in dem Fall, daß eine Variable mehrfach als triviale Affixform auf applizierenden Positionen

einer linken Seite steht. In diesem Fall ist ein Transfer nötig. Kommt eine Variable hingegen als triviale Affiform auf definierender Position vor, darf diese Optimierung nur für die Definitionsposition der Variablen erfolgen, da ansonsten der Wert der temporären Variable überschrieben wird.

Die Evaluation von Wiederholungen erfordert zusätzlich zu den temporären Variablen für die applizierenden Positionen der formalen Parameter jeweils eine Laufvariable (siehe Abschnitt 6.1.1.7).

Weiterhin werden für die Traversierung in Analysen (und bei Evaluation mit Freispeicherverwaltung in Synthesen) temporäre Variablen für jeden traversierten Knoten des Baums benötigt.

In dem Fall, daß der Code für anonyme Nichtterminale in Prozeduren eingebettet wird, erfolgt eine Parameterübergabe dadurch, daß die temporären Variablen für die Affixpositionen des Aufrufs identisch mit den temporären Variablen der Affixformen der linken Regelseiten des anonymen Nichtterminals sind. Dabei ist darauf zu achten, daß die Werte temporärer Variablen für die Evaluation eines Nichtterminals nicht durch die Evaluation des eingebetteten anonymen Nichtterminals überschrieben werden. Dieses Problem wird durch die Wahl disjunkter temporärer Variablen gelöst. Dazu werden hier für eine triviale Affixform beim Aufruf *zwei* Oberon-Variablen verwendet, und zwar eine für die Variable auf der Position und eine für die Position.

Die für die Generierung berechneten temporären Variablen werden in folgenden Datenstrukturen repräsentiert:

```
VAR
  HNontVars: POINTER TO ARRAY OF INTEGER;
  FormalName: POINTER TO ARRAY OF INTEGER;
  AffixName: POINTER TO ARRAY OF INTEGER;
  VarName: POINTER TO ARRAY OF INTEGER;
  NodeName: POINTER TO ARRAY OF INTEGER;
```

Eine temporäre Variable wird durch eine Zahl dargestellt. Die temporären Variablen werden aufsteigend vergeben. Für ihre Deklaration genügt daher die Kenntnis der Anzahl der in der Prozedur vorkommenden Variablen. In dieser in `HNontVars` dargestellten Zahl sind allerdings die Namen für die formalen Prozedurparameter mit berücksichtigt.

Die Oberon-Variablen für Variablen und Affixformen sind in den Feldern `VarName` und `AffixName` abgespeichert, die temporären Variablen für Analysen und Synthesen in dem Feld `NodeName`.

In dem Feld `FormalName` sind die zusätzlich für die Evaluation von Wiederholungen benötigten Variablen abgespeichert. Die Laufvariablen stehen dabei in Einträgen von `AffixName` für die Affixpositionen der linken Seite und die der Rekursionparameter, die Wurzelvariablen in dem Feld `FormalName`. Das Feld `Actual` wird „lokal“ zu `ComputeVarNames` dazu verwendet, sicherzustellen, daß die jeweiligen Laufvariablen in allen Alternativen identisch sind.

Die Vergabe der Variablen erfolgt durch eine die Evaluation vorwegnehmende Traversierung der Regeln in der Prozedur `ComputeVarNames`.

Um die Anzahl der temporären Variablen so gering wie möglich zu halten, wird eine Lebenszeitanalyse durchgeführt. Für die Durchführung wird ein Referenzzählerverfahren verwendet. Ein Referenzzähler signalisiert die Benutzung einer Oberon-Variablen. Durch Erhöhung des Referenzzählers wird eine verfrühte Wiederbenutzung einer Oberon-Variablen verhindert.

Um die Lebensdauer einer temporären Variable für eine Variable *M* zu ermitteln, werden deren Abhängigkeiten im Gültigkeitsbereich symbolisch herabgezählt. Diese bestehen aus der Anzahl der Variablenvorkommen, erhöht um 1, falls die Variable *#M* im Gültigkeitsbereich vorkommt, da sich in diesem Fall die Lebensdauer bis zu dem notwendigen Vergleich verlängert. Diese Abhängigkeiten werden in dem Feld `VarDeps` dargestellt und „lokal“ zu `ComputeVarNames` verwendet.

```
VAR
  VarDeps: POINTER TO ARRAY OF INTEGER;
```

#### 6.1.2.4 Datenstrukturen für die Generierung

Folgende Datenstrukturen werden weiterhin für die Generierung benötigt:

```
VAR
  SavePos, UseConst, UseRefCnt, TraversePass: BOOLEAN;
  VarCnt, VarAppls: POINTER TO ARRAY OF INTEGER;
```

Die Schalter steuern die Generierung. In den Feldern `VarCnt` und `VarAppls` sind statistische Informationen zu den Variablen einer Alternative abgelegt. `VarCnt` bezeichnet die Anzahl der Variablenvorkommen, `VarAppls`, wie oft die Variable auf applizierender Position vorkommt.

```
VAR
  NodeIdent: POINTER TO ARRAY OF INTEGER;
  MaxMAlt: INTEGER;
  RefConst: LONGINT;
```

In dem Feld `NodeIdent` wird jeder Meta-Regel ein Knotenbezeichner zugeordnet. Die Einträge werden in der Prozedur `ComputeNodeIdent` berechnet. Dabei werden die Werte `ArityConst` und `RefConst` zur Bestimmung der Komponenten bestimmt.

```
VAR
  Leaf: POINTER TO ARRAY OF INTEGER;
  AffixPlace: POINTER TO ARRAY OF INTEGER;
  AffixSpace: POINTER TO ARRAY OF INTEGER;
  FirstHeap: INTEGER;
```

Bei der Generierung mit Optimierung terminaler Affixformen werden Verweise auf ein bereits synthetisiertes Blatt bzw. einen konstanten Baum direkt in den generierten Code eingetragen. Diese Verweise werden in der Prozedur `ComputeConstDat` berechnet und in die Felder `Leaf` und `AffixPlace` eingetragen. Dabei wird für jede Affixposition die Größe eines zu synthetisierenden Baums bestimmt; diese im Feld `AffixSpace` hinterlegte Größe wird in Synthesen für die Überlaufüberprüfung verwendet. Die Variable `FirstHeap` bezeichnet den ersten freien Eintrag im Heap des generierten Compilers. Dieser Wert wird für eine korrekte Initialisierung verwendet.

```
VAR
  VarRefCnt: POINTER TO ARRAY OF INTEGER;
  VarDepPos: POINTER TO ARRAY OF INTEGER;
```

In den Feldern `VarRefCnt` und `VarDepPos` werden Informationen für die Optimierung der Freispeicherverwaltung in den Prozeduren der Grundnichtterminale abgelegt. Ein Eintrag des Feldes `VarRefCnt` ordnet einer Variablen denjenigen Wert zu, um den der Referenzzähler bei der Definition der Variablen erhöht werden muß. Ein positiver Eintrag in `VarDepPos` bezeichnet diejenige Affixposition, an der nach einem Vergleich die Freigabe des Unterbaums erfolgen kann.

```
VAR
  RepAppls: POINTER TO ARRAY OF BOOLEAN;
  RepVar, EmptySet: Sets.OpenSet;
```

Die Menge `RepVar` enthält diejenigen Variablen, die auf definierenden Positionen von Rekursionsparametern vorkommen. Diese Menge wird für die Synthesen in Wiederholungen benötigt. Die Menge `EmptySet` stellt die leere Menge dar und wird Synthesen in allen anderen Fällen übergeben. In dem Feld `RepAppls` ist zu jeder Variablen auf definierender Position der Rekursionsparameter die Information gespeichert, ob sie in allen Alternativen der Wiederholung appliziert wird.

## 6.2 Die Ausgabe der Übersetzung

Im generierten Compiler stellen die vom Evaluator berechneten Werte der Ausgabeparameter des Startsymbols die Übersetzung dar. Das Modul `EmitGen` generiert Code, der diese Übersetzung ausgibt. Intern wird die Übersetzung vom Evaluator durch einen Ableitungsbaum zur Meta-Grammatik dargestellt. Die Ausgabe erfolgt daher durch eine Traversierung dieses Ableitungsbaums.

Dabei ist zu beachten, daß die Ausgabe von Alternativen eines Nichtterminals, das mit einer Tokenmarkierung versehen ist (einem sogenannten Token) so wie deren Unterbäume bündig erfolgen soll, d.h. ohne die Ausgabe eines Trennzeichens zwischen den einzelnen Terminalen; im anderen Fall jedoch soll nach jedem Terminal ein Trennzeichen ausgegeben werden. Dadurch ergibt sich die Situation, daß ein Nichtterminal ein Subtoken sein kann, d.h. sowohl in einem Unterbaum eines Tokens als auch in einem nicht von einem Token ausgehenden Unterbaum vorkommen kann. Bei der Ausgabe von Alternativen eines Subtokens müssen also abhängig vom Kontext im Ableitungsbaum Trennzeichen ausgegeben bzw. nicht ausgegeben werden.

Die Meta-Nichtterminale lassen sich jetzt nach dieser Ausgabeeigenschaft in zwei Mengen einteilen. In der Menge `Type3` sollen sich die Token sowie die von Token erreichbaren Nichtterminale befinden. In der Menge `Type2` sollen sich die vom Startsymbol der Meta-Grammatik ausgehend erreichbaren Nichtterminale befinden, jedoch ohne die Token und ohne die nur über Token erreichbaren Nichtterminale. Die Subtoken bilden also genau die Schnittmenge dieser beiden Mengen.

Die Ausgabe erfolgt in der Implementierung durch Prozeduren, in denen eine Fallunterscheidung über Alternativen eines Nichtterminals stattfindet. Für die Ausgabe mit und ohne Trennzeichen werden getrennte Prozeduren generiert. Für Subtoken werden insbesondere zwei Prozeduren generiert. In den Prozeduren wird zusätzlich die Größe des ausgegebenen Baums berechnet, d.h. die Anzahl der benötigten Heapeinträge bei einer nichtkollabierten Speicherweise.

Die generierten Prozeduren lassen sich durch folgendes Codeschema darstellen:

```
(* Meta-Nichtterminal M *)
PROCEDURE EmitMType2/3 (Ptr: HeapType);
BEGIN
  INC(OutputSize, Heap[Ptr] DIV arityConst + 1);
  CASE Heap[Ptr] MOD arityConst OF
  | 1:  Code für erste Alternative
     ⋮
  | n:  Code für n-te Alternative
  END
END EmitMType2/3;

(* Ausgabe von Meta-Terminal T *)
IO.WriteText(Out, T); (evtl. IO.Write(Trennzeichen); )

(* Prozeduraufruf für Meta-Nichtterminal N *)
EmitNType2/3(Heap[Ptr + i]);
```

Die Generierung der Ausgabeprozeduren erfolgt durch die Prozedur `GenEmitProc`. In dieser erfolgt die Berechnung der Mengen `Type2` und `Type3` durch Traversierung der Meta-Grammatik. Diese wird von der rekursiven Prozedur `CalcSets` ausgeführt. `GenEmitCall` erzeugt den Aufruf der korrekten Prozedur. `GenShowHeap` generiert die Ausgabe von Informationen über den Speicherbedarf der generierten Compiler .

## 6.3 Implementierungen

### 6.3.1 eSLEAGGen.Mod

```
MODULE eSLEAGGen;   (* JoDe 12.12.96, Version 1.06 *)

  IMPORT Sets := eSets, IO := eIO, EAG := eEAG;

  CONST
    parsePass* = 0; onePass* = 1; sSweepPass* = 2;
  TYPE
    OpenInt = POINTER TO ARRAY OF INTEGER;
    OpenBool = POINTER TO ARRAY OF BOOLEAN;
  VAR
    Mod, RC: IO.TextOut;
    SavePos, UseConst, UseRefCnt, TraversePass, DebugRC: BOOLEAN;
    VarCnt, VarAppls: OpenInt;
    Testing, Generating: BOOLEAN; PreparedHNonts: Sets.OpenSet;
(* ComputeVarNames *)
    VarDeps: OpenInt;
    HNontDef: Sets.OpenSet;
    HNontVars: OpenInt;
    AffixName: OpenInt;
    VarName: OpenInt;
    NodeName: OpenInt;
    FormalName: OpenInt;
    ActualName: OpenInt;
(* ComputeConstDat *)
    FirstHeap: INTEGER;
    Leaf: OpenInt;
    AffixPlace: OpenInt;
    AffixSpace: OpenInt;
(* ComputeNodeIdent *)
    NodeIdent: OpenInt;
    ArityConst: INTEGER; RefConst: LONGINT;
(* generating procs *)
    VarRefCnt: OpenInt;
    VarDepPos: OpenInt;
    IfLevel: INTEGER;      (* wie viele IF-Konstrukte muessen nach Analyse in Praedikaten geschlossen werden *)
    RepAppls: OpenBool;
    RepVar, EmptySet: Sets.OpenSet;


  PROCEDURE InitScope*(Scope: EAG.ScopeDesc);
    VAR i: INTEGER;
  BEGIN
    FOR i := Scope.Beg TO Scope.End - 1 DO EAG.Var[i].Def := FALSE END
  END InitScope;

  PROCEDURE PrepareInit;
  BEGIN
    NEW(VarCnt, EAG.NextVar); NEW(VarAppls, EAG.NextVar);
    Sets.New(RepVar, EAG.NextVar); Sets.New(PreparedHNonts, EAG.NextHNont)
  END PrepareInit;

  PROCEDURE PrepareFinit;
  BEGIN
     VarCnt := NIL; VarAppls := NIL; RepVar := NIL; PreparedHNonts := NIL
  END PrepareFinit;

  PROCEDURE Prepare(N: INTEGER);
    VAR Node: EAG.Rule; A: EAG.Alt; F: EAG.Factor; P: INTEGER;

    PROCEDURE Traverse(P: INTEGER);

      PROCEDURE DefPos(Node: INTEGER);
        VAR n: INTEGER;
      BEGIN
        IF Node < 0 THEN
          INC(VarCnt[- Node])
        ELSE
          FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
            DefPos(EAG.NodeBuf[Node + n])
          END
        END
      END DefPos;
```

```
    PROCEDURE ApplPos(Node: INTEGER);
      VAR n: INTEGER;
    BEGIN
      IF Node < 0 THEN
        INC(VarCnt[- Node]); INC(VarAppls[- Node])
      ELSE
        FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
          ApplPos(EAG.NodeBuf[Node + n])
        END
      END
    END ApplPos;

  BEGIN (* Traverse(P: INTEGER); *)
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF EAG.ParamBuf[P].isDef THEN DefPos(EAG.ParamBuf[P].Affixform)
      ELSE ApplPos(EAG.ParamBuf[P].Affixform)
      END;
      INC(P)
    END
  END Traverse;

  PROCEDURE InitArray(Scope: EAG.ScopeDesc);
    VAR i: INTEGER;
  BEGIN
    FOR i := Scope.Beg TO Scope.End - 1 DO VarCnt[i] := 0; VarAppls[i] := 0 END
  END InitArray;

BEGIN (* Prepare(N: INTEGER); *)
  IF ~ Sets.In(PreparedHNonts, N) THEN
    Node := EAG.HNont[N].Def;
    IF Node IS EAG.Rep THEN
      InitArray(Node(EAG.Rep).Scope); Traverse(Node(EAG.Rep).Formal.Params)
    ELSIF Node IS EAG.Opt THEN
      InitArray(Node(EAG.Opt).Scope); Traverse(Node(EAG.Opt).Formal.Params)
    END;
    A := Node.Sub;
    REPEAT
      InitArray(A.Scope); Traverse(A.Formal.Params);
      F := A.Sub;
      WHILE F # NIL DO
        IF F IS EAG.Nont THEN Traverse(F(EAG.Nont).Actual.Params) END;
        F := F.Next
      END;
      IF Node IS EAG.Rep THEN
        Traverse(A.Actual.Params); P := A.Actual.Params;
        WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
          IF EAG.ParamBuf[P].isDef & (EAG.ParamBuf[P].Affixform < 0) THEN
            Sets.Incl(RepVar, - EAG.ParamBuf[P].Affixform)
          END;
          INC(P)
        END
      END;
      A := A.Next
    UNTIL A = NIL;
    Sets.Incl(PreparedHNonts, N)
  END
END Prepare;

PROCEDURE TestHNont(N: INTEGER; EmitErr, SLEAG: BOOLEAN): BOOLEAN;
  VAR Node: EAG.Rule; A: EAG.Alt; F: EAG.Factor; isSLEAG, isLEAG: BOOLEAN; V: INTEGER;

  PROCEDURE Error(Pos: IO.Position; Msg: ARRAY OF CHAR);
  BEGIN
    isSLEAG := FALSE;
    IF EmitErr THEN
      IO.WriteText(IO.Msg, "\n\t"); IO.WritePos(IO.Msg, Pos); IO.WriteText(IO.Msg, "\t");
      IO.WriteText(IO.Msg, Msg); IO.Update(IO.Msg)
    END
  END Error;

  PROCEDURE CheckDefPos(P: INTEGER);

    PROCEDURE DefPos(Node: INTEGER);
      VAR n, V: INTEGER;
    BEGIN
      IF Node < 0 THEN V := - Node;
        IF ~ EAG.Var[V].Def THEN EAG.Var[V].Def := TRUE END;
      ELSE
        FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
          DefPos(EAG.NodeBuf[Node + n])
```

```
        END
      END
    END DefPos;

  BEGIN  (* CheckDefPos(P: INTEGER) *)
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF EAG.ParamBuf[P].isDef THEN
      DefPos(EAG.ParamBuf[P].Affixform) END;
      INC(P)
    END
  END CheckDefPos;

  PROCEDURE CheckApplPos(P: INTEGER);

    PROCEDURE ApplPos(Node: INTEGER);
      VAR n, V: INTEGER;
    BEGIN
      IF Node < 0 THEN V := - Node;
        IF ~ EAG.Var[V].Def THEN
          Error(EAG.Var[V].Pos, "not left-defining"); isLEAG := FALSE
        END
      ELSE
        FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
          ApplPos(EAG.NodeBuf[Node + n])
        END
      END
    END ApplPos;

  BEGIN  (* CheckApplPos(P: INTEGER) *)
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF ~ EAG.ParamBuf[P].isDef THEN ApplPos(EAG.ParamBuf[P].Affixform) END;
      INC(P)
    END
  END CheckApplPos;

  PROCEDURE CheckSLEAGCond(P: INTEGER);
    VAR Node: INTEGER;
  BEGIN
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      Node := EAG.ParamBuf[P].Affixform;
      IF EAG.ParamBuf[P].isDef THEN
        IF Node >= 0 THEN
          Error(EAG.ParamBuf[P].Pos, "Can't generate anal-predicate here")
        ELSIF EAG.Var[- Node].Def THEN
          Error(EAG.ParamBuf[P].Pos, "Can't generate equal-predicate here")
        ELSIF EAG.Var[EAG.Var[- Node].Neg].Def THEN
          Error(EAG.ParamBuf[P].Pos, "Can't generate unequal-predicate here")
        ELSIF VarAppls[- Node] > 1 THEN
          Error(EAG.ParamBuf[P].Pos, "Can't synthesize this variable several times")
        END
      END;
      INC(P)
    END
  END CheckSLEAGCond;

BEGIN (* TestHNont(N: INTEGER; EmitErr, SLEAG: BOOLEAN): BOOLEAN *)
  ASSERT(Sets.In(EAG.Prod, N), 98); isSLEAG := TRUE; isLEAG := TRUE;
  Prepare(N);
  Node := EAG.HNont[N].Def;
  IF Node IS EAG.Rep THEN
    InitScope(Node(EAG.Rep).Scope);
    CheckDefPos(Node(EAG.Rep).Formal.Params);
    CheckApplPos(Node(EAG.Rep).Formal.Params)
  ELSIF Node IS EAG.Opt THEN
    InitScope(Node(EAG.Opt).Scope);
    CheckDefPos(Node(EAG.Opt).Formal.Params);
    CheckApplPos(Node(EAG.Opt).Formal.Params)
  END;
  A := Node.Sub;
  REPEAT
    InitScope(A.Scope);
    CheckDefPos(A.Formal.Params);
    F := A.Sub;
    WHILE F # NIL DO
      IF F IS EAG.Nont THEN
        CheckApplPos(F(EAG.Nont).Actual.Params);
        IF SLEAG & (EAG.HNont[F(EAG.Nont).Sym].Id < 0) THEN (* & ~ isPred & isProd *)
          isSLEAG := isSLEAG & TestHNont(F(EAG.Nont).Sym, EmitErr, SLEAG)
        END;
        CheckDefPos(F(EAG.Nont).Actual.Params);
```

```
      END;
      F := F.Next;
    END;
    IF Node IS EAG.Rep THEN
      CheckApplPos(A.Actual.Params);
      IF SLEAG THEN CheckSLEAGCond(A.Actual.Params) END;
      CheckDefPos(A.Actual.Params);
    END;
    CheckApplPos(A.Formal.Params);
    A := A.Next
  UNTIL A = NIL;
  IF SLEAG THEN RETURN isSLEAG ELSE RETURN isLEAG END
END TestHNont;

PROCEDURE IsSLEAG*(N: INTEGER; EmitErr: BOOLEAN): BOOLEAN;
BEGIN
  RETURN TestHNont(N, EmitErr, TRUE)
END IsSLEAG;

PROCEDURE IsLEAG*(N: INTEGER; EmitErr: BOOLEAN): BOOLEAN;
BEGIN
  RETURN TestHNont(N, EmitErr, FALSE)
END IsLEAG;

PROCEDURE InitTest*;
BEGIN
  IF ~ Generating & ~ Testing THEN PrepareInit END; Testing := TRUE
END InitTest;

PROCEDURE FinitTest*;
BEGIN
  IF ~ Generating THEN PrepareFinit END; Testing := FALSE
END FinitTest;

PROCEDURE PredsOK*(): BOOLEAN;
  VAR N: INTEGER; OK: BOOLEAN;
BEGIN
  OK := TRUE;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(EAG.Pred, N) THEN OK := OK & IsLEAG(N, TRUE) END
  END;
  RETURN OK
END PredsOK;

PROCEDURE Test*;
  VAR N: INTEGER; isSLEAG, isLEAG: BOOLEAN;
BEGIN
  IO.WriteString(IO.Msg, "SLEAG testing   "); IO.WriteString(IO.Msg, EAG.BaseName); IO.Update(IO.Msg);
  IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
    EXCL(EAG.History, EAG.isSLEAG);
    InitTest; isSLEAG := TRUE; isLEAG := TRUE;
    FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
      IF Sets.In(EAG.Prod, N) THEN (* & ~ Sets.In(EAG.Pred, N)  *)
        IF isSLEAG & (EAG.HNont[N].Id >= 0) THEN
          IF ~ TestHNont(N, TRUE, TRUE) THEN
            isSLEAG := FALSE;
            IF ~ TestHNont(N, FALSE, FALSE) THEN isLEAG := FALSE END
          END
        END
      END
    END;
    IF isSLEAG THEN
      INCL(EAG.History, EAG.isSLEAG); IO.WriteText(IO.Msg, "   ok")
    ELSE
      IF isLEAG THEN
        IO.WriteText(IO.Msg, "\n\tno SLEAG but LEAG")
      ELSE
        IO.WriteText(IO.Msg, "\n\tno LEAG")
      END
    END
  END;
  FinitTest;
  IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END Test;

  (* ------------------------ BerechnungsProzeduren ------------------------- *)

PROCEDURE ComputeNodeIdent;
  VAR N, A, i, temp: INTEGER;
BEGIN
```

```
  NEW(NodeIdent, EAG.NextMAlt);
  FOR A := EAG.firstMAlt TO EAG.NextMAlt - 1 DO NodeIdent[A] := -1 END;
  ArityConst := 0;
  FOR N := EAG.firstMNont TO EAG.NextMNont - 1 DO
    A := EAG.MNont[N].MRule; i := 0;
    WHILE A # EAG.nil DO INC(i); NodeIdent[A] := i; A := EAG.MAlt[A].Next END;
    IF i > ArityConst THEN ArityConst := i END
  END;
  i := 1; WHILE i <= ArityConst DO i := i*2 END; ArityConst := i; RefConst := 0;
  FOR A := EAG.firstMAlt TO EAG.NextMAlt - 1 DO
    ASSERT(NodeIdent[A] >=0, 89);
    (* INC(NodeIdent[A], EAG.MAlt[A].Arity * ArityConst); *)
    temp := NodeIdent[A] + (EAG.MAlt[A].Arity * ArityConst); NodeIdent[A] := temp;
    IF RefConst < NodeIdent[A] THEN RefConst := NodeIdent[A] END
  END;
  i := 1; WHILE i <= RefConst DO i := i*2 END; RefConst := i;
END ComputeNodeIdent;

PROCEDURE ComputeConstDat;
  VAR A, i, NextHeap: INTEGER;

  PROCEDURE Traverse(N: INTEGER; VAR NextHeap: INTEGER);
    VAR Node: EAG.Rule; A: EAG.Alt; F: EAG.Factor;

    PROCEDURE CheckParams(P: INTEGER; VAR NextHeap: INTEGER);
      VAR isConst: BOOLEAN; Tree: INTEGER;

      PROCEDURE TraverseTree(Node: INTEGER; VAR Next: INTEGER);
        VAR n, Arity: INTEGER;
      BEGIN
        IF Node < 0 THEN
          isConst := FALSE
        ELSE
          Arity := EAG.MAlt[EAG.NodeBuf[Node]].Arity;
          IF ~ UseConst OR (Arity # 0) THEN INC(Next, 1+ Arity) END;
          FOR n := 1 TO Arity DO
            TraverseTree(EAG.NodeBuf[Node + n], Next)
          END
        END
      END TraverseTree;

    BEGIN  (* CheckParams(P: INTEGER; VAR NextHeap: INTEGER) *)
      WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
        Tree := EAG.ParamBuf[P].Affixform; isConst := TRUE;
        TraverseTree(Tree, AffixSpace[P]);
        IF (Tree > 0) & (EAG.MAlt[EAG.NodeBuf[Tree]].Arity = 0) THEN
          IF isConst THEN AffixPlace[P] := Leaf[EAG.NodeBuf[Tree]] END
        ELSE
          IF isConst THEN
            AffixPlace[P] := NextHeap; INC(NextHeap, AffixSpace[P])
          END
        END;
        INC(P)
      END;
    END CheckParams;

  BEGIN (* Traverse(N: INTEGER; VAR NextHeap: INTEGER) *)
    Node := EAG.HNont[N].Def;
    IF Node IS EAG.Rep THEN
      CheckParams(Node(EAG.Rep).Formal.Params, NextHeap)
    ELSIF Node IS EAG.Opt THEN
      CheckParams(Node(EAG.Opt).Formal.Params, NextHeap)
    END;
    A := Node.Sub;
    REPEAT
      CheckParams(A.Formal.Params, NextHeap);
      F := A.Sub;
      WHILE F # NIL DO
        IF F IS EAG.Nont THEN CheckParams(F(EAG.Nont).Actual.Params, NextHeap) END;
        F := F.Next
      END;
      IF Node IS EAG.Rep THEN
        CheckParams(A.Actual.Params, NextHeap)
      END;
      A := A.Next
    UNTIL A = NIL
  END Traverse;

BEGIN (* ComputeConstDat; *)
  NEW(AffixSpace, EAG.NextParam); NEW(AffixPlace, EAG.NextParam);
```

```
  FOR i := EAG.firstParam TO EAG.NextParam - 1 DO AffixSpace[i] := 0; AffixPlace[i] := -1 END;
  NEW(Leaf, EAG.NextMAlt);
  NextHeap := EAG.MaxMArity + 1;
  FirstHeap := NextHeap;
  FOR A := EAG.firstMAlt TO EAG.NextMAlt - 1 DO
    IF EAG.MAlt[A].Arity = 0 THEN Leaf[A] := NextHeap; INC(NextHeap)
    ELSE Leaf[A] := -1
    END
  END;
  FOR i := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(EAG.Prod, i) THEN Traverse(i, NextHeap) END
  END;
  IF UseConst THEN FirstHeap := NextHeap END
END ComputeConstDat;

PROCEDURE ComputeVarNames*(N: INTEGER; Embed: BOOLEAN);
  VAR
    FreeVar, RefCnt: OpenInt; Top, NextFreeVar: INTEGER;
    temp: INTEGER;

  PROCEDURE WriteRefCnt;
    VAR i: INTEGER;
  BEGIN
    IO.WriteText(RC, "WriteRefCnt: ");
    FOR i := 0 TO NextFreeVar DO IO.WriteInt(RC, RefCnt[i]); IO.WriteText(RC, ", ") END;
    IO.WriteText(RC, " : \n");
  END WriteRefCnt;

  PROCEDURE VarExpand;
    VAR Int: OpenInt; i: INTEGER;
  BEGIN
    IF NextFreeVar >= LEN(RefCnt^) THEN
      NEW(Int, 2 * LEN(RefCnt^));
      FOR i := 0 TO NextFreeVar - 1 DO Int[i] := RefCnt[i] END;
      FOR i := NextFreeVar TO SHORT(LEN(Int^) - 1) DO Int[i] := 0 END;
      RefCnt := Int
    END;
    IF Top >= LEN(FreeVar^) THEN
      NEW(Int, 2 * LEN(FreeVar^)); FOR i := 0 TO Top - 1 DO Int[i] := FreeVar[i] END; FreeVar := Int
    END
  END VarExpand;

  PROCEDURE GetFreeVar(): INTEGER;
    VAR Name: INTEGER;
  BEGIN
    IF Top > 0 THEN
      DEC(Top); Name := FreeVar[Top]
    ELSE
      INC(NextFreeVar); IF NextFreeVar >= LEN(RefCnt^) THEN VarExpand END;
      RefCnt[NextFreeVar] := 0; Name := NextFreeVar
    END;
    IF DebugRC THEN IO.WriteText(RC, " -"); IO.WriteInt(RC, Name) END;
    RETURN Name
  END GetFreeVar;

  PROCEDURE Dispose(Var: INTEGER);
  BEGIN
    DEC(RefCnt[Var]);
    IF RefCnt[Var] = 0 THEN
      IF DebugRC THEN IO.WriteText(RC, " +"); IO.WriteInt(RC, Var) END;
      FreeVar[Top] := Var; INC(Top); IF Top >= LEN(FreeVar^) THEN VarExpand END
    END
  END Dispose;

  PROCEDURE Traverse(N: INTEGER);
    VAR
      Node: EAG.Rule; A: EAG.Alt; F: EAG.Factor;
      Dom, P, Tree: INTEGER; Repetition, isPred: BOOLEAN;

    PROCEDURE CheckDefPos(P: INTEGER);
      VAR Tree, V: INTEGER;

      PROCEDURE DefPos(Node, Var: INTEGER);
        VAR n, Arity, Node1, Var1, V, Vn: INTEGER; NeedVar: BOOLEAN;
      BEGIN
        IF Node < 0 THEN V := - Node;
          IF ~ EAG.Var[V].Def THEN EAG.Var[V].Def := TRUE;
            IF VarName[V] < 0 THEN
              VarName[V] := GetFreeVar(); INC(RefCnt[VarName[V]])
            END;
```

```
        IF EAG.Var[EAG.Var[V].Neg].Def THEN Vn := EAG.Var[V].Neg;
          DEC(VarDeps[V]); DEC(VarDeps[Vn]);
          IF VarDeps[Vn] = 0 THEN VarDepPos[Vn] := P; Dispose(VarName[Vn]) END
        END
      ELSE
        IF VarDeps[V] = 1 THEN VarDepPos[V] := P END
      END;
      DEC(VarDeps[V]); IF VarDeps[V] = 0 THEN Dispose(VarName[V]) END
    ELSE
      Arity := EAG.MAlt[EAG.NodeBuf[Node]].Arity;
      (*NodeName[Node] := Var; *)
      IF Arity # 0 THEN NodeName[Node] := Var END;
      FOR n := 1 TO Arity DO
        Node1 := EAG.NodeBuf[Node + n];
        NeedVar := (((isPred OR UseRefCnt) & (Var = AffixName[P])) OR (n # Arity)) & (Node1 >= 0)
                & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity > 0);
        IF NeedVar THEN Var1 := GetFreeVar(); INC(RefCnt[Var1])
        ELSE Var1 := Var END;
        DefPos(Node1, Var1);
        IF NeedVar THEN Dispose(Var1) END;
      END
    END
  END DefPos;

BEGIN (* CheckDefPos(P: INTEGER) *)
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    Tree := EAG.ParamBuf[P].Affixform;
    IF EAG.ParamBuf[P].isDef THEN
      IF (Tree < 0) & (VarName[- Tree] < 0) THEN V := - Tree;
        VarName[V] := AffixName[P]; INC(RefCnt[VarName[V]])
      END;
      DefPos(Tree, AffixName[P])
    END;
    INC(P)
  END
END CheckDefPos;

PROCEDURE CheckApplPos(P: INTEGER; Repetition: BOOLEAN);
  VAR Tree, V, P1: INTEGER;

  PROCEDURE ApplPos(Node, Var: INTEGER);
    VAR n, Arity, Node1, Var1, V: INTEGER; NeedVar: BOOLEAN;
  BEGIN
    IF Node < 0 THEN V := - Node;
      DEC(VarDeps[V]); IF VarDeps[V] = 0 THEN Dispose(VarName[V]) END;
      IF VarDepPos[V] >= 0 THEN VarDepPos[V] := -1 END
    ELSE
      Arity := EAG.MAlt[EAG.NodeBuf[Node]].Arity;
      NodeName[Node] := Var;
      FOR n := 1 TO Arity DO
        Node1 := EAG.NodeBuf[Node + n];
        NeedVar := ~ (UseConst & (AffixPlace[P] > 0))
                & UseRefCnt & ((Var = AffixName[P]) OR (n # Arity))
                & (Node1 >= 0) & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity > 0);
        IF NeedVar THEN Var1 := GetFreeVar(); INC(RefCnt[Var1])
        ELSE Var1 := Var END;
        ApplPos(Node1, Var1);
        IF NeedVar THEN Dispose(Var1) END;
      END
    END
  END ApplPos;

BEGIN  (* CheckApplPos(P: INTEGER; Repetition: BOOLEAN) *)
  P1 := P;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF (Tree < 0) & (VarName[- Tree] < 0) THEN V := - Tree;
        VarName[V] := AffixName[P]; INC(RefCnt[VarName[V]])
      END;
      IF ~ (UseConst & (AffixPlace[P] > 0)) & Repetition & (Tree >= 0) THEN
        NodeName[Tree] := GetFreeVar(); INC(RefCnt[NodeName[Tree]]);
        ApplPos(Tree, NodeName[Tree])
      ELSE
        ApplPos(Tree, AffixName[P])
      END
    END;
    INC(P)
  END;
  IF Repetition THEN
```

```
    WHILE EAG.ParamBuf[P1].Affixform # EAG.nil DO
      IF ~ EAG.ParamBuf[P1].isDef & ~ (UseConst & (AffixPlace[P1] > 0)) THEN
        Tree := EAG.ParamBuf[P1].Affixform;
        IF Tree >= 0 THEN Dispose(NodeName[Tree]) END
      END;
      INC(P1)
    END
  END
END CheckApplPos;

PROCEDURE GetFormalParamNames(N, P: INTEGER);
  VAR Repetition: BOOLEAN; Dom, Tree, V: INTEGER;
BEGIN
  Repetition := ~ Sets.In(EAG.Pred, N) & (EAG.HNont[N].Def IS EAG.Rep);
  Dom := EAG.HNont[N].Sig;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF Repetition THEN
      AffixName[P] := ActualName[Dom]
    ELSE
      AffixName[P] := FormalName[Dom];
      Tree := EAG.ParamBuf[P].Affixform;
      IF ~ EAG.ParamBuf[P].isDef & (Tree < 0) THEN V := - Tree;
        (* Optimize usage of variables of formal parameters *)
        VarName[V] := AffixName[P]; INC(RefCnt[VarName[V]])
      END
    END;
    INC(P); INC(Dom)
  END
END GetFormalParamNames;

PROCEDURE GetActualParamNames(N, P: INTEGER);
  VAR P1, Tree, V: INTEGER;

  PROCEDURE FindVarName(P, VarName: INTEGER): INTEGER;
  BEGIN
    WHILE AffixName[P] # VarName DO INC(P) END;
    RETURN P
  END FindVarName;

BEGIN (* GetActualParamNames(N, P: INTEGER) *)
  P1 := P;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    Tree := EAG.ParamBuf[P].Affixform;
    IF AffixName[P] < 0 THEN
      IF (Tree < 0) & (VarName[- Tree] >=  0) THEN V := - Tree;
        IF ~ EAG.ParamBuf[P].isDef THEN
          IF Embed & ~ Sets.In(EAG.Pred, N) & (EAG.HNont[N].Id < 0) & (VarDeps[V] > 1) THEN
            (* content of variable can be lost in anonym nont *)
            AffixName[P] := GetFreeVar()
          ELSIF Embed & ~ Sets.In(EAG.Pred, N) & (EAG.HNont[N].Id < 0) THEN
            (* analyze in anonym nont can destroy content of variable *)
            AffixName[P] := VarName[V];
            IF FindVarName(P1, VarName[V]) # P THEN AffixName[P] := GetFreeVar() END
          ELSE
            AffixName[P] := VarName[V]
          END
        ELSE (* EAG.ParamBuf[P].isDef *)
          AffixName[P] := VarName[V];
          IF EAG.Var[V].Def OR (FindVarName(P1, VarName[V]) # P) THEN
            AffixName[P] := GetFreeVar()
          END
        END
      ELSE
        AffixName[P] := GetFreeVar();
      END
    END;
    INC(RefCnt[AffixName[P]]);
    IF isPred & EAG.ParamBuf[P].isDef THEN INC(RefCnt[AffixName[P]]) END;
    INC(P)
  END
END GetActualParamNames;

PROCEDURE FreeActualParamNames(P: INTEGER);
BEGIN
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    Dispose(AffixName[P]); INC(P)
  END
END FreeActualParamNames;

PROCEDURE FreeAllDefPosVarNames(A: EAG.Alt);
```

```
    VAR F: EAG.Factor;

    PROCEDURE FreeVarNames(P: INTEGER);
    BEGIN
      WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
        IF EAG.ParamBuf[P].isDef THEN Dispose(AffixName[P]) END; INC(P)
      END
    END FreeVarNames;

  BEGIN (* FreeAllDefPosVarNames(A: EAG.Alt); *)
    F := A.Sub;
    WHILE F # NIL DO
      IF F IS EAG.Nont THEN FreeVarNames(F(EAG.Nont).Actual.Params) END;
      F := F.Next
    END;
    IF Node IS EAG.Rep THEN FreeVarNames(A.Actual.Params) END
  END FreeAllDefPosVarNames;

  PROCEDURE InitComputation(Scope: EAG.ScopeDesc);
    VAR i: INTEGER;
  BEGIN
    InitScope(Scope);
    FOR i := Scope.Beg TO Scope.End - 1 DO
      VarDeps[i] := VarCnt[i]; IF EAG.Var[i].Neg # EAG.nil THEN INC(VarDeps[i]) END
    END;
    IF DebugRC THEN IO.WriteText(RC, "\nOpen: ") END
  END InitComputation;

  PROCEDURE FinitComputation(Scope: EAG.ScopeDesc);
    VAR i: INTEGER;
  BEGIN
    IF DebugRC THEN IO.WriteText(RC, "\nClose: "); END;
    FOR i := Scope.Beg TO Scope.End - 1 DO
      VarRefCnt[i] := VarAppls[i]; IF VarDepPos[i] >= 0 THEN INC(VarRefCnt[i]) END;
      IF DebugRC THEN
        EAG.WriteVar(RC, i); IO.Write(RC, " ");
        IO.WriteInt(RC, VarDeps[i]); IO.WriteString(RC, " V");
        IO.WriteInt(RC, VarName[i]); IO.WriteString(RC, " (");
        IO.WriteInt(RC, RefCnt[VarName[i]]); IO.WriteString(RC, "), ")
      END
    END;
    IF DebugRC THEN IO.WriteLn(RC) END
  END FinitComputation;

BEGIN (* Traverse(N: INTEGER) *)
  Prepare(N);
  IF DebugRC THEN
    IO.WriteText(RC, "\nStart: "); EAG.WriteHNont(RC, N); IO.Write(RC, ":"); WriteRefCnt
  END;
  Node := EAG.HNont[N].Def; isPred := Sets.In(EAG.Pred, N);
  Repetition := ~ isPred & (Node IS EAG.Rep);
  Dom := EAG.HNont[N].Sig;
  WHILE EAG.DomBuf[Dom] # EAG.nil DO
    IF FormalName[Dom] < 0 THEN FormalName[Dom] := GetFreeVar() END;
    INC(RefCnt[FormalName[Dom]]); INC(Dom)
  END;
  IF Repetition THEN
    Dom := EAG.HNont[N].Sig; P := Node(EAG.Rep).Formal.Params;
    WHILE EAG.DomBuf[Dom] # EAG.nil DO
      ActualName[Dom] := FormalName[Dom];
      IF ~ EAG.ParamBuf[P].isDef THEN
        ActualName[Dom] := GetFreeVar(); INC(RefCnt[ActualName[Dom]])
      END;
      INC(Dom); INC(P)
    END
  END;
  A := Node.Sub;
  REPEAT
    InitComputation(A.Scope);
    IF DebugRC THEN WriteRefCnt END;
    GetFormalParamNames(N, A.Formal.Params);
    IF Repetition THEN (* GetActualParamNames for Repetition *)
      Dom := EAG.HNont[N].Sig; P := A.Actual.Params;
      WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
        IF EAG.ParamBuf[P].isDef THEN
          AffixName[P] := ActualName[Dom];
          IF EAG.ParamBuf[P].Affixform < 0 THEN
            VarName[- EAG.ParamBuf[P].Affixform] := AffixName[P]; INC(RefCnt[AffixName[P]])
          END
        END;
```

```
        INC(P); INC(Dom)
      END;
    END;
    CheckDefPos(A.Formal.Params);
    F := A.Sub;
    WHILE F # NIL DO
      IF F IS EAG.Nont THEN
        GetActualParamNames(F(EAG.Nont).Sym, F(EAG.Nont).Actual.Params);
        CheckApplPos(F(EAG.Nont).Actual.Params, FALSE);
        IF Embed & Sets.In(EAG.Prod, F(EAG.Nont).Sym) & ~ Sets.In(EAG.Pred, F(EAG.Nont).Sym)
            & (EAG.HNont[F(EAG.Nont).Sym].Id < 0) THEN
          Dom := EAG.HNont[F(EAG.Nont).Sym].Sig; P := F(EAG.Nont).Actual.Params;
          WHILE EAG.DomBuf[Dom] # EAG.nil DO
            FormalName[Dom] := AffixName[P]; INC(Dom); INC(P)
          END;
          Traverse(F(EAG.Nont).Sym)
        END;
        CheckDefPos(F(EAG.Nont).Actual.Params);
        FreeActualParamNames(F(EAG.Nont).Actual.Params)
      END;
      F := F.Next;
    END;
    IF Node IS EAG.Rep THEN
      GetActualParamNames(N, A.Actual.Params) ;
      CheckApplPos(A.Actual.Params, FALSE);
      CheckDefPos(A.Actual.Params);
      FreeActualParamNames(A.Actual.Params) ;
    END;
    CheckApplPos(A.Formal.Params, Repetition);
    IF isPred THEN FreeAllDefPosVarNames(A) END;
    FinitComputation(A.Scope);
    IF DebugRC THEN WriteRefCnt END;
    A := A.Next
  UNTIL A = NIL;
  IF Node IS EAG.Rep THEN
    InitComputation(Node(EAG.Rep).Scope);
    GetFormalParamNames(N, Node(EAG.Rep).Formal.Params);
    CheckDefPos(Node(EAG.Rep).Formal.Params);
    CheckApplPos(Node(EAG.Rep).Formal.Params, TRUE);
    FinitComputation(Node(EAG.Rep).Scope)
  ELSIF Node IS EAG.Opt THEN
    InitComputation(Node(EAG.Opt).Scope);
    GetFormalParamNames(N, Node(EAG.Opt).Formal.Params);
    CheckDefPos(Node(EAG.Opt).Formal.Params);
    CheckApplPos(Node(EAG.Opt).Formal.Params, FALSE);
    FinitComputation(Node(EAG.Opt).Scope)
  END;
  IF Repetition THEN
    P := Node(EAG.Rep).Formal.Params;
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF ~ EAG.ParamBuf[P].isDef THEN Dispose(AffixName[P]) END;
      INC(P)
    END
  END;
  Dom := EAG.HNont[N].Sig;
  WHILE EAG.DomBuf[Dom] # EAG.nil DO
    Dispose(FormalName[Dom]); INC(Dom)
  END;
  IF DebugRC THEN
    IO.WriteText(RC, "\nEnde "); EAG.WriteHNont(RC, N); IO.WriteLn(RC); WriteRefCnt
  END;
END Traverse;

PROCEDURE ComputeRepAppls(N: INTEGER);
  VAR A: EAG.Alt; P: INTEGER;
BEGIN
  IF EAG.HNont[N].Def IS EAG.Rep THEN
    A := EAG.HNont[N].Def.Sub;
    REPEAT
      P := A.Actual.Params;
      WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
        IF EAG.ParamBuf[P].isDef THEN
          RepAppls[N] := RepAppls[N] & (VarAppls[- EAG.ParamBuf[P].Affixform] = 1)
        END;
        INC(P)
      END;
      A := A.Next
    UNTIL A = NIL
  END
END ComputeRepAppls;
```

```
BEGIN (* ComputeVarNames(N: INTEGER; Embed: BOOLEAN); *)
  ASSERT(Sets.In(EAG.Prod, N), 98); ASSERT(~ Sets.In(HNontDef, N), 97);
  NEW(FreeVar, 63); NEW(RefCnt, 63);
  ComputeRepAppls(N);
  NextFreeVar := 0; Top := 0; Traverse(N); HNontVars[N] := NextFreeVar;
  Sets.Incl(HNontDef, N)
END ComputeVarNames;

    (* ------------------------- Generierungs - Prozeduren ------------------------- *)

PROCEDURE InitGen*(MOut: IO.TextOut; Treatment: INTEGER);
  VAR isSLEAG, isLEAG: BOOLEAN; i, N: INTEGER;

  PROCEDURE SetFlags(Treatment: INTEGER);
  BEGIN
    CASE Treatment OF
      parsePass: SavePos := TRUE; UseConst := FALSE; UseRefCnt := FALSE;
    | onePass:
    | sSweepPass: TraversePass := TRUE;
    END
  END SetFlags;

BEGIN
  IF Generating THEN
    IO.WriteText(IO.Msg, "\nresetting SLEAG\n"); IO.Update(IO.Msg)
  END;
  Mod := MOut;
  SavePos := FALSE;
  TraversePass := FALSE;
  UseConst := ~ IO.IsOption("c");
  UseRefCnt := ~ IO.IsOption("r");
  DebugRC := IO.IsLongOption("d", "R");
  SetFlags(Treatment);
  IF UseRefCnt THEN IO.Write(IO.Msg, "+") ELSE IO.Write(IO.Msg, "-") END; IO.WriteString(IO.Msg, "rc ");
  IF UseConst THEN IO.Write(IO.Msg, "+") ELSE IO.Write(IO.Msg, "-") END; IO.WriteString(IO.Msg, "ct ");
  IO.Update(IO.Msg);
    (* ------ Berechnungen ------ *)
  IF ~ Testing THEN PrepareInit END;
  ComputeNodeIdent;
  ComputeConstDat;
    (* ------ Variableninit ------ *)
  IF DebugRC THEN IO.CreateOut(RC, "Debug.RefCnt") END;
  NEW(AffixName, EAG.NextParam);
  FOR i := EAG.firstParam TO EAG.NextParam - 1 DO AffixName[i] := -1 END;
  NEW(NodeName, EAG.NextNode);
  NEW(VarName, EAG.NextVar); NEW(VarDeps, EAG.NextVar);
  NEW(VarRefCnt, EAG.NextVar); NEW(VarDepPos, EAG.NextVar);
  FOR i := EAG.firstVar TO EAG.NextVar - 1 DO
    VarRefCnt[i] := 0; VarDepPos[i] := -1; VarName[i] := -1
  END;
  NEW(ActualName, EAG.NextDom); NEW(FormalName, EAG.NextDom);
  FOR i := EAG.firstDom TO EAG.NextDom - 1 DO ActualName[i] := -1; FormalName[i] := -1 END;
  NEW(HNontVars, EAG.NextHNont);
  Sets.New(HNontDef, EAG.NextHNont);
  NEW(RepAppls, EAG.NextHNont);
  FOR i := EAG.firstHNont TO EAG.NextHNont - 1 DO RepAppls[i] := TRUE END;
  Sets.New(EmptySet, EAG.NextVar);
  Generating := TRUE
END InitGen;

PROCEDURE FinitGen*;
BEGIN
  IF ~ Testing THEN PrepareFinit END;
  EmptySet := NIL;
  NodeIdent := NIL;
  AffixSpace := NIL; AffixPlace := NIL; Leaf := NIL;
  AffixName := NIL; NodeName := NIL; VarName := NIL; VarDeps := NIL; VarRefCnt := NIL; VarDepPos := NIL;
  ActualName := NIL; FormalName := NIL; HNontVars := NIL; RepAppls := NIL;
  IF DebugRC THEN IO.Update(RC); IO.Show(RC) END;
  Generating := FALSE
END FinitGen;

PROCEDURE Str(s: ARRAY OF CHAR);
BEGIN IO.WriteText(Mod, s)
END Str;

PROCEDURE Int(i: LONGINT);
BEGIN IO.WriteInt(Mod, i)
END Int;
```

```
PROCEDURE GenVar(Var: INTEGER);
BEGIN
  Str("V"); Int(Var)
END GenVar;

PROCEDURE GenHeap(Var, Offset: INTEGER);
BEGIN
  Str("Heap[");
  IF Var <= 0 THEN Str("NextHeap") ELSE GenVar(Var) END;
  IF Offset > 0 THEN Str(" + "); Int(Offset)
  ELSIF Offset < 0 THEN Str(" - "); Int(- Offset)
  END;
  Str("]")
END GenHeap;

PROCEDURE GenIncRefCnt(Var, n: INTEGER);
BEGIN
  Str("INC(Heap[");
  IF Var < 0 THEN Int(- Var) ELSE GenVar(Var) END;
  Str("], ");
  IF n # 1 THEN Int(n); Str(" *") END;
  Str(" refConst); \n");
END GenIncRefCnt;

PROCEDURE GenOverflowGuard(n: INTEGER);
BEGIN
  IF n > 0 THEN
    Str("IF NextHeap >= LEN(Heap^) - "); Int(n); Str(" THEN EvalExpand END; \n")
  END
END GenOverflowGuard;

PROCEDURE GenFreeHeap(Var: INTEGER);
BEGIN
  Str("FreeHeap("); GenVar(Var); Str("); \n")
END GenFreeHeap;

PROCEDURE GenHeapInc(n: INTEGER);
BEGIN
  IF n # 0 THEN
    IF n = 1 THEN
      Str("INC(NextHeap); \n")
    ELSE
      Str("INC(NextHeap, "); Int(n); Str("); \n")
    END
  END
END GenHeapInc;

PROCEDURE GenDeclarations*;
  VAR
    Fix : IO.TextIn;
    Name : ARRAY EAG.BaseNameLen + 10 OF CHAR;
    OpenError : BOOLEAN;
    TabTimeStamp : LONGINT;

  PROCEDURE Append(VAR Dest : ARRAY OF CHAR; Src, Suf : ARRAY OF CHAR);
    VAR i, j : INTEGER;
  BEGIN
    i := 0; j := 0;
    WHILE (Src[i] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Src[i]; INC(i) END;
    WHILE (Suf[j] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Suf[j]; INC(i); INC(j) END;
    Dest[i] := 0X
  END Append;

  PROCEDURE InclFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = 0X THEN
        IO.WriteText(IO.Msg, "\n\terror: unexpected end of eSLEAGGen.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Write(Mod, c); IO.Read(Fix, c)
    END
  END InclFix;

  PROCEDURE SkipFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
```

```
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = OX THEN
        IO.WriteText(IO.Msg, "\n\terror: unexpected end of eSLEAGGen.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Read(Fix, c)
    END
  END SkipFix;

  PROCEDURE GenTabFile(TabTimeStamp: LONGINT);
    CONST errVal = 0; magic = 6C617645H;
    VAR i, P, Next, Start: INTEGER; Tab: IO.File;
      Heap: POINTER TO ARRAY OF INTEGER;

    PROCEDURE SynTree(Node: INTEGER; VAR Next: INTEGER);
      VAR n, Node1, Next1, Len1: INTEGER;
    BEGIN
      Heap[Next] := NodeIdent[EAG.NodeBuf[Node]];
      Next1 := Next; INC(Next, 1 + EAG.MAlt[EAG.NodeBuf[Node]].Arity);
      FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
        Node1 := EAG.NodeBuf[Node + n];
        IF EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0 THEN
          Heap[Next1 + n] := Leaf[EAG.NodeBuf[Node1]]
        ELSE
          Heap[Next1 + n] := Next;
          SynTree(Node1, Next)
        END
      END
    END SynTree;

  BEGIN (* GenTabFile(TabTimeStamp: LONGINT); *)
    IO.CreateFile(Tab, Name);
    IO.PutLInt(Tab, magic);
    IO.PutLInt(Tab, TabTimeStamp);
    IO.PutLInt(Tab, FirstHeap - 1); (* predefined *)
    NEW(Heap, FirstHeap);
    Heap[errVal] := 0;         (* Arity[errVal] = 0, errVal describes no existing MAlt *)
    FOR i := 1 TO EAG.MaxMArity DO Heap[i] := errVal END;
    IF UseConst THEN
      FOR i := EAG.firstMAlt TO EAG.NextMAlt - 1 DO
        IF Leaf[i] >= errVal THEN Heap[Leaf[i]] := NodeIdent[i] END
      END;
      FOR P := EAG.firstParam TO EAG.NextParam - 1 DO
        IF (EAG.ParamBuf[P].Affixform # EAG.nil) & (AffixPlace[P] >= 0) THEN
          Next := AffixPlace[P]; SynTree(EAG.ParamBuf[P].Affixform, Next)
        END
      END
    END;
    FOR i := 0 TO FirstHeap - 1 DO IO.PutLInt(Tab, Heap[i]) END;
    IO.PutLInt(Tab, TabTimeStamp);
    IO.CloseFile(Tab);
  END GenTabFile;

BEGIN (* GenDeclarations *)
  IF TraversePass THEN
    Append(Name, EAG.BaseName, "Eval.EvalTab")
  ELSE
    Append(Name, EAG.BaseName, ".EvalTab")
  END;
  TabTimeStamp := IO.TimeStamp();
  IO.OpenIn(Fix, "eSLEAGGen.Fix", OpenError);
  IF OpenError THEN
    IO.WriteText(IO.Msg, "\n\terror: could not open eELL1Gen.Fix\n"); IO.Update(IO.Msg); HALT(99)
  END;
  InclFix("$"); Int(FirstHeap - 1);                                  (* predefined *)
  InclFix("$"); Int(ArityConst);                                     (* arityConst *)
  InclFix("$");                                                      (* HeapType *)
    IF SavePos THEN Str("Eval.TreeType") ELSE Str("LONGINT") END;
  InclFix("$");                                                      (* OpenHeap *)
    IF SavePos THEN Str("Eval.OpenTree")
    ELSE Str("POINTER TO ARRAY OF HeapType")
    END;
  InclFix("$");                                                      (* PosTypes *)
    IF SavePos THEN InclFix("$") ELSE SkipFix("$") END;
  InclFix("$"); Int(EAG.MaxMArity + 1);                              (* maxArity *)
  InclFix("$"); Int(RefConst);                                       (* refConst *)
  InclFix("$");                                                      (* EvalExpand *)
    IF SavePos THEN SkipFix("$"); InclFix("$") ELSE InclFix("$"); SkipFix("$") END;
  IF UseRefCnt THEN InclFix("$"); SkipFix("$") ELSE SkipFix("$"); InclFix("$") END;   (* RefCnt-Procs *)
```

```
  InclFix("$"); IF ~ TraversePass THEN Str("S.") END;                  (* Error-position *)
  InclFix("$"); IF UseRefCnt THEN InclFix("$") ELSE SkipFix("$") END;  (* AnalyseError *)
  InclFix("$"); Str(Name);                                             (* EvalTabName *)
  InclFix("$"); Int(TabTimeStamp);                                     (* tabTimeStamp *)
  InclFix("$"); IF SavePos THEN InclFix("$") ELSE SkipFix("$") END;    (* PosInit *)
  InclFix("$");
  GenTabFile(TabTimeStamp);
  IO.CloseIn(Fix)
END GenDeclarations;

PROCEDURE PosNeeded*(P: INTEGER): BOOLEAN;
  VAR V: INTEGER;
BEGIN
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF EAG.ParamBuf[P].isDef THEN
      V := - EAG.ParamBuf[P].Affixform;
      IF V < 0 THEN RETURN TRUE
      ELSIF EAG.Var[V].Def THEN RETURN TRUE
      ELSIF EAG.Var[EAG.Var[V].Neg].Def THEN RETURN TRUE
      END
    END;
    INC(P)
  END;
  RETURN FALSE
END PosNeeded;

PROCEDURE GenAnalPred*(Sym, P: INTEGER);
  VAR Node, Tree, n, V, Vn: INTEGER; MakeRefCnt, IsPred: BOOLEAN;

  PROCEDURE Comp;
  BEGIN
    IF UseRefCnt THEN Str(" MOD refConst") END;
    IF IsPred THEN Str(" = ") ELSE Str(" # ") END
  END Comp;

  PROCEDURE GenEqualErrMsg(Sym, Var: INTEGER);
  BEGIN
    Str("\''"); EAG.WriteVar(Mod, Var); Str("' failed in '"); EAG.WriteNamedHNont(Mod, Sym); Str("'\'")
  END GenEqualErrMsg;

  PROCEDURE GenAnalErrMsg(Sym: INTEGER);
  BEGIN
    Str('"'); EAG.WriteNamedHNont(Mod, Sym); Str('"');
  END GenAnalErrMsg;

  PROCEDURE GenEqualPred(VarName1, Var2: INTEGER; Eq: BOOLEAN);
  BEGIN
    IF IsPred THEN
      Str("IF "); IF ~ Eq THEN Str(" ~ ") END; Str("Equal("); GenVar(VarName1); Str(", ");
      GenVar(VarName[Var2]); Str(") THEN \n"); INC(IfLevel)
    ELSE
      IF ~ Eq THEN Str("Un") END; Str("Eq("); GenVar(VarName1);
      Str(", "); GenVar(VarName[Var2]); Str(", "); GenEqualErrMsg(Sym, Var2); Str("); ")
    END;
  END GenEqualPred;

  PROCEDURE GenAnalTree(Node: INTEGER);
    VAR n, Node1, V, Vn: INTEGER;
  BEGIN
    Str("IF ");
    (*IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node]].Arity = 0) THEN
      GenVar(NodeName[Node]); IF IsPred THEN Str(" = ") ELSE Str(" # ") END; Int(Leaf[EAG.NodeBuf[Node]])
    ELSE
      GenHeap(NodeName[Node], 0); Comp; Int(NodeIdent[EAG.NodeBuf[Node]])
    END; *)
    GenHeap(NodeName[Node], 0); Comp; Int(NodeIdent[EAG.NodeBuf[Node]]);
    Str(" THEN ");
    IF IsPred THEN Str("\n"); INC(IfLevel)
    ELSE
      Str("AnalyseError("); GenVar(NodeName[Node]); Str(", ");
      GenAnalErrMsg(Sym); Str(") END; \n")
    END;
    FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
      Node1 := EAG.NodeBuf[Node + n];
      IF Node1 < 0 THEN V := - Node1;
        IF EAG.Var[V].Def THEN
          IF IsPred THEN
            Str("IF Equal("); GenHeap(NodeName[Node], n);
            Str(", "); GenVar(VarName[V]); Str(") THEN \n");
            INC(IfLevel)
```

```
          ELSE
            Str("Eq("); GenHeap(NodeName[Node], n);
            Str(", "); GenVar(VarName[V]); Str(", "); GenEqualErrMsg(Sym, V); Str("); ");
          END;
        ELSE EAG.Var[V].Def := TRUE;
          GenVar(VarName[V]); Str(" := "); GenHeap(NodeName[Node], n); Str("; ");
          IF EAG.Var[EAG.Var[V].Neg].Def THEN Vn := EAG.Var[V].Neg;
            GenEqualPred(VarName[Vn], V, FALSE);
            IF MakeRefCnt THEN
              IF VarDepPos[Vn] = P THEN GenFreeHeap(VarName[Vn]); VarDepPos[Vn] := -2 END
            END
          END;
          IF MakeRefCnt & (VarRefCnt[V] > 0) THEN
            GenIncRefCnt(VarName[V], VarRefCnt[V])
          END
        END;
        IF MakeRefCnt THEN
          IF VarDepPos[V] = P THEN GenFreeHeap(VarName[V]); VarDepPos[V] := -2 END
        END
      ELSE
        IF EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0 THEN
          IF UseConst THEN
            Str("IF "); GenHeap(NodeName[Node], n); Comp; Int(Leaf[EAG.NodeBuf[Node1]])
          ELSE
            Str("IF Heap["); GenHeap(NodeName[Node], n); Str("]");
            Comp; Int(NodeIdent[EAG.NodeBuf[Node1]])
          END; Str(" THEN ");
          IF IsPred THEN INC(IfLevel)
          ELSE
            IO.WriteString(Mod, "AnalyseError("); GenHeap(NodeName[Node], n); Str(", ");
            GenAnalErrMsg(Sym); IO.WriteString(Mod, ") END; ")
          END; Str("\n")
        ELSE
          GenVar(NodeName[Node1]); Str(" := "); GenHeap(NodeName[Node], n); Str("; ");
          GenAnalTree(Node1)
        END
      END
    END
  END GenAnalTree;

BEGIN (* GenAnalPred*(Sym, P: INTEGER) *)
  IsPred := Sets.In(EAG.Pred, Sym); IfLevel := 0;
  MakeRefCnt := UseRefCnt & ~ IsPred;
  WHILE (EAG.ParamBuf[P].Affixform # EAG.nil) DO
    IF EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF Tree < 0 THEN V := - Tree;
        IF EAG.Var[V].Def THEN ASSERT(AffixName[P] # VarName[V]);
          GenEqualPred(AffixName[P], V, TRUE);
          IF MakeRefCnt THEN GenFreeHeap(AffixName[P]) END
        ELSE EAG.Var[V].Def := TRUE;
          IF AffixName[P] # VarName[V] THEN
            GenVar(VarName[V]); Str(" := "); GenVar(AffixName[P]); Str("; \n")
          END;
          IF EAG.Var[EAG.Var[V].Neg].Def THEN Vn := EAG.Var[V].Neg;
            GenEqualPred(VarName[Vn], V, FALSE);
            IF MakeRefCnt THEN
              IF VarDepPos[Vn] = P THEN GenFreeHeap(VarName[Vn]); VarDepPos[Vn] := -2 END
            END
          END;
          IF MakeRefCnt THEN
            IF VarRefCnt[V] > 1 THEN
              GenIncRefCnt(VarName[V], VarRefCnt[V] - 1)
            ELSIF VarRefCnt[V] = 0 THEN
              GenFreeHeap(AffixName[P])
            END
          END
        END;
        IF MakeRefCnt THEN
          IF VarDepPos[V] = P THEN GenFreeHeap(VarName[V]); VarDepPos[V] := -2 END
        END
      ELSE
        IF EAG.MAlt[EAG.NodeBuf[Tree]].Arity = 0 THEN
          Str("IF ");
          GenHeap(AffixName[P], 0); Comp; IO.WriteInt(Mod, NodeIdent[EAG.NodeBuf[Tree]]);
          Str(" THEN ");
          IF IsPred THEN INC(IfLevel);
          ELSE
            Str("AnalyseError("); GenVar(AffixName[P]); Str(", "); GenAnalErrMsg(Sym); Str(") END; ")
          END; Str("\n")
```

```
        ELSE
          GenAnalTree(Tree)
        END;
        IF MakeRefCnt THEN GenFreeHeap(AffixName[P]) END
      END
    END;
    INC(P)
  END;
  IF SavePos THEN Str("PushPos; \n") END
END GenAnalPred;

PROCEDURE GenSynTree(Node: INTEGER; RepVar: Sets.OpenSet; VAR Next: INTEGER);
  (* RepVar ist nur im Kontext der Generierung von Repetition-Code zu verstehen *)
  VAR n, Next1, Node1, V, Alt: INTEGER;
BEGIN
  Alt := EAG.NodeBuf[Node];
  GenHeap(0, Next); Str(" := "); Int(NodeIdent[Alt]); Str("; ");
  Next1 := Next; INC(Next, 1+ EAG.MAlt[Alt].Arity);
  FOR n := 1 TO EAG.MAlt[Alt].Arity DO
    Node1 := EAG.NodeBuf[Node + n];
    IF Node1 < 0 THEN V := - Node1;
      IF Sets.In(RepVar, V) THEN
        GenVar(VarName[V]); Str(" := NextHeap + "); Int(Next1 + n)
      ELSE
        GenHeap(0, Next1 + n); Str(" := "); GenVar(VarName[V])
      END;
      Str("; ")
    ELSE
      GenHeap(0, Next1 + n); Str(" := ");
      IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0) THEN
        Int(Leaf[EAG.NodeBuf[Node1]]); Str("; ")
      ELSE
        Str("NextHeap + "); Int(Next); Str("; ");
        GenSynTree(Node1, RepVar, Next)
      END
    END
  END
END GenSynTree;

PROCEDURE Gen1SynTree(Node: INTEGER; RepVar: Sets.OpenSet; IsPred: BOOLEAN);
  (* RepVar ist nur im Kontext der Generierung von Repetition-Code zu verstehen *)
  VAR n, Node1, V: INTEGER;
BEGIN
  GenHeap(NodeName[Node], 0); Str(" := "); Int(NodeIdent[EAG.NodeBuf[Node]]); Str("; ");
  FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
    Node1 := EAG.NodeBuf[Node + n];
    IF Node1 < 0 THEN V := - Node1;
      IF Sets.In(RepVar, V) THEN
        GenVar(VarName[V]); Str(" := "); GenVar(NodeName[Node]); Str(" + "); Int(n); Str("; ")
      ELSE
        GenHeap(NodeName[Node], n); Str(" := "); GenVar(VarName[V]); Str("; ");
        IF IsPred THEN GenIncRefCnt(VarName[V], 1) END
      END
    ELSIF EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0 THEN
      IF UseConst THEN
        GenHeap(NodeName[Node], n); Str(" := "); Int(Leaf[EAG.NodeBuf[Node1]]); Str("; ");
        GenIncRefCnt(- Leaf[EAG.NodeBuf[Node1]], 1)
      ELSE
        Str("GetHeap(0, "); GenHeap(NodeName[Node], n);
        Str("); Heap["); GenHeap(NodeName[Node], n); Str("] := ");
        Int(NodeIdent[EAG.NodeBuf[Node1]]); Str("; ")
      END
    ELSE
      Str("GetHeap("); Int(EAG.MAlt[EAG.NodeBuf[Node1]].Arity); Str(", ");
      IF NodeName[Node] = NodeName[Node1] THEN
        GenHeap(NodeName[Node], n); Str("); ");
        GenVar(NodeName[Node1]); Str(" := "); GenHeap(NodeName[Node], n);
      ELSE
        GenVar(NodeName[Node1]); Str("); ");
        GenHeap(NodeName[Node], n); Str(" := "); GenVar(NodeName[Node1]);
      END;
      Str("; \n"); Gen1SynTree(Node1, RepVar, IsPred)
    END
  END
END Gen1SynTree;

PROCEDURE GetAffixSpace(P: INTEGER);
  VAR Heap: INTEGER;
BEGIN Heap := 0;
  WHILE (EAG.ParamBuf[P].Affixform # EAG.nil) DO
```

```
    IF ~ EAG.ParamBuf[P].isDef & (~ UseConst OR (UseConst & (AffixPlace[P] < 0))) THEN
      INC(Heap, AffixSpace[P])
    END;
    INC(P)
  END;
  GenOverflowGuard(Heap)
END GetAffixSpace;

PROCEDURE GenSynPred*(Sym, P: INTEGER);
  VAR Next, Tree, n, V: INTEGER; IsPred: BOOLEAN;
BEGIN
  IsPred := Sets.In(EAG.Pred, Sym);
  IF ~ UseRefCnt THEN GetAffixSpace(P) END;
  WHILE (EAG.ParamBuf[P].Affixform # EAG.nil) DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF SavePos THEN
        Str("PopPos("); Int(EAG.MAlt[EAG.NodeBuf[Tree]].Arity); Str("); \n")
      END;
      IF UseConst & (AffixPlace[P] >= 0) THEN
        GenVar(AffixName[P]); Str(" := "); Int(AffixPlace[P]); Str("; \n");
        IF UseRefCnt THEN GenIncRefCnt(- AffixPlace[P], 1) END
      ELSIF Tree < 0 THEN V := - Tree;
        IF UseRefCnt & IsPred THEN GenIncRefCnt(VarName[V], 1) END;
        IF AffixName[P] # VarName[V] THEN
          GenVar(AffixName[P]); Str(" := "); GenVar(VarName[V]); Str(";\n")
        END;
      ELSE
        IF UseRefCnt THEN
          Str("GetHeap("); Int(EAG.MAlt[EAG.NodeBuf[Tree]].Arity);
          Str(", "); GenVar(NodeName[Tree]); Str("); ");
          Gen1SynTree(Tree, EmptySet, IsPred); Str("\n")
        ELSE
          GenVar(AffixName[P]); Str(" := NextHeap; ");
          Next := 0; GenSynTree(Tree, EmptySet, Next); GenHeapInc(Next)
        END
      END
    END;
    INC(P)
  END
END GenSynPred;

PROCEDURE GenRepStart*(Sym: INTEGER);
  VAR P, Dom, Next: INTEGER;
BEGIN
  IF ~ UseRefCnt THEN
    Next := 0; P := EAG.HNont[Sym].Def(EAG.Rep).Sub.Formal.Params;
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF ~ EAG.ParamBuf[P].isDef THEN INC(Next) END; INC(P)
    END;
    GenOverflowGuard(Next)
  END;
  Dom := EAG.HNont[Sym].Sig; P := EAG.HNont[Sym].Def(EAG.Rep).Sub.Formal.Params;
  WHILE EAG.DomBuf[Dom] # EAG.nil DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      IF UseRefCnt THEN
        Str("GetHeap(0, "); GenVar(FormalName[Dom]); Str("); ")
      ELSE
        GenVar(FormalName[Dom]); Str(" := NextHeap; INC(NextHeap); ")
      END;
      GenVar(AffixName[P]); Str(" := "); GenVar(FormalName[Dom]); Str("; \n")
    END;
    INC(P); INC(Dom)
  END
END GenRepStart;

PROCEDURE GenHangIn(P: INTEGER; Guard: BOOLEAN);
  VAR Tree, Next: INTEGER;

  PROCEDURE FreeVariables(Node: INTEGER);
    VAR n: INTEGER;
  BEGIN
    IF Node < 0 THEN
      IF ~ Sets.In(RepVar, - Node) THEN Str("FreeHeap("); GenVar(VarName[- Node]); Str("); ") END
    ELSE
      FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO
        FreeVariables(EAG.NodeBuf[Node + n])
      END
    END
  END FreeVariables;
```

```
BEGIN (* GenHangIn(P: INTEGER; Guard: BOOLEAN);  *)
  Next := 0;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF Guard THEN Str("IF "); GenVar(AffixName[P]); Str(" # undef THEN \n") END;
      IF UseConst & (AffixPlace[P] >= 0) THEN
        GenHeap(AffixName[P], 0); Str(" := "); Int(AffixPlace[P]); Str("; \n");
        IF UseRefCnt THEN GenIncRefCnt(- AffixPlace[P], 1) END
      ELSIF Tree < 0 THEN
        IF AffixName[P] # VarName[- Tree] THEN
          GenHeap(AffixName[P], 0); Str(" := "); GenVar(VarName[- Tree]); Str("; \n");
          IF Guard THEN
            Str("ELSE FreeHeap("); GenVar(VarName[- Tree]); Str(") \n")
          END
        END
      ELSE
        IF UseRefCnt THEN
          Str("GetHeap("); Int(EAG.MAlt[EAG.NodeBuf[Tree]].Arity);
          Str(", "); GenVar(NodeName[Tree]); Str("); ");
          GenHeap(AffixName[P], 0); Str(" := "); GenVar(NodeName[Tree]); Str("; \n");
          IF Guard THEN Str("ELSE "); FreeVariables(Tree) END
        ELSE
          GenHeap(AffixName[P], 0);  Str(" := NextHeap");
          IF Next # 0 THEN Str(" + "); Int(Next) END; Str("; \n");
          INC(Next, AffixSpace[P]);
          IF Guard THEN Str("ELSE ") END
        END;
        IF Guard THEN GenVar(NodeName[Tree]); Str(" := undef; \n") END
      END;
      IF Guard THEN Str("END; \n") END
    END;
    INC(P)
  END
END GenHangIn;

PROCEDURE GenRepAlt*(Sym: INTEGER; A: EAG.Alt);
  VAR P, P1, Dom, Tree, Next: INTEGER; Guard: BOOLEAN;
BEGIN
  Guard := ~ RepAppls[Sym];
  GenSynPred(Sym, A.Actual.Params);
  IF SavePos THEN Str("PushPos; \n") END;
  P := A.Actual.Params; Dom := EAG.HNont[Sym].Sig;
  WHILE (EAG.ParamBuf[P].Affixform # EAG.nil) DO
    IF ~ EAG.ParamBuf[P].isDef & (AffixName[P] # FormalName[Dom]) THEN
      GenVar(FormalName[Dom]); Str(" := "); GenVar(AffixName[P]); Str("; \n")
    END;
    INC(P); INC(Dom)
  END;
  P1 := A.Actual.Params; Dom := EAG.HNont[Sym].Sig;
  P := A.Formal.Params;
  IF ~ UseRefCnt THEN GetAffixSpace(P) END;
  GenHangIn(P, Guard); Next := 0;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF SavePos THEN
        Str("PopPos("); Int(EAG.MAlt[EAG.NodeBuf[Tree]].Arity); Str("); \n")
      END;
      IF (Tree > 0) & ~ (UseConst & (AffixPlace[P] >= 0)) THEN
        IF Guard THEN Str("IF "); GenVar(NodeName[Tree]); Str(" # undef THEN \n") END;
        IF UseRefCnt THEN
          Gen1SynTree(Tree, RepVar, Sets.In(EAG.Pred, Sym))
        ELSE
          GenSynTree(Tree, RepVar, Next)
        END;
        Str("\n");
        IF Guard THEN Str("END; \n") END
      END;
      IF Guard & (VarAppls[- EAG.ParamBuf[P1].Affixform] = 0) THEN
        GenVar(AffixName[P1]); Str(" := undef; \n")
      END
    END;
    INC(P); INC(P1); INC(Dom)
  END;
  IF ~ UseRefCnt THEN GenHeapInc(Next) END
END GenRepAlt;

PROCEDURE GenRepEnd*(Sym: INTEGER);
```

```
  VAR P, P1, Dom, Tree, Next: INTEGER; Guard: BOOLEAN;
BEGIN
  InitScope(EAG.HNont[Sym].Def(EAG.Rep).Scope);
  P := EAG.HNont[Sym].Def(EAG.Rep).Formal.Params;
  P1 := EAG.HNont[Sym].Def.Sub.Actual.Params;
  Dom := EAG.HNont[Sym].Sig;
  GenAnalPred(Sym, P);
  IF ~ UseRefCnt THEN GetAffixSpace(P) END;
  GenHangIn(P, Guard); Next := 0;
  WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
    IF ~ EAG.ParamBuf[P].isDef THEN
      Tree := EAG.ParamBuf[P].Affixform;
      IF SavePos THEN
        Str("PopPos("); Int(EAG.MAlt[EAG.NodeBuf[Tree]].Arity); Str("); \n")
      END;
      IF (Tree > 0) & ~ (UseConst & (AffixPlace[P] >= 0)) THEN
        IF Guard THEN Str("IF "); GenVar(NodeName[Tree]); Str(" # undef THEN \n") END;
        IF UseRefCnt THEN
          Gen1SynTree(Tree, EmptySet, Sets.In(EAG.Pred, Sym))
        ELSE
          GenSynTree(Tree, EmptySet, Next)
        END;
        Str("\n");
        IF Guard THEN Str("END; \n") END
      END;
      IF UseRefCnt THEN
        GenVar(AffixName[P]); Str(" := "); GenVar(FormalName[Dom]); Str("; ")
      END;
      GenVar(FormalName[Dom]); Str(" := "); GenHeap(FormalName[Dom], 0); Str("; \n");
      IF UseRefCnt THEN
        GenHeap(AffixName[P], 0); Str(" := 0; FreeHeap("); GenVar(AffixName[P]); Str("); \n")
      END
    END;
    INC(P); INC(P1); INC(Dom)
  END;
  IF ~ UseRefCnt THEN GenHeapInc(Next) END
END GenRepEnd;

PROCEDURE GenFormalParams*(N: INTEGER; ParNeeded: BOOLEAN);
  VAR Dom, i: INTEGER;
BEGIN
  Dom := EAG.HNont[N].Sig; i := 1;
  IF ParNeeded THEN Str("(") END;
  IF EAG.DomBuf[Dom] # EAG.nil THEN
    IF ~ ParNeeded THEN Str("; ") END;
    LOOP
      IF EAG.DomBuf[Dom] > 0 THEN Str("VAR ") END;
      GenVar(i); Str(": HeapType"); INC(i);
      INC(Dom);
      IF EAG.DomBuf[Dom] = EAG.nil THEN EXIT END;
      Str("; ")
    END;
  END;
  IF ParNeeded THEN
    Str(")"); IF Sets.In(EAG.Pred, N) THEN Str(": BOOLEAN") END
  END
END GenFormalParams;

PROCEDURE GenVarDecl*(N: INTEGER);
  VAR Dom, FormalVars, i: INTEGER;
BEGIN
  Dom := EAG.HNont[N].Sig; FormalVars := 1;
  WHILE EAG.DomBuf[Dom] # EAG.nil DO INC(Dom); INC(FormalVars) END;
  IF HNontVars[N] - FormalVars >= 0 THEN
    Str("\tVAR ");
    FOR i := FormalVars TO HNontVars[N]  DO
      IF i # FormalVars THEN Str(", ") END;
      GenVar(i)
    END;
    Str(": HeapType;\n")
  END;
  IF Sets.In(EAG.Pred, N) THEN Str("\tVAR Failed: BOOLEAN; \n") END
END GenVarDecl;

PROCEDURE GenActualParams*(P: INTEGER; ParNeeded: BOOLEAN);
BEGIN
  IF ParNeeded THEN Str("(") END;
  IF EAG.ParamBuf[P].Affixform # EAG.nil THEN
    IF ~ ParNeeded THEN Str(", ") END;
    LOOP
```

```
      ASSERT(AffixName[P] >= 0, 89); GenVar(AffixName[P]);
      INC(P);
      IF EAG.ParamBuf[P].Affixform = EAG.nil THEN EXIT END;
      Str(", ")
    END;
  END;
  IF ParNeeded THEN Str(")") END
END GenActualParams;

    (* ---------------------- Prdikat-Prozeduren ---------------------- *)

PROCEDURE GenPredProcs*;
  VAR N: INTEGER;

  PROCEDURE GenForward(N: INTEGER);

    PROCEDURE GenPredCover(N: INTEGER);
      VAR Dom, i: INTEGER;
    BEGIN ASSERT(Sets.In(EAG.Pred, N), 98);
      Str("PROCEDURE Check"); Int(N);
      Str("(ErrMsg: ARRAY OF CHAR"); GenFormalParams(N, FALSE);
      Str("); \nBEGIN\n"); Str("\tIF ~ Pred");
      Int(N); Str("(");
      Dom := EAG.HNont[N].Sig; i := 1;
      IF EAG.DomBuf[Dom] # EAG.nil THEN
        LOOP
          GenVar(i); INC(Dom); INC(i);
          IF EAG.DomBuf[Dom] = EAG.nil THEN EXIT END;
          Str(", ")
        END;
      END;
      Str(") THEN ");
      Dom := EAG.HNont[N].Sig; i := 1;
      WHILE EAG.DomBuf[Dom] >0 DO INC(Dom) END;
      IF EAG.DomBuf[Dom] # EAG.nil THEN
        Str("IF (");
        LOOP
          GenVar(i); Str(" # errVal) ");
          REPEAT INC(Dom); INC(i) UNTIL EAG.DomBuf[Dom] <=0;
          IF EAG.DomBuf[Dom] = EAG.nil THEN EXIT END;
          Str("& (")
        END;
        Str("THEN PredError(ErrMsg) END ")
      ELSE
        Str("Error(ErrMsg) ")
      END;
      Str("END \n");
      Str("END Check"); Int(N); Str(";\n\n")
    END GenPredCover;

  BEGIN (* GenForward(N: INTEGER); *)
    Str("PROCEDURE ^ Pred"); Int(N);
    GenFormalParams(N, TRUE);
    Str("; (* "); EAG.WriteHNont(Mod, N); Str(" *) \n\n");
    GenPredCover(N)
  END GenForward;

  PROCEDURE GenPredicateCode(N: INTEGER);
    VAR
      Node: EAG.Rule; A: EAG.Alt; Scope: EAG.ScopeDesc;
      P, Level, AltLevel, i: INTEGER;

    PROCEDURE CleanLevel(Level: INTEGER);
      VAR i: INTEGER;
    BEGIN
      IF Level >= 1 THEN
        FOR i := 0 TO Level - 1 DO Str("END ") END; Str("; \n")
      END
    END CleanLevel;

    PROCEDURE FreeParamTrees(P: INTEGER);
    BEGIN
      WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
        IF EAG.ParamBuf[P].isDef THEN GenFreeHeap(AffixName[P]) END;
        INC(P)
      END;
    END FreeParamTrees;

    PROCEDURE TraverseFactor(F: EAG.Factor; FormalParams: INTEGER);
      VAR Level: INTEGER;
```

```
    BEGIN
      IF F # NIL THEN
        ASSERT(F IS EAG.Nont, 99); ASSERT(Sets.In(EAG.Pred, F(EAG.Nont).Sym), 98);
        GenSynPred(N, F(EAG.Nont).Actual.Params);
        Str("\tIF Pred"); Int(F(EAG.Nont).Sym);
        GenActualParams(F(EAG.Nont).Actual.Params, TRUE); Str(" THEN (* ");
        EAG.WriteHNont(Mod, F(EAG.Nont).Sym); Str(" *) \n");
        GenAnalPred(N, F(EAG.Nont).Actual.Params); Level := IfLevel;
        TraverseFactor(F.Next, FormalParams);
        CleanLevel(Level);
        Str(" END; (* "); EAG.WriteHNont(Mod, F(EAG.Nont).Sym);
        Str(" *) \n");
        IF UseRefCnt THEN FreeParamTrees(F(EAG.Nont).Actual.Params) END;
      ELSE
        IF Node IS EAG.Rep THEN
          GenSynPred(N, A.Actual.Params);
          Str("\tIF Pred"); Int(N);
          GenActualParams(A.Actual.Params, TRUE); Str(" THEN (* ");
          EAG.WriteHNont(Mod, N); Str(" *) \n");
          GenAnalPred(N, A.Actual.Params); Level := IfLevel;
          GenSynPred(N, FormalParams);
          Str("Failed := FALSE; \n");
          CleanLevel(Level);
          Str(" END; (* "); EAG.WriteHNont(Mod, N); Str(" *) \n");
          IF UseRefCnt THEN FreeParamTrees(A.Actual.Params) END;
        ELSE
          GenSynPred(N, FormalParams);
          Str("Failed := FALSE; \n")
        END
      END
    END TraverseFactor;

  BEGIN (* GenPredicateCode *)
    Node := EAG.HNont[N].Def; AltLevel := 0;
    Str("\tFailed := TRUE; \n");
    IF (Node IS EAG.Rep) OR (Node IS EAG.Opt) THEN
      IF Node IS EAG.Opt THEN
        P := Node(EAG.Opt).Formal.Params; Scope := Node(EAG.Opt).Scope
      ELSE
        P := Node(EAG.Rep).Formal.Params; Scope := Node(EAG.Rep).Scope
      END;
      InitScope(Scope);
      GenAnalPred(N, P); Level := IfLevel;
      GenSynPred(N, P);
      Str("Failed := FALSE; \n");
      CleanLevel(Level); INC(AltLevel)
    END;
    A := Node.Sub;
    LOOP
      IF AltLevel > 0 THEN Str("IF Failed THEN (* "); Int(AltLevel+1); Str(". Alternative *) \n") END;
      InitScope(A.Scope);
      GenAnalPred(N, A.Formal.Params); Level := IfLevel;
      TraverseFactor(A.Sub, A.Formal.Params);
      CleanLevel(Level); INC(AltLevel);
      A := A.Next;
      IF A = NIL THEN EXIT END;
    END;
    FOR i := 1 TO AltLevel - 1 DO Str("END ") END; Str("; \n");
    P := Node.Sub.Formal.Params;
    IF UseRefCnt THEN FreeParamTrees(P) END;
    Str("IF Failed THEN ");
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      IF ~ EAG.ParamBuf[P].isDef THEN
        GenVar(AffixName[P]); Str(" := errVal; ");
        IF UseRefCnt THEN Str("INC(Heap[errVal], refConst); ") END
      END;
      INC(P)
    END;
    Str(" END; \n")
  END GenPredicateCode;

BEGIN (* GenPredProcs; *)
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(EAG.Pred, N) THEN GenForward(N) END
  END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(EAG.Pred, N) THEN
      ComputeVarNames(N, FALSE);
      Str("PROCEDURE Pred"); Int(N);
      GenFormalParams(N, TRUE); Str("; (* ");
```

```
        EAG.WriteHNont(Mod, N); Str(" *) \n");
        GenVarDecl(N);
        Str("BEGIN \n");
        GenPredicateCode(N);
        Str("RETURN ~ Failed\n");
        Str("END Pred"); Int(N); Str("; \n\n")
      END
    END
  END GenPredProcs;

  PROCEDURE GenPredCall*(N, ActualParams: INTEGER);
  BEGIN
    ASSERT(Sets.In(EAG.Pred, N), 90);
    Str("\tCheck"); Int(N); Str("(\'");
    IF EAG.HNont[N].Id < 0 THEN Str("in ") END; Str("'");
    EAG.WriteNamedHNont(Mod, N); Str("'\'");
    GenActualParams(ActualParams, FALSE); Str("); \n")
  END GenPredCall;

BEGIN
  Testing := FALSE; Generating := FALSE
END eSLEAGGen.
```

### 6.3.2 eSLEAGGen.Fix

```
(* ----------- eSLEAGGen.Fix Version 1.02 --  4.12.96 --------- *)
  CONST
    errVal = 0;
    predefined = $;
    arityConst = $;
    undef = - 1;
    initialHeapSize = 8192;
  TYPE
    HeapType = $;
    OpenHeap = $;
  VAR
    Heap: OpenHeap; NextHeap: HeapType;
    OutputSize: LONGINT;

  $TYPE
    OpenPos = Eval.OpenPos;
  VAR
    PosHeap, PosStack: OpenPos; PosTop: LONGINT;

  $CONST
    maxArity = $;
    refConst = $;
  VAR
    FreeList: ARRAY maxArity OF HeapType;

$  PROCEDURE EvalExpand;
    VAR Heap1: OpenHeap; i: LONGINT;
  BEGIN
    NEW(Heap1, 2 * LEN(Heap^));
    FOR i := 0 TO LEN(Heap^) - 1 DO Heap1[i] := Heap[i] END;
    Heap := Heap1
  END EvalExpand;

  PROCEDURE Reset*;
  BEGIN
    Heap := NIL
  END Reset;

$  PROCEDURE EvalExpand;
    VAR Heap1: OpenHeap; PosHeap1: OpenPos; i: LONGINT;
  BEGIN
    NEW(Heap1, 2 * LEN(Heap^)); NEW(PosHeap1, 2 * LEN(Heap^));
    FOR i := 0 TO LEN(Heap^) - 1 DO Heap1[i] := Heap[i]; PosHeap1[i] := PosHeap[i] END;
    Heap := Heap1; PosHeap := PosHeap1
  END EvalExpand;

  PROCEDURE Reset*;
  BEGIN
    Heap := NIL; PosHeap := NIL; Eval.Reset
  END Reset;

  PROCEDURE PushPos;

    PROCEDURE PosExpand;
      VAR PosStack1: OpenPos; i: LONGINT;
    BEGIN
      NEW(PosStack1, LEN(PosStack^) * 2);
      FOR i := 0 TO LEN(PosStack^) - 1 DO PosStack1[i] := PosStack[i] END;
      PosStack := PosStack1
    END PosExpand;

  BEGIN
    INC(PosTop); IF PosTop = LEN(PosStack^) THEN PosExpand END;
    PosStack[PosTop] := S.Pos
  END PushPos;

  PROCEDURE PopPos(Arity: HeapType);
    VAR i: LONGINT;
  BEGIN
    FOR i := NextHeap + Arity TO NextHeap BY -1 DO
      PosHeap[i] := PosStack[PosTop]; DEC(PosTop)
    END
  END PopPos;

$  PROCEDURE GetHeap(Arity: HeapType; VAR Node: HeapType);
  BEGIN
    IF FreeList[Arity] = 0 THEN
```

```
      Node := NextHeap; IF NextHeap >= LEN(Heap^) - Arity - 1 THEN EvalExpand END;
      Heap[NextHeap] := 0; INC(NextHeap, Arity + 1)
    ELSE
      Node := FreeList[Arity]; FreeList[Arity] := Heap[FreeList[Arity]]; Heap[Node] := 0
    END;
    ASSERT(Heap[Node] DIV refConst = 0, 95)
  END GetHeap;

  PROCEDURE FreeHeap(Node: HeapType);
    VAR RArity: LONGINT; i: HeapType;
  BEGIN
    ASSERT(Node >= 0, 97);
    IF Heap[Node] DIV refConst <= 0 THEN
      RArity := (Heap[Node] MOD refConst) DIV arityConst;
      FOR i := Node + 1 TO Node + RArity DO FreeHeap(Heap[i]) END;
      ASSERT(Heap[Node] DIV refConst = 0, 96); ASSERT(Node > 0, 95);
      Heap[Node] := FreeList[RArity]; FreeList[RArity] := Node
    ELSE
      DEC(Heap[Node], refConst)
    END
  END FreeHeap;

  PROCEDURE CountHeap(): LONGINT;
    VAR i, HeapCells: LONGINT; Node: HeapType;
  BEGIN
    HeapCells := NextHeap;
    FOR i := 0 TO maxArity - 1 DO
      Node := FreeList[i];
      WHILE Node # 0 DO DEC(HeapCells, i + 1); Node := Heap[Node] END
    END;
    RETURN HeapCells
  END CountHeap;

$  PROCEDURE CountHeap(): LONGINT;
  BEGIN RETURN NextHeap
  END CountHeap;

$  PROCEDURE SetErr;
  BEGIN
    INC(ErrorCounter); IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, $Pos); IO.WriteText(IO.Msg, "  ");
  END SetErr;

  PROCEDURE Error(Msg: ARRAY OF CHAR);
  BEGIN
    SetErr; IO.WriteText(IO.Msg, Msg); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END Error;

  PROCEDURE PredError(Msg: ARRAY OF CHAR);
  BEGIN
    SetErr; IO.WriteText(IO.Msg, "predicate "); IO.WriteText(IO.Msg, Msg); IO.WriteText(IO.Msg, " failed");
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END PredError;

  PROCEDURE AnalyseError(VAR V: HeapType; Msg: ARRAY OF CHAR);
  BEGIN
    IF V # errVal THEN
      SetErr;
      IO.WriteText(IO.Msg, "analysis in '"); IO.WriteText(IO.Msg, Msg); IO.WriteText(IO.Msg, "' failed");
      IO.WriteLn(IO.Msg); IO.Update(IO.Msg);
$      INC(Heap[errVal], refConst); FreeHeap(V);
$      V := errVal;
    END
  END AnalyseError;

  PROCEDURE Equal(Ptr1, Ptr2: HeapType): BOOLEAN;
    VAR i: LONGINT;
  BEGIN
    IF Ptr1 = Ptr2 THEN RETURN TRUE
    ELSIF Heap[Ptr1] MOD refConst = Heap[Ptr2] MOD refConst THEN
      FOR i := 1 TO (Heap[Ptr1] MOD refConst) DIV arityConst DO
        IF ~ Equal(Heap[Ptr1 + i], Heap[Ptr2 + i]) THEN RETURN FALSE END
      END;
      RETURN TRUE
    END;
    RETURN FALSE
  END Equal;

  PROCEDURE Eq(Ptr1, Ptr2: HeapType; ErrMsg: ARRAY OF CHAR);
  BEGIN
    IF ~ Equal(Ptr1, Ptr2) THEN IF (Ptr1 # errVal) & (Ptr2 # errVal) THEN Error(ErrMsg) END END
```

```
  END Eq;

  PROCEDURE UnEq(Ptr1, Ptr2: HeapType; ErrMsg: ARRAY OF CHAR);
  BEGIN
    IF Equal(Ptr1, Ptr2) THEN IF (Ptr1 # errVal) & (Ptr2 # errVal) THEN Error(ErrMsg) END END
  END UnEq;

  PROCEDURE EvalInitSucceeds*(): BOOLEAN;
    CONST
      magic = 1818326597;
      name = "$";
      tabTimeStamp = $;
    VAR
      Tab : IO.File; OpenError : BOOLEAN;
      i : INTEGER; l : LONGINT;

    PROCEDURE LoadError(Msg : ARRAY OF CHAR);
    BEGIN
      IO.WriteText(IO.Msg, "  loading the evaluator table "); IO.WriteString(IO.Msg, name);
      IO.WriteText(IO.Msg, " failed\n\t");
      IO.WriteText(IO.Msg, Msg); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
    END LoadError;

  BEGIN   (* EvalInitSucceeds*(): BOOLEAN; *)
    IO.OpenFile(Tab, name, OpenError);
    IF OpenError THEN LoadError("it could not be opened"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # magic THEN LoadError("not an evaluator table"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # tabTimeStamp THEN LoadError("wrong time stamp"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # predefined THEN LoadError("wrong heap size"); RETURN FALSE END;
    IF Heap = NIL THEN NEW(Heap, initialHeapSize) END;
    WHILE predefined >= LEN(Heap^) DO EvalExpand END;
    FOR i := 0 TO predefined DO IO.GetLInt(Tab, l); Heap[i] := l END;
    IO.GetLInt(Tab, l);
    IF l # tabTimeStamp THEN LoadError("file corrupt"); RETURN FALSE END;
    IO.CloseFile(Tab);
    FOR i := 0 TO maxArity - 1 DO FreeList[i] := 0 END;
    NextHeap := predefined + 1; OutputSize := 0;
$    PosTop := -1; NEW(PosStack, 128);
    IF PosHeap = NIL THEN NEW(PosHeap, LEN(Heap^)) END;
$    RETURN TRUE
  END EvalInitSucceeds;

(* -------------- Ende eSLEAGGen.Fix --------------- *)

$
```

### 6.3.3 eEmitGen.Mod

```
MODULE eEmitGen;   (* JoDe 07.11.96, Version 1.01 *)

  IMPORT  Sets := eSets, IO := eIO, Scanner := eScanner, EAG := eEAG;

  CONST
    CaseLabels = 127;
  VAR
    Type3, Type2: Sets.OpenSet; StartMNont: INTEGER;

  PROCEDURE GenEmitProc*(Mod: IO.TextOut);
    VAR Lo, Hi: INTEGER; EmitSpace: BOOLEAN;

    PROCEDURE CalcSets(Nont: INTEGER);
      VAR A, F, M: INTEGER;
    BEGIN
      ASSERT(Nont >= EAG.firstMNont, 98); ASSERT(Nont < EAG.NextMNont, 97);
      IF EAG.MNont[Nont].IsToken THEN Sets.Incl(Type3, Nont) END;
      A := EAG.MNont[Nont].MRule;
      WHILE A # EAG.nil DO
        F := EAG.MAlt[A].Right;
        WHILE EAG.MembBuf[F] # EAG.nil DO
          M := EAG.MembBuf[F];
          IF (M > 0) THEN
            IF Sets.In(Type3, Nont) & ~ Sets.In(Type3, M) THEN Sets.Incl(Type3, M); CalcSets(M) END;
            IF Sets.In(Type2, Nont) & ~ Sets.In(Type2, M) THEN
              IF  ~ EAG.MNont[M].IsToken THEN Sets.Incl(Type2, M) END; CalcSets(M)
            END;
          END;
          INC(F)
        END;
        A := EAG.MAlt[A].Next
      END
    END CalcSets;

    PROCEDURE GenEmitProcs(MNonts: Sets.OpenSet);
      VAR N: INTEGER;

      PROCEDURE GenProcName(N: INTEGER; Type: Sets.OpenSet);
      BEGIN
        IO.WriteString(Mod, "Emit"); IO.WriteInt(Mod, N);
        IO.WriteString(Mod, "Type"); IF Type = Type2 THEN IO.Write(Mod, "2") ELSE IO.Write(Mod, "3") END;
      END GenProcName;

      PROCEDURE GenAlts(N: INTEGER);
        VAR A, F, M, arity, ANum: INTEGER;

        PROCEDURE WhiteSpace;
        BEGIN
          IF EmitSpace THEN IO.WriteString(Mod, "IO.Write(Out, ' '); ")
          ELSE IO.WriteString(Mod, "IO.WriteLn(Out); ")
          END
        END WhiteSpace;

      BEGIN (* GenAlts(N: INTEGER) *)
        A := EAG.MNont[N].MRule; ANum := 1;
        IO.WriteText(Mod, "\tCASE Heap[Ptr] MOD arityConst OF \n");
        WHILE A # EAG.nil DO
          IF ANum > CaseLabels THEN
            IO.WriteString(IO.Msg, "internal error: Too many metaalts in ");
            Scanner.WriteRepr(IO.Msg, EAG.MTerm[N].Id);
            IO.WriteLn(IO.Msg); IO.Update(IO.Msg); HALT(99)
          END;
          F := EAG.MAlt[A].Right; arity := 0;
          IO.WriteText(Mod, "\t| "); IO.WriteInt(Mod, ANum); IO.WriteText(Mod, ": ");
          WHILE EAG.MembBuf[F] # EAG.nil DO
            M := EAG.MembBuf[F];
            IF M < 0 THEN (* MetaTerminal *)
              IO.WriteText(Mod, "IO.WriteText(Out, ");
              Scanner.WriteRepr(Mod, EAG.MTerm[- M].Id);
              IO.WriteText(Mod, "); ");
              IF MNonts = Type2 THEN WhiteSpace END
            ELSE (* MetaNonTerminal *)
              IF (MNonts = Type3) OR EAG.MNont[M].IsToken THEN GenProcName(M, Type3)
              ELSE GenProcName(M, Type2)
              END;
              INC(arity); IO.WriteText(Mod, "(Heap[Ptr + ");
              IO.WriteInt(Mod, arity); IO.WriteString(Mod, "]); ");
```

```
            IF EAG.MNont[M].IsToken & (MNonts = Type2) THEN WhiteSpace END
          END;
          INC(F)
        END; IO.WriteLn(Mod);
        A := EAG.MAlt[A].Next; INC(ANum)
      END; IO.WriteText(Mod, "\tELSE IO.WriteInt(Out, Heap[Ptr]) \n\tEND \n")
    END GenAlts;

  BEGIN (* GenEmitProcs(MNonts: Sets.OpenSet) *)
    FOR N := EAG.firstMNont TO EAG.NextMNont - 1 DO
      IF Sets.In(MNonts, N) THEN
        IO.WriteText(Mod, "PROCEDURE ^ ");
        GenProcName(N, MNonts); IO.WriteText(Mod, "(Ptr: HeapType);\n")
      END
    END;
    IO.WriteLn(Mod);
    FOR N := EAG.firstMNont TO EAG.NextMNont - 1 DO
      IF Sets.In(MNonts, N) THEN
        IO.WriteText(Mod, "PROCEDURE ");
        GenProcName(N, MNonts); IO.WriteText(Mod, "(Ptr: HeapType);\n");
        IO.WriteText(Mod, "BEGIN \n\tINC(OutputSize, ((Heap[Ptr] MOD refConst) DIV arityConst) + 1); \n");
        GenAlts(N);
        IO.WriteText(Mod, "END "); GenProcName(N, MNonts); IO.WriteText(Mod, ";\n\n")
      END
    END
  END GenEmitProcs;

BEGIN (* GenEmitProc; *)
  EmitSpace := IO.IsOption("s");
  StartMNont := EAG.DomBuf[EAG.HNont[EAG.StartSym].Sig];
  Sets.New(Type3, EAG.NextMNont); Sets.New(Type2, EAG.NextMNont);
  IF ~ EAG.MNont[StartMNont].IsToken THEN Sets.Incl(Type2, StartMNont) END;
  CalcSets(StartMNont);
  IF ~ Sets.IsEmpty(Type3) THEN GenEmitProcs(Type3) END;
  IF ~ Sets.IsEmpty(Type2) THEN GenEmitProcs(Type2) END
END GenEmitProc;

PROCEDURE GenShowHeap*(Mod: IO.TextOut);
BEGIN
  IO.WriteText(Mod, '\t\tIF IO.IsOption("i") THEN \n');
  IO.WriteText(Mod, '\t\t\tIO.WriteText(IO.Msg, "\\ttree of "); ');
  IO.WriteText(Mod, 'IO.WriteInt(IO.Msg, OutputSize); \n\t\t\t');
  IO.WriteString(Mod, 'IO.WriteText(IO.Msg, " uses "); IO.WriteInt(IO.Msg, CountHeap()); ');
  IO.WriteText(Mod, 'IO.WriteText(IO.Msg, " of "); \n\t\t\t');
  IO.WriteString(Mod, 'IO.WriteInt(IO.Msg, NextHeap);  IO.WriteText(IO.Msg, " allocated, with "); ');
  IO.WriteText(Mod, 'IO.WriteInt(IO.Msg, predefined + 1); \n\t\t\t');
  IO.WriteText(Mod, 'IO.WriteText(IO.Msg, " predefined\\n"); IO.Update(IO.Msg); \n\t\tEND; \n');
END GenShowHeap;

PROCEDURE GenEmitCall*(Mod: IO.TextOut);
BEGIN
  IO.WriteText(Mod, "\t\t\tIF "); IF IO.IsOption("w") THEN IO.WriteText(Mod, "~ ") END;
  IO.WriteText(Mod, "IO.IsOption(\'w\') THEN IO.CreateOut(Out, \'");
  IO.WriteString(Mod, EAG.BaseName); IO.WriteText(Mod, ".Out\') ELSE Out := IO.Msg END; \n\t\t\t");
  IO.WriteString(Mod, "Emit"); IO.WriteInt(Mod, StartMNont); IO.WriteString(Mod, "Type");
  IF Sets.In(Type2, StartMNont) THEN IO.Write(Mod, "2") ELSE IO.Write(Mod, "3") END;
  IO.WriteText(Mod, '(V1); IO.WriteLn(Out); IO.Show(Out); \n');
END GenEmitCall;

END eEmitGen.
```

# Kapitel 7

# Compiler mit mächtigeren Auswertungsverfahren

Für nicht linksdefinierende EAGen kann die Affixauswertung nicht in einer depth-first left-to-right-Traversierung vorgenommen werden; sie erfüllen also nicht die LEAG-Bedingung, und es müssen mächtigere Auswertungsverfahren zur Anwendung kommen. Bei solchen Verfahren fällt nun die für die Evaluation nötige Traversierung des Ableitungsbaums nicht mehr mit den entsprechenden Prozeduren des Parsers zu dessen Aufbau zusammen, was eine Trennung der Evaluation von der Parsierung und somit eine statische Repräsentation des Ableitungsbaums notwendig macht. In Abschnitt 7.1 wird die Erstellung von Ableitungsbäumen durch automatisch generierte Compiler im bereits bekannten „Heap“ vorgestellt.

Als Beispiel eines mächtigeren Evaluationsverfahrens und damit gleichzeitig als Beispiel einer echten Generierung im Sinne einer *Abstraktion von Algorithmen* wird das *single sweep-Verfahren* und seine Implementierung vorgestellt. Dieses ist eine Verallgemeinerung des LEAG-Verfahrens, bei dem die Besuche von Unterbäumen eines Knotens im Ableitungsbaum permutiert durchgeführt werden dürfen. Jeder Unterbaum wird dabei genau einmal besucht; dies ist auch die in der Praxis irrelevante Einschränkung gegenüber dem bekannteren single visit-Verfahren, bei dem Unterbäume *höchstens* einmal besucht werden. Formal ist eine EAG genau dann single sweep-auswertbar, wenn es für die gemäß Abschnitt 2.2 transformierte EAG für jede Alternative eine Permutation der Faktoren gibt, für die die EAG linksdefinierend, also eine LEAG ist.

## 7.1 Die Erstellung statischer Ableitungsbäume

Anstatt einen generierten Parser zur Erstellung statischer Ableitungsbäume um entsprechende Konstruktoranweisungen anzureichern, wird er als spezieller Ein-Pass-Compiler generiert, dessen Spezifikation vom Modul `Shift` automatisch erzeugt wird (siehe Abschnitt 7.3). Dieser Compiler besitzt keine Ausgabeprozeduren, sondern übergibt die „Übersetzung“ seiner Eingabe, nämlich den in `Heap` vorliegenden Ableitungsbaum, der Traversierungsprozedur `TraverseSyntaxTree` eines separaten Evaluators.

Die Codierung des Ableitungsbaums entspricht dem üblichen Schema, das auf Seite 101 im Rahmen des Evaluatorgenerators vorgestellt wurde, und soll hier nur kurz wiederholt werden: Ein Knoten steht für eine Regelanwendung und wird durch eine Folge von Einträgen im Feld `Heap` realisiert. Der erste Eintrag kennzeichnet dabei die angewendete Regel durch die Nummer der entsprechenden Alternative. In den weiteren `Heap`-Einträgen sind die Verweise auf die zugehörigen Unterbäume gespeichert; ihre Anzahl ist durch die Stelligkeit der Alternative (ohne Prädikate) gegeben.

Um dem nachfolgenden Evaluator die Meldung der Positionen bei Erkennung von Kontextfehlern zu ermöglichen, muß zu den Knoten des Ableitungsbaums für jede mögliche Fehlerstelle eine Position gespeichert werden. Mögliche Fehlerstellen sind alle definierenden Parameterpositionen, für die Analysen oder Vergleiche scheitern können, sowie Vorkommen von Prädikaten in Regeln. Aus Gründen der Einfachheit wird in dieser Implementierung für *alle* Stellen, an denen gemäß den Codeschemata Analysecode stehen könnte, eine Position gespeichert. Diese Analysen können bei Betreten eines Knotens sowie nach Verlassen eines Unterbaums durchgeführt werden. Prädikate sollen als Fehlerposition die Position des davor besuchten Unterbaums verwenden. Daraus ergibt sich, daß zu jedem Eintrag in `Heap` genau eine Position gespeichert werden muß: Die des ersten Eintrags eines Knotens ist die Anfangsposition des durch die Regel erkannten Worts; zu einem Verweis auf einen Unterbaum wird die Position direkt hinter dem durch den Unterbaum repräsentierten Eingabewort gespeichert.

Eine einfache Zuordnung zwischen `Heap`-Einträgen und ihren Positionen wird durch ein Parallelfeld `PosHeap` realisiert. Zur Laufzeit wird dieses Positionsfeld durch Anweisungen gefüllt, die vom Evaluationsgenerator `SLEAGGen` in den generierten Compiler eingetragen werden. Dazu übergibt der Parsergenerator dem Evaluationsgenerator bei dessen Initialisierung die „Vertragsvereinbarung“ `sSweepPass`.

## 7.2 Die generierten single sweep-Evaluatoren

Ein generierter Evaluator ist ein eigenständiges Modul, das von einem vorgeschalteten Compiler importiert wird, der einen Ableitungsbaum aufbaut und anschließend dem Evaluator übergibt. Für jedes Nichtterminal wird eine eigene Traversierungsprozedur erzeugt, in die der Evaluationscode wie in den generierten Ein-Pass-Compilern direkt eingebettet wird. Da Knoten des Ableitungsbaums nicht mehrfach besucht werden, müssen die berechneten Affixwerte nicht statisch zwischen den Besuchen gespeichert werden, sondern können wieder dynamisch in Prozedurparametern gehalten werden. Das single sweep-Verfahren ist das mächtigste praktisch relevante Auswertungsverfahren, das diese speichereffiziente Repräsentation von Affixwerten erlaubt.

### 7.2.1 Überblick über die single sweep-Evaluatoren

Ein generierter Evaluator besteht aus einem Modul, dessen Grobstruktur im folgenden vereinfachten Programmfragment dargestellt ist:

```
MODULE Evaluatorname;

TYPE TreeType* = LONGINT;
     OpenTree* = POINTER TO ARRAY OF TreeType;
     OpenPos*  = POINTER TO ARRAY OF eIO.Position;
     ...

VAR Tree    : OpenTree;
    PosTree : OpenPos;
    Pos     : eIO.Position;

PROCEDURE P0(Adr : TreeType; VAR V1 : HeapType); (* Startsymbol *)
... (* Berechnet die Übersetzung des Ableitungsbaums *)
END P0;
(* ... weitere Prozeduren für die Nichtterminale: P1, P2,... *)
```

```
PROCEDURE Emit(Ptr : HeapType);
... (* Gibt die Übersetzung aus *)
END Emit;

PROCEDURE TraverseSyntaxTree*(Tree1 : OpenTree;
                              PosTree1 : OpenPos; Adr : TreeType);
  VAR V1 : HeapType;
BEGIN
  Tree := Tree1; PosTree := PosTree1;
  P0(Adr, V1); Emit(V1)
END TraverseSyntaxTree;

END Evaluatorname.
```

Die strenge Typprüfung der Programmiersprache Oberon macht es notwendig, die Typen `TreeType`, `OpenTree` und `OpenPos` zu exportieren, damit der vorgeschaltete Compiler den Ableitungsbaum und die Positionen in derart typisierten Feldern ablegen und der ebenfalls exportierten Prozedur `TraverseSyntaxTree` zusammen mit der Wurzel des Ableitungsbaums übergeben kann. Letztere Prozedur macht die übergebenen Felder durch Kopieren der Zeiger in globale Variablen allgemein zugänglich und ruft anschließend die für das Startsymbol erzeugte Prozedur zur Evaluation auf. Dort wird die statische Semantik überprüft und die Übersetzung der Eingabe – wiederum in einem hier nicht aufgeführten Feld `Heap` – erzeugt, die durch die Variable `V1` der Ausgabeprozedur übergeben wird.

### 7.2.2 Codeschemata für die Evaluationsprozeduren

Der Aufbau der für die Grundnichtterminale erzeugten Prozeduren ähnelt dem der Prozeduren eines Ein-Pass-Compilers und läßt sich durch folgendes Codeschema beschreiben:

```
PROCEDURE N(Adr : TreeType; ...);
...
BEGIN
  CASE Tree[Adr] MOD hyperArityConst OF
    | 1 : ...
    ...
    | k : Pos := PosTree[Adr];
          Analyse der Eingabeparameter der linken Seite der k-ten Alt.
          Code für die k-te Alternative
          Synthese der Ausgabeparameter der linken Seite der k-ten Alt.
    ...
    | n : ...
  END
END N;
```

Im Parameter `Adr` wird jeder Prozedur die Wurzel des auszuwertenden Unterbaums übergeben. Innerhalb der CASE-Anweisung wird die an diesem Knoten angewendete Regel bestimmt und der Code der entsprechenden Alternative ausgeführt. Dieser besteht aus den Aufrufen der Prozeduren für die in dieser Alternative vorkommenden Grundnichtterminale bzw. Prädikate zusammen mit den zugehörigen Analysen und Synthesen von Parametern. Die Aufrufe erfolgen dabei in einer vom Generator festgelegten Permutation der textuellen Reihenfolge nach folgendem Schema:

```
(* Aufruf eines Grundnichtterminals N *)
```

```
Synthese der Eingabeparameter zu N
N(Tree[Adr + Offset des Verweises auf den Unterbaum], ...);
Pos := PosTree[Adr + Offset des Verweises auf den Unterbaum];
Analyse der Ausgabeparameter zu N

(* Aufruf eines Prädikats N *)
Synthese der Eingabeparameter zu N
Pos := PosTree[Adr + vorheriger Offset];
N(...);
Analyse der Ausgabeparameter zu N
```

Kontextfehler können während der Evaluation nur in Analysen (und Vergleichen) sowie Prädikaten entdeckt werden; um in dem für Analysen generierten Code zusammen mit einer Meldung einfach die zugehörige Position ausgeben zu können, wird vor jeder Analyse die globale Variable `Pos` entsprechend gesetzt. Dies ist allerdings nur erforderlich, wenn an den entsprechenden Stellen auch tatsächlich Analysen durchgeführt werden. Prädikate bearbeiten keine Teile des Ableitungsbaums, weshalb den zugehörigen Prozeduren auch kein Verweis darauf übergeben wird. Im Falle des Scheiterns soll aber eine aussagekräftige Fehlerposition gemeldet werden. Hierfür wird die Position des unmittelbar vorher besuchten Teilbaums verwendet; wird in einer Alternative als erstes eine Prädikatprozedur aufgerufen, so gibt es keinen solchen vorher besuchten Teilbaum und die zum ersten Heapeintrag eines Knotens gespeicherte Position wird gemeldet.

## 7.3 Das Modul Shift

Dieses Modul wird vom Parsergenerator zur Erstellung von „Compilern“ verwendet, die lediglich Ableitungsbäume statisch aufbauen und die beliebigen separaten Evaluatoren vorgeschaltet werden können. Dazu wird aus der Grundgrammatik einer EAG eine weitere EAG automatisch generiert, die die identische Abbildung auf der kontextfreien Sprache beschreibt; strukturell wird dabei zwangsläufig die Transformation aus Abschnitt 2.2 realisiert, da die Meta-Grammatik keine EBNF-Konstrukte enthält. Jedes (eventuell anonyme) Hyper-Nichtterminal besitzt genau einen Ausgabeparameter zur Anhebung der erkannten Struktur auf die Meta-Ebene. Damit ist die generierte EAG eine SLEAG, für die sich ein Ein-Pass-Compiler erzeugen läßt; ohne Ausgabe liefert dieser Compiler die gewünschten Ableitungsbäume im Heap.

Für die generierte EAG wird keine textuelle Spezifikation erstellt, sondern die Repräsentation der internalisierten EAG wird entsprechend modifiziert. Die Grundgrammatik bleibt dabei unverändert, es werden jedoch alle Prädikate entfernt, und die Meta-Grammatik wird durch ein Pendant zur Grundgrammatik ersetzt, wobei EBNF-Konstrukte gemäß der erwähnten Transformation umgesetzt werden. Ebenso wird die Parametrisierung der Hyper-Nichtterminale überschrieben; die einfache Affixform auf einer linken Regelseite entspricht dabei direkt der jeweiligen Alternative. Der Earley-Parser ist daher für die Erstellung der zugehörigen Ableitungsbäume nicht erforderlich.

Da die Modifikation der Repräsentation einer EAG destruktiv ist, muß bei der Generierung eines Compilers zuerst der separate Evaluator erzeugt werden. Erst anschließend darf das Kommando `eELL1Gen.GenerateParser` verwendet werden, das die EAG mittels des Moduls `Shift` modifiziert und den Ein-Pass-Compiler generiert, der zuletzt den Ableitungsbaum dem Evaluator übergibt.

## 7.4 Der single sweep-Evaluatorgenerator

Der single sweep-Evaluatorgenerator versucht, für jede Alternative einer EAG eine Permutation der Faktoren zu bestimmen, mit der die Alternative linksdefinierend wird. Gelingt dies, so wird ein Evaluatormodul mit den in Abschnitt 7.2 vorgestellten Prozeduren zur Traversierung eines Ableitungsbaums generiert. Der Evaluationscode kann auch hier wieder unter Verwendung des Moduls `SLEAGGen` in diese eingebettet werden.

### 7.4.1 Generierung des Evaluatormoduls

Das Evaluatormodul wird in der Prozedur `GenerateMod` erstellt. Das Modulgerüst wird dazu aus einer festen Textdatei kopiert und an den Einfügemarken „`$`“ um die variablen Teile erweitert. Die Berechnung der Permutationen erfolgt nichtterminalweise während der Generierung.

Zuerst werden einige feste Teile sowie die Prädikatprozeduren mittels der Generierungsprozeduren des Moduls `SLEAGGen` eingefügt. Dahinter werden die Prozeduren für die Nichtterminale der Grundgrammatik geschrieben; wie schon im Parsergenerator werden auch hier davor einfach noch Vorwärtsdeklarationen für alle Prozeduren erzeugt, da sich diese im allgemeinen beliebig gegenseitig aufrufen können. Zuletzt werden die Ausgabeprozeduren unter Verwendung des Moduls `EmitGen` in den Evaluator geschrieben.

Um bei dem EBNF-Operator für Wiederholungen keiner Einschränkung zu unterliegen, wird für die Generierung die Transformation aus Abschnitt 2.2 tatsächlich durchgeführt. Da das Modul `SLEAGGen` zur Erzeugung des eingebetteten Evaluationscodes verwendet werden soll, müssen sowohl diese Änderungen als auch die berechneten Permutationen direkt in die Datenstrukturen des Moduls `EAG` eingearbeitet werden. Dabei können zur Vereinfachung die Terminale aus den Alternativen gelöscht werden, da sie für die Generierung des Evaluators irrelevant sind. Abschließend sollen alle Modifikationen wieder rückgängig gemacht werden.

Die Prozedur `SaveAndPatchNont` realisiert die Transformation der Alternativen eines Nichtterminals durch das Duplizieren der Datenstrukturen für die Alternativen (evtl. mit Modifikation der Parametrisierung) und für die darin vorkommenden Nichtterminale; gegebenenfalls werden noch Nichtterminale für den rekursiven Wiederaufruf neu angelgt. Die Kopien werden untereinander verkettet und in die Datenstruktur für das jeweilige Nichtterminal „eingehängt“. Zur Wiederherstellung durch die Prozedur `RestoreNont` wird einfach der Zeiger auf die ursprüngliche Datenstruktur gespeichert.

Geeignete Permutationen werden in der im nächsten Abschnitt beschriebenen Prozedur `ComputePermutation` berechnet. Die Erstellung der Nichtterminalprozeduren nach dem in Abschnitt 7.2 vorgestellten Schema erfolgt schließlich in der Prozedur `GenerateNont`.

Verstöße gegen die single sweep-Bedingung werden erst erkannt, wenn für ein Nichtterminal keine geeigneten Permutationen bestimmt werden können. In diesem Fall werden die bereits generierten Teile des Evaluators wieder gelöscht, und die weitere Generierung wird unterdrückt; die Bestimmung der übrigen Permutationen wird aber noch weiter versucht, um dem Benutzer alle Verstöße melden zu können. Ein reiner Test auf single sweep-Auswertbarkeit einer EAG wird dadurch realisiert, daß die Generierung von Anfang an unterdrückt wird, also nur die Permutationen berechnet werden.

### 7.4.2 Berechnung der Permutationen

In einem generierten Evaluator dürfen die Prozeduren zur Auswertung von Unterbäumen und die Prädikatprozeduren nur aufgerufen werden, wenn alle Eingabeparameter berechnet sind. Beim

single sweep-Verfahren legt bereits der Generator für jede Alternative eine Reihenfolge der Aufrufe fest; dazu werden die Nichtterminalvorkommen einer jeden Alternative derart umsortiert, daß diese linksdefinierend wird.

Eine naheliegende Lösung des Problems, eine geeignete Permutation zu finden, wäre, für jede Alternative einen „Abhängigkeitsgraphen“ aufzubauen, dessen Knoten die Nichtterminalvorkommen sind und dessen Kanten die durch applizierende und definierende Variablenvorkommen entstehenden Abhängigkeiten widerspiegeln, und diesen Graphen anschließend topologisch zu sortieren. Für das folgende nichttriviale Beispiel zeigt Abbildung 7.1 den Abhängigkeitsgraphen der einen Alternative des Nichtterminals `S`. Variablen auf definierender Position sind im Beispiel kursiv gestellt.

```
n = .     o = .     p = .     q = .     r = .

SS <+p> : S < , p> .
S <-r : r, +p : p> :
  A <n, o, r>  B <n, o, p>  C <n, r>  D <p, q, n>  E <n, q, r, p>.

A <+  : n, -o : o, -r : r> : "a".
B <-n : n, +  : o, -p : p> : "b".
C <+  : n, -r : r>         : "c".
D <+  : p, -q : q, -n : n> : "d".
E <-n : n, +  : q, -r : r, +  : p> : "e".
```

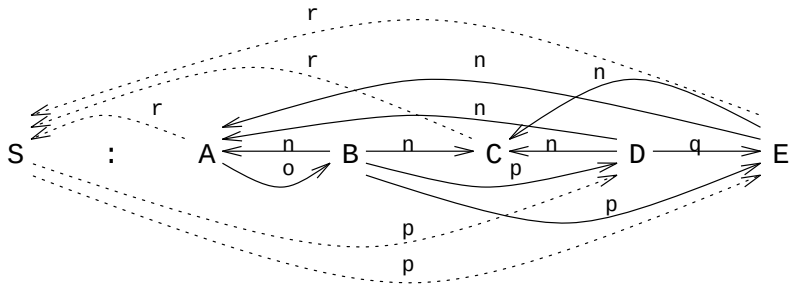



Abbildung 7.1: Abhängigkeitsgraph der einen Alternative des Nichtterminals `S`

Problematisch hieran ist, daß Variablen mehrfach auf definierender Position stehen können und noch nicht klar ist, welches dynamisch die erste definierende Position ist. In Eta wurde die linkeste Position als Definitionspunkt einfach festgelegt; dieses Vorgehen ist für das single sweep-Verfahren jedoch nicht geeignet. Der gezeigte Abhängigkeitsgraph enthält folglich zu viele Kanten und läßt sich dadurch nicht mehr topologisch sortieren (der Graph enthält Zyklen). Beispielsweise hängt der erste Faktor vom zweiten ab und umgekehrt. Wird jedoch der dritte Faktor nach vorne sortiert, so ist die Variable `n` dahinter definiert und für die weitere Sortierung müssen alle mit `n` markierten Kanten, die zu anderen Faktoren zeigen, aus dem Graphen gelöscht werden. Unter Vornahme solcher Löschungen kann der gezeigte Abhängigkeitsgraph nun sortiert werden.

Im allgemeinen gibt es zu einer Alternative viele mögliche Auswertungsreihenfolgen. Hier soll eine solche bestimmt werden, bei der die Faktoren möglichst ihre textuelle Reihenfolge behalten; insbesondere ergibt sich also für linksdefinierende EAGen eine Auswertung nach dem LEAG-Verfahren. Dies führt zu einer möglichst natürlichen Reihenfolge der Meldungen von Kontextfehlern durch den generierten Evaluator.

Die Bestimmung der Permutationen kann effizient wiederum mit einer Modifikation des Algorithmus zur Berechnung leerableitbarer Nichtterminale erfolgen (vgl. Abschnitt 4.2.3). Die wesentliche

Idee dabei ist, für Faktoren nicht die Abhängigkeiten von anderen Faktoren, sondern von Variablen darzustellen. Dazu werden die folgenden Datenstrukturen verwendet:

```
Var    : POINTER TO ARRAY OF RECORD Factors : INTEGER END;
Factor : POINTER TO ARRAY OF RECORD
           Vars, CountAppl, Prio : INTEGER;
           F : EAG.Factor
         END;

Edge     : POINTER TO ARRAY OF RECORD Dest, Next : INTEGER END;
NextEdge : INTEGER;
Stack    : POINTER TO ARRAY OF INTEGER; NextStack : INTEGER;

DefVars : Sets.OpenSet;
```

Initial werden für jeden Faktor einer Alternative eine Liste der bei ihm auf definierender Position stehenden Variablen, die Anzahl der Applikationen noch nicht definierter Variablen, seine Priorität (analog zur Bestimmung der Fortsetzungsgrammatik in Abschnitt 5.2.2.3) und ein Verweis auf den Faktor gespeichert. Zu jeder Variablen einer Alternative wird eine Liste derjenigen Faktoren angelegt, bei denen sie auf applizierender Position steht. Um lineare Suche zu vermeiden, führen in allen Listen mehrfache Applikationen bzw. Definitionen einer Variablen bei einem Faktor zu mehrfachen Eintragungen. Sämtliche Listen werden im Feld `Edge` realisiert.

In `Stack` wird die Menge derjenigen Faktoren gespeichert, deren Applikationszähler auf Null gesunken ist, die also nur bereits definierte Variablen applizieren. In `DefVars` werden während der Sortierung der Faktoren sukzessive die bereits definierten Variablen aufgenommen; initialisiert wird diese Menge mit den auf der linken Regelseite definierten Variablen. Die in Abbildung 7.1 gepunktet gezeichneten Abhängigkeiten zwischen Faktoren und linker Regelseite müssen in die Anfangsbelegung der Datenstrukturen erst gar nicht übernommen werden, wie in Abbildung 7.2 dargestellt ist.

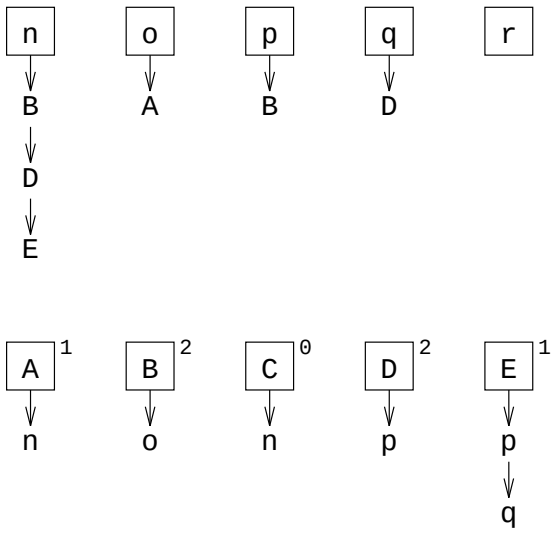



Abbildung 7.2: Anfangsbelegung der Datenstrukturen für das Beispiel

In `Stack` ist für das Beispiel anfangs nur der Faktor `C` eingetragen, in `DefVars` nur die Variable `r` enthalten.

Die Priorität der Faktoren steuert die Wahl des Faktors, der aus `Stack` entfernt wird; eine Auswahl entspricht der Definition der bei diesem Faktor auf definierender Position stehenden Variablen,

was durch die geschickte Wahl der Datenstrukturen nun leicht in die Menge `DefVars` und die Applikationszähler der Faktoren eingearbeitet werden kann. Sinkt dabei ein solcher Zähler auf Null, so wird der zugehörige Faktor in `Stack` aufgenommen.

Ist `Stack` schließlich völlig geleert, so müssen alle Faktoren genau einmal daraus entfernt worden sein und dadurch eine neue Position in der Umsortierung bekommen haben; ferner müssen alle von der linken Regelseite applizierten Variablen definiert sein. Sind diese Bedingungen erfüllt, so ist die Alternative single sweep-auswertbar und die Auswertungsreihenfolge ist durch Umsortierung der Faktoren im Modul `EAG` „gespeichert“. Ansonsten wird eine entsprechende Fehlermeldung ausgegeben und eine globale Variable `Error` auf `TRUE` gesetzt, um eine weitere Generierung zu unterdrücken.

## 7.5 Implementierungen

### 7.5.1 eSSweep.Mod

```
MODULE eSSweep;    (* SteWe 08/96  -  Ver 1.1 *)

IMPORT Sets := eSets, IO := eIO, EAG := eEAG, EmitGen := eEmitGen, EvalGen := eSLEAGGen;

CONST nil = 0; indexOfFirstAlt = 1;

VAR FactorOffset : POINTER TO ARRAY OF INTEGER;
    GenNonts, GenFactors : Sets.OpenSet;
    Error, ShowMod, Compiled : BOOLEAN;

PROCEDURE Init;
BEGIN
  NEW(FactorOffset, EAG.NextHFactor + EAG.NextHAlt + 1);
  Sets.New(GenFactors, EAG.NextHNont);
  Sets.Intersection(GenFactors, EAG.Prod, EAG.Reach);
  Sets.New(GenNonts, EAG.NextHNont);
  Sets.Difference(GenNonts, GenFactors, EAG.Pred);
  Error := FALSE; ShowMod := IO.IsOption("m")
END Init;

PROCEDURE Finit;
BEGIN
  FactorOffset := NIL
END Finit;

PROCEDURE GenerateMod(CreateMod : BOOLEAN);
  CONST firstEdge = 1; firstStack = 0;

  TYPE OpenEdge = POINTER TO ARRAY OF RECORD Dest, Next : INTEGER END;

  VAR
    N, V : INTEGER;
    Mod  : IO.TextOut; Fix : IO.TextIn;
    Name : ARRAY EAG.BaseNameLen + 10 OF CHAR;
    OpenError, CompileError : BOOLEAN;
    SavedNontDef : EAG.Rule;
    SavedNextHFactor, SavedNextHAlt : INTEGER;
    Factor : POINTER TO ARRAY OF RECORD Vars, CountAppl, Prio : INTEGER; F : EAG.Factor END;
    Var    : POINTER TO ARRAY OF RECORD Factors : INTEGER END;
    Edge   : OpenEdge; NextEdge : INTEGER;
    Stack  : POINTER TO ARRAY OF INTEGER; NextStack : INTEGER;
    DefVars : Sets.OpenSet;

  PROCEDURE Expand;
    VAR Edge1 : OpenEdge; i : LONGINT;
  BEGIN
    IF NextEdge >= LEN(Edge^) THEN NEW(Edge1, 2 * LEN(Edge^));
      FOR i := firstEdge TO LEN(Edge^) - 1 DO Edge1[i] := Edge[i] END;
      Edge := Edge1
    END
  END Expand;

  PROCEDURE InclFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = 0X THEN
        IO.WriteText(IO.Msg, "\n  error: unexpected end of eSSweep.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Write(Mod, c); IO.Read(Fix, c)
    END
  END InclFix;

  PROCEDURE Append(VAR Dest : ARRAY OF CHAR; Src, Suf : ARRAY OF CHAR);
    VAR i, j : INTEGER;
  BEGIN
    i := 0; j := 0;
    WHILE (Src[i] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Src[i]; INC(i) END;
    WHILE (Suf[j] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Suf[j]; INC(i); INC(j) END;
    Dest[i] := 0X
  END Append;
```

```
PROCEDURE HyperArity() : INTEGER;
  VAR N, i, Max : INTEGER; A : EAG.Alt; Nonts : Sets.OpenSet;
BEGIN
  Sets.New(Nonts, EAG.NextHNont);
  Sets.Difference(Nonts, EAG.All, EAG.Pred);
  Max := 0;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(Nonts, N) THEN
      A := EAG.HNont[N].Def.Sub; i := 0;
      REPEAT INC(i); A := A.Next UNTIL A = NIL;
      IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN INC(i) END;
      IF i > Max THEN Max := i END;
    END
  END;
  i := 1; WHILE i <= Max DO i := i * 2 END;
  RETURN i
END HyperArity;

PROCEDURE SaveAndPatchNont(N : INTEGER);
  VAR Def : EAG.Grp; A, A1, A2 : EAG.Alt; F : EAG.Factor; F1, F2 : EAG.Nont;
BEGIN
  SavedNontDef := EAG.HNont[N].Def;
  SavedNextHFactor := EAG.NextHFactor; SavedNextHAlt := EAG.NextHAlt;
  NEW(Def);
  A := EAG.HNont[N].Def.Sub; A2 := NIL;
  REPEAT
    NEW(A1); A1^ := A^; A1.Sub := NIL; A1.Last := NIL; A1.Next := NIL;
    IF A2 # NIL THEN A2.Next := A1 ELSE Def.Sub := A1 END; A2 := A1;
    F := A.Sub; F2 := NIL;
    WHILE F # NIL DO
      IF (F IS EAG.Nont) & Sets.In(GenFactors, F(EAG.Nont).Sym) THEN
        NEW(F1); F1^ := F(EAG.Nont)^;
        F1.Prev := F2; F1.Next := NIL; A1.Last := F1;
        IF F2 # NIL THEN F2.Next := F1 ELSE A1.Sub := F1 END; F2 := F1
      END;
      F := F.Next
    END;
    IF EAG.HNont[N].Def IS EAG.Rep THEN
      NEW(F1);
      F1.Ind := EAG.NextHFactor; INC(EAG.NextHFactor);
      F1.Prev := A1.Last; A1.Last := F1; IF A1.Sub = NIL THEN A1.Sub := F1 END;
      IF F1.Prev # NIL THEN F1.Prev.Next := F1 END;
      F1.Next := NIL;
      F1.Sym := N;
      F1.Pos := A1.Actual.Pos;
      F1.Actual := A1.Actual; A1.Actual.Pos := IO.UndefPos; A1.Actual.Params := EAG.empty
    END;
    A := A.Next
  UNTIL A = NIL;
  IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN
    NEW(A1);
    A1.Ind := EAG.NextHAlt; INC(EAG.NextHAlt);
    A1.Up := N; A1.Next := NIL; A1.Sub := NIL; A1.Last := NIL;
    IF EAG.HNont[N].Def IS EAG.Opt THEN
      A1.Scope := EAG.HNont[N].Def(EAG.Opt).Scope;
      A1.Formal := EAG.HNont[N].Def(EAG.Opt).Formal;
      A1.Pos := EAG.HNont[N].Def(EAG.Opt).EmptyAltPos
    ELSE A1.Scope := EAG.HNont[N].Def(EAG.Rep).Scope;
      A1.Formal := EAG.HNont[N].Def(EAG.Rep).Formal;
      A1.Pos := EAG.HNont[N].Def(EAG.Rep).EmptyAltPos
    END;
    A1.Actual.Params := EAG.empty; A1.Actual.Pos := IO.UndefPos;
    A2.Next := A1
  END;
  EAG.HNont[N].Def := Def
END SaveAndPatchNont;

PROCEDURE RestoreNont(N : INTEGER);
BEGIN
  EAG.HNont[N].Def := SavedNontDef;
  EAG.NextHFactor := SavedNextHFactor;
  EAG.NextHAlt := SavedNextHAlt
END RestoreNont;

PROCEDURE ComputePermutation(N : INTEGER);
  CONST def = 0; right = 1; appl = 2;

  VAR
    A : EAG.Alt; F, F1 : EAG.Factor;
```

```
    Prio, Index, Offset, NE, VE, V : INTEGER;

  PROCEDURE TravParams(op, P : INTEGER; F : EAG.Factor);
    VAR Def : BOOLEAN;

    PROCEDURE NewEdge(VAR From : INTEGER; To : INTEGER);
    BEGIN
      IF NextEdge >= LEN(Edge^) THEN Expand END;
      Edge[NextEdge].Dest := To; Edge[NextEdge].Next := From; From := NextEdge;
      INC(NextEdge)
    END NewEdge;

    PROCEDURE Tree(Node : INTEGER);
      VAR n : INTEGER;
    BEGIN
      IF Node < 0 THEN
        CASE op OF
          def   : IF Def THEN Sets.Incl(DefVars, - Node) END
          | right : IF ~ Sets.In(DefVars, - Node) THEN
                IF Def THEN NewEdge(Factor[F.Ind].Vars, - Node)
                ELSE NewEdge(Var[- Node].Factors, F.Ind); INC(Factor[F.Ind].CountAppl)
                END
              END
          | appl : IF ~ Def & ~ Sets.In(DefVars, - Node) THEN
                IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, EAG.ParamBuf[P].Pos);
                IO.WriteText(IO.Msg, "  variable '"); EAG.WriteVar(IO.Msg, - Node);
                IO.WriteText(IO.Msg, " is not defined"); IO.Update(IO.Msg); Error := TRUE
              END
        END
      ELSE FOR n := 1 TO EAG.MAlt[EAG.NodeBuf[Node]].Arity DO Tree(EAG.NodeBuf[Node + n]) END
      END
    END Tree;

  BEGIN
    WHILE EAG.ParamBuf[P].Affixform # EAG.nil DO
      Def := EAG.ParamBuf[P].isDef; Tree(EAG.ParamBuf[P].Affixform); INC(P)
    END
  END TravParams;

  PROCEDURE Pop(VAR F : EAG.Factor);
    VAR i, MinPrio, MinIndex : INTEGER;
  BEGIN
    MinPrio := MAX(INTEGER);
    FOR i := firstStack TO NextStack - 1 DO
      IF Factor[Stack[i]].Prio < MinPrio THEN
        MinPrio := Factor[Stack[i]].Prio; MinIndex := i
      END
    END;
    F := Factor[Stack[MinIndex]].F;
    Stack[MinIndex] := Stack[NextStack - 1]; DEC(NextStack)
  END Pop;

BEGIN  (* ComputePermutation(N : INTEGER) *)
  A := EAG.HNont[N].Def.Sub;
  REPEAT
    Sets.Empty(DefVars); NextEdge := firstEdge; NextStack := firstStack;
    (* Var is initialized global, Factor below *)
    TravParams(def, A.Formal.Params, NIL);
    F := A.Sub; Prio := 0; Offset := 1;
    WHILE F # NIL DO
      Factor[F.Ind].Vars := nil; Factor[F.Ind].CountAppl := 0;
      Factor[F.Ind].Prio := Prio; INC(Prio); Factor[F.Ind].F := F;
      IF ~ Sets.In(EAG.Pred, F(EAG.Nont).Sym) THEN
        FactorOffset[F.Ind] := Offset; INC(Offset)
      END;
      TravParams(right, F(EAG.Nont).Actual.Params, F);
      IF Factor[F.Ind].CountAppl = 0 THEN
        Stack[NextStack] := F.Ind; INC(NextStack)
      END;
      F := F.Next
    END;
    A.Sub := NIL; A.Last := NIL; F1 := NIL; Index := 0;
    WHILE NextStack > firstStack DO
      Pop(F); F.Prev := F1; F.Next := NIL; A.Last := F;
      IF F1 # NIL THEN F1.Next := F ELSE A.Sub := F END; F1 := F; INC(Index);
      VE := Factor[F.Ind].Vars;
      WHILE VE # nil DO
        V := Edge[VE].Dest;
        IF ~ Sets.In(DefVars, V) THEN
          NE := Var[V].Factors;
```

```
          WHILE NE # nil DO
            DEC(Factor[Edge[NE].Dest].CountAppl);
            IF Factor[Edge[NE].Dest].CountAppl = 0 THEN
              Stack[NextStack] := Edge[NE].Dest; INC(NextStack)
            END;
            NE := Edge[NE].Next
          END;
          Sets.Incl(DefVars, V)
        END;
        VE := Edge[VE].Next
      END;
    END;
    IF Index = Prio THEN TravParams(appl, A.Formal.Params, NIL);
    ELSE
      IO.WriteText(IO.Msg, "\n  "); IO.WritePos(IO.Msg, A.Pos);
      IO.WriteText(IO.Msg, "  alternative is not single sweep");
      IO.Update(IO.Msg); Error := TRUE
    END;
    A := A.Next
  UNTIL A = NIL;
END ComputePermutation;

PROCEDURE GenerateNont(N : INTEGER);
  VAR A : EAG.Alt; F, F1 : EAG.Factor; AltIndex : INTEGER;
BEGIN
  EvalGen.ComputeVarNames(N, FALSE);
  IO.WriteText(Mod, "PROCEDURE P"); IO.WriteInt(Mod, N);
  IO.WriteText(Mod, "(Adr : TreeType");
  EvalGen.GenFormalParams(N, FALSE); IO.WriteText(Mod, ");");
  IO.WriteText(Mod, "   (* "); EAG.WriteHNont(Mod, N);
  IF EAG.HNont[N].Id < 0 THEN
    IO.WriteText(Mod, " in "); EAG.WriteNamedHNont(Mod, N);
  END;
  IO.WriteText(Mod, " *)\n");
  EvalGen.GenVarDecl(N);
  IO.WriteText(Mod, "BEGIN\n");
  IO.WriteText(Mod, "\tCASE Tree[Adr] MOD hyperArityConst OF\n");
  A := EAG.HNont[N].Def.Sub; AltIndex := indexOfFirstAlt;
  REPEAT
    IO.WriteText(Mod, "\t\t| "); IO.WriteInt(Mod, AltIndex);
    IO.WriteText(Mod, " : \n");
    EvalGen.InitScope(A.Scope);
    IF EvalGen.PosNeeded(A.Formal.Params) THEN
      IO.WriteText(Mod, "Pos := PosTree[Adr];\n")
    END;
    EvalGen.GenAnalPred(N, A.Formal.Params);
    F := A.Sub;
    WHILE F # NIL DO
      IF ~ Sets.In(EAG.Pred, F(EAG.Nont).Sym) THEN
        EvalGen.GenSynPred(N, F(EAG.Nont).Actual.Params);
        IO.WriteText(Mod, "\t\tP"); IO.WriteInt(Mod, F(EAG.Nont).Sym);
        IO.WriteText(Mod, "(Tree[Adr + "); IO.WriteInt(Mod, FactorOffset[F.Ind]);
        IO.WriteText(Mod, "]"); EvalGen.GenActualParams(F(EAG.Nont).Actual.Params, FALSE);
        IO.WriteText(Mod, ");   (* "); EAG.WriteHNont(Mod, F(EAG.Nont).Sym);
        IF EAG.HNont[F(EAG.Nont).Sym].Id < 0 THEN
          IO.WriteText(Mod, " in "); EAG.WriteNamedHNont(Mod, F(EAG.Nont).Sym);
        END;
        IO.WriteText(Mod, " *)\n");
        IF EvalGen.PosNeeded(F(EAG.Nont).Actual.Params) THEN
          IO.WriteText(Mod, "Pos := PosTree[Adr + "); IO.WriteInt(Mod, FactorOffset[F.Ind]);
          IO.WriteText(Mod, "];\n")
        END;
        EvalGen.GenAnalPred(N, F(EAG.Nont).Actual.Params);
      ELSE
        EvalGen.GenSynPred(N, F(EAG.Nont).Actual.Params);
        IO.WriteText(Mod, "Pos := PosTree[Adr + ");
        F1 := F.Prev;
        WHILE (F1 # NIL) & Sets.In(EAG.Pred, F1(EAG.Nont).Sym) DO F1 := F1.Prev END;
        IF F1 = NIL THEN IO.WriteInt(Mod, 0)
        ELSE IO.WriteInt(Mod, FactorOffset[F1.Ind])
        END;
        IO.WriteText(Mod, "];\n");
        EvalGen.GenPredCall(F(EAG.Nont).Sym, F(EAG.Nont).Actual.Params);
        EvalGen.GenAnalPred(N, F(EAG.Nont).Actual.Params)
      END;
      F := F.Next
    END;
    EvalGen.GenSynPred(N, A.Formal.Params);
    A := A.Next; INC(AltIndex)
  UNTIL A = NIL;
```

```
    IO.WriteText(Mod, "\tEND;\n");
    IO.WriteText(Mod, "END P"); IO.WriteInt(Mod, N); IO.WriteText(Mod, ";\n\n")
  END GenerateNont;

BEGIN  (* GenerateMod *)
  EvalGen.InitTest; Error := Error OR ~ EvalGen.PredsOK();
  IF CreateMod THEN
    IO.OpenIn(Fix, "eSSweep.Fix", OpenError);
    IF OpenError THEN
      IO.WriteText(IO.Msg, "\n  error: could not open eSSweep.Fix\n"); IO.Update(IO.Msg); HALT(99)
    END;
    Append(Name, EAG.BaseName, "Eval");
    IO.CreateModOut(Mod, Name);
    IF ~ Error THEN
      EvalGen.InitGen(Mod, EvalGen.sSweepPass);
      InclFix("$"); IO.WriteText(Mod, Name);        (* module name *)
      InclFix("$"); IO.WriteInt(Mod, HyperArity());      (* hyperArityConst *)
      InclFix("$");
      EvalGen.GenDeclarations;                  (* evaluator global things *)
      FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
        IF Sets.In(GenNonts, N) THEN
          IO.WriteText(Mod, "PROCEDURE^ P"); IO.WriteInt(Mod, N);
          IO.WriteText(Mod, "(Adr : TreeType");
          EvalGen.GenFormalParams(N, FALSE); IO.WriteText(Mod, ");");
          IO.WriteText(Mod, "   (* "); EAG.WriteHNont(Mod, N);
          IF EAG.HNont[N].Id < 0 THEN
            IO.WriteText(Mod, " in "); EAG.WriteNamedHNont(Mod, N);
          END;
          IO.WriteText(Mod, " *)\n");
        END
      END;
      EvalGen.GenPredProcs; IO.WriteLn(Mod)
    END
  END;
  NEW(Factor, EAG.NextHFactor + EAG.NextHAlt + 1);
  NEW(Var, EAG.NextVar + 1); NEW(Edge, 127); NEW(Stack, EAG.NextHFactor + 1);
  Sets.New(DefVars, EAG.NextVar);
  FOR V := EAG.firstVar TO EAG.NextVar - 1 DO Var[V].Factors := nil END;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, N) THEN
      SaveAndPatchNont(N);
      ComputePermutation(N);
      IF ~ Error THEN
        Error :=  ~ EvalGen.IsLEAG(N, TRUE);
        IF ~ Error & CreateMod THEN GenerateNont(N) END
      END;
      RestoreNont(N)
    END
  END;
  IF CreateMod THEN
    IF ~ Error THEN
      EmitGen.GenEmitProc(Mod);
      InclFix("$"); IO.WriteText(Mod, "P"); IO.WriteInt(Mod, EAG.StartSym);
      InclFix("$"); EmitGen.GenEmitCall(Mod);
      InclFix("$"); EmitGen.GenShowHeap(Mod);
      InclFix("$"); IO.WriteText(Mod, EAG.BaseName); IO.WriteText(Mod, "Eval");
      InclFix("$");                             (* rest of file *)
      IO.Update(Mod);
      IF ShowMod THEN IO.Show(Mod)
      ELSE IO.Compile(Mod, CompileError); Compiled := TRUE;
        IF CompileError THEN IO.Show(Mod) END
      END
    END;
    EvalGen.FinitGen; IO.CloseIn(Fix); IO.CloseOut(Mod)
  END;
  EvalGen.FinitTest
END GenerateMod;

PROCEDURE Test*;
  VAR SaveHistory : SET;
BEGIN
  IO.WriteText(IO.Msg, "SSweep testing "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteText(IO.Msg, "   "); IO.Update(IO.Msg);
  IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
    EXCL(EAG.History, EAG.isSSweep);
    Init; SaveHistory := EAG.History; EAG.History := {};
    GenerateMod(FALSE);
    EAG.History := SaveHistory;
    IF ~ Error THEN IO.WriteText(IO.Msg, "ok"); INCL(EAG.History, EAG.isSSweep) END;
    Finit
```

```
  END;
  IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
END Test;

PROCEDURE Generate*;
  VAR SaveHistory : SET;
BEGIN
  IO.WriteText(IO.Msg, "SSweep writing "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteText(IO.Msg, "   "); IO.Update(IO.Msg); Compiled := FALSE;
  IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
    EXCL(EAG.History, EAG.isSSweep);
    Init; SaveHistory := EAG.History; EAG.History := {};
    GenerateMod(TRUE);
    EAG.History := SaveHistory;
    IF ~ Error THEN INCL(EAG.History, EAG.isSSweep); INCL(EAG.History, EAG.hasEvaluator) END;
    Finit
  END;
  IF ~ Compiled THEN IO.WriteLn(IO.Msg) END; IO.Update(IO.Msg)
END Generate;

END eSSweep.
```

### 7.5.2 eSSweep.Fix

```
MODULE $;    (* eSSweep.Fix, SteWe 08/96  -  Ver 1.2 *)

IMPORT IO := eIO;

CONST hyperArityConst = $;

TYPE TreeType* = LONGINT;
     OpenTree* = POINTER TO ARRAY OF TreeType;
     OpenPos* = POINTER TO ARRAY OF IO.Position;

VAR Tree : OpenTree;
    PosTree : OpenPos;
    ErrorCounter : LONGINT;
    Pos : IO.Position;
    Out : IO.TextOut;

$  (* insert evaluator global things *)

PROCEDURE TraverseSyntaxTree*(Tree1 : OpenTree; PosTree1 : OpenPos;
                  ErrCounter : LONGINT; Adr : TreeType; HyperArity : INTEGER);
  VAR V1 : HeapType;
BEGIN
  IF HyperArity # hyperArityConst THEN
    IO.WriteText(IO.Msg, "\n   internal error: 'arityConst' is wrong\n");
    IO.Update(IO.Msg); HALT(99)
  END;
  Tree := Tree1;
  PosTree := PosTree1;
  ErrorCounter := ErrCounter;
  $(Adr, V1);
  IF ErrorCounter > 0 THEN
    IO.WriteText(IO.Msg, "  "); IO.WriteInt(IO.Msg, ErrorCounter);
    IO.WriteText(IO.Msg, " errors detected\n"); IO.Update(IO.Msg)
  ELSE
$  END;
$  Tree := NIL; PosTree := NIL
END TraverseSyntaxTree;

END $.  (* insert module name *)
$
```

### 7.5.3 eShift.Mod

```
MODULE eShift;    (* SteWe 07/96  -  Ver 1.0 *)

IMPORT Sets := eSets, IO := eIO, EAG := eEAG;

CONST nil = EAG.nil;

PROCEDURE Shift*(Dummy : INTEGER);
  VAR
    HN, HT : INTEGER; HA : EAG.Alt; HF : EAG.Factor;
    Rhs, Num, Var : INTEGER;
    GenNonts, Del : Sets.OpenSet;

BEGIN
  Sets.New(GenNonts, EAG.NextHNont);
  Sets.Difference(GenNonts, EAG.All, EAG.Pred);
  Sets.New(Del, EAG.NextHNont); Sets.Empty(Del);
  EAG.NextMAlt := EAG.firstMAlt;
  EAG.NextMemb := EAG.firstMemb;
  EAG.NextVar := EAG.firstVar;
  EAG.NextNode := EAG.firstNode;
  EAG.NextParam := EAG.firstParam;
  EAG.NextMTerm := EAG.NextHTerm - EAG.firstHTerm + EAG.firstMTerm;
  WHILE EAG.NextMTerm >= LEN(EAG.MTerm^) DO EAG.Expand END;
  FOR HT := EAG.firstHTerm TO EAG.NextHTerm - 1 DO
    EAG.MTerm[HT - EAG.firstHTerm + EAG.firstMTerm].Id := EAG.HTerm[HT].Id
  END;
  EAG.NextMNont := EAG.NextHNont - EAG.firstHNont + EAG.firstMNont;
  WHILE EAG.NextMNont >= LEN(EAG.MNont^) DO EAG.Expand END;
  FOR HN := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, HN) THEN
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].Id := EAG.HNont[HN].NamedId;
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].MRule := nil;
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].IsToken := EAG.HNont[HN].IsToken;
      HA := EAG.HNont[HN].Def.Sub;
      REPEAT
        EAG.Scope := EAG.NextVar; HA.Scope.Beg := EAG.NextVar;
        HA.Formal.Pos := IO.UndefPos;
        HA.Formal.Params := EAG.NextParam;
        EAG.AppParam(EAG.NextNode, IO.UndefPos);
        EAG.ParamBuf[EAG.NextParam - 1].isDef := FALSE;
        EAG.AppParam(nil, IO.UndefPos);
        EAG.NodeBuf[EAG.NextNode] := EAG.NextMAlt;
        INC(EAG.NextNode); IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END;
        Rhs := EAG.NextMemb;
        HF := HA.Sub; Num := 2;
        WHILE HF # NIL DO
          IF HF IS EAG.Term THEN
            EAG.AppMemb(- (HF(EAG.Term).Sym - EAG.firstHTerm + EAG.firstMTerm))
          ELSIF Sets.In(GenNonts, HF(EAG.Nont).Sym) THEN
            EAG.AppMemb(HF(EAG.Nont).Sym - EAG.firstHNont + EAG.firstMNont);
            Var := EAG.FindVar(HF(EAG.Nont).Sym - EAG.firstHNont + EAG.firstMNont,
              Num, IO.UndefPos, TRUE);
            EAG.NodeBuf[EAG.NextNode] := - Var;
            INC(EAG.NextNode); IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END;
            HF(EAG.Nont).Actual.Pos := IO.UndefPos;
            HF(EAG.Nont).Actual.Params := EAG.NextParam;
            EAG.AppParam(- Var, IO.UndefPos);
            EAG.ParamBuf[EAG.NextParam - 1].isDef := TRUE;
            EAG.AppParam(nil, IO.UndefPos);
            INC(Num)
          ELSE
            IF HF.Next # NIL THEN HF.Next.Prev := HF.Prev END;
            IF HF.Prev # NIL THEN HF.Prev.Next := HF.Next END;
            IF HF = HA.Sub THEN HA.Sub := HF.Next END;
            IF HF = HA.Last THEN HA.Last := HF.Prev END
          END;
          HF := HF.Next
        END;
        IF EAG.HNont[HN].Def IS EAG.Rep THEN
          EAG.AppMemb(HN - EAG.firstHNont + EAG.firstMNont);
          Var := EAG.FindVar(HN - EAG.firstHNont + EAG.firstMNont, Num, IO.UndefPos, TRUE);
          EAG.NodeBuf[EAG.NextNode] := - Var;
          INC(EAG.NextNode); IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END;
          HA.Actual.Pos := IO.UndefPos;
          HA.Actual.Params := EAG.NextParam;
          EAG.AppParam(- Var, IO.UndefPos);
          EAG.ParamBuf[EAG.NextParam - 1].isDef := TRUE;
```

```
          EAG.AppParam(nil, IO.UndefPos)
        END;
        EAG.AppMemb(nil);
        EAG.AppMemb(EAG.NewMAlt(HN - EAG.firstHNont + EAG.firstMNont, Rhs));
        HA.Scope.End := EAG.NextVar;
        HA := HA.Next
      UNTIL HA = NIL;
      IF (EAG.HNont[HN].Def IS EAG.Opt) OR (EAG.HNont[HN].Def IS EAG.Rep) THEN
        IF EAG.HNont[HN].Def IS EAG.Opt THEN
          EAG.HNont[HN].Def(EAG.Opt).Formal.Pos := IO.UndefPos;
          EAG.HNont[HN].Def(EAG.Opt).Formal.Params := EAG.NextParam;
          EAG.HNont[HN].Def(EAG.Opt).Scope.Beg := EAG.NextVar;
          EAG.HNont[HN].Def(EAG.Opt).Scope.End := EAG.NextVar
        ELSE
          EAG.HNont[HN].Def(EAG.Rep).Formal.Pos := IO.UndefPos;
          EAG.HNont[HN].Def(EAG.Rep).Formal.Params := EAG.NextParam;
          EAG.HNont[HN].Def(EAG.Rep).Scope.Beg := EAG.NextVar;
          EAG.HNont[HN].Def(EAG.Rep).Scope.End := EAG.NextVar
        END;
        EAG.AppParam(EAG.NextNode, IO.UndefPos);
        EAG.ParamBuf[EAG.NextParam - 1].isDef := FALSE;
        EAG.AppParam(nil, IO.UndefPos);
        EAG.NodeBuf[EAG.NextNode] := EAG.NextMAlt;
        INC(EAG.NextNode); IF EAG.NextNode >= LEN(EAG.NodeBuf^) THEN EAG.Expand END;
        Rhs := EAG.NextMemb;
        EAG.AppMemb(nil);
        EAG.AppMemb(EAG.NewMAlt(HN - EAG.firstHNont + EAG.firstMNont, Rhs))
      END
    ELSE
      EAG.HNont[HN].Def := NIL; Sets.Incl(Del, HN);
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].Id := nil;
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].MRule := nil;
      EAG.MNont[HN - EAG.firstHNont + EAG.firstMNont].IsToken := FALSE
    END
  END;
  (* DomBuf *)
  EAG.NextDom := EAG.firstDom;
  FOR HN := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(GenNonts, HN) THEN
      EAG.HNont[HN].Sig := EAG.NextDom;
      EAG.AppDom("+", HN - EAG.firstHNont + EAG.firstMNont); EAG.AppDom("+", nil)
    END
  END;
  Sets.Difference(EAG.All, EAG.All, Del);
  Sets.Difference(EAG.Pred, EAG.Pred, Del);
  Sets.Difference(EAG.Prod, EAG.Prod, Del);
  Sets.Difference(EAG.Reach, EAG.Reach, Del);
  Sets.Difference(EAG.Null, EAG.Null, Del);
END Shift;

END eShift.
```

# Kapitel 8

# Abschließende Bewertung

Um eine Bewertung der generierten Compiler und damit des Epsilon-Compilergenerators zu ermöglichen, stellen wir an dieser Stelle einige konkrete Meßdaten für einen aktuellen PC mit Pentium-90 Prozessor vor.

Der Umfang des Epsilon-Compilergenerators ist mit 6500 Zeilen Quellcode, also etwa 80 Seiten, sehr kompakt ausgefallen. Aus der in Anhang C abgedruckten 14-seitigen Spezifikation eines Oberon-0-Compilers für einen RISC-Prozessor läßt sich in 1,5 Sekunden der 170 kB große Quellcode eines 1-Pass-Compilers in der Generatorzielsprache Oberon generieren, aus dem ein Compiler mit 140 kB Objektcode resultiert (single sweep-Compiler: 170 kB Objektcode).

Für die Kompilation eines 30-seitigen Oberon-0-Programms (43 kB), das geschachtelte Sortier- und Multiplikationsprozeduren enthält, benötigt der 1-Pass-Compiler eine Sekunde für die Berechnung der 180 kB großen Übersetzung und eine weitere Sekunde für deren Ausgabe (single sweep-Compiler: 3+1 Sekunden). Der dafür benötigte Speicherplatz ist von den Optimierungen abhängig, mit denen die Compiler generiert wurden, und folgender Tabelle zu entnehmen:

| Optimierungen | Speicherbedarf |
|---|---:|
| ohne #-Operator, ohne Optimierungen | 18000 kB |
| mit #-Operator, ohne Optimierungen | 3342 kB |
| mit #-Operator, mit Konstantenfaltung | 822 kB |
| mit #-Operator, mit Freispeicherverwaltung | 476 kB |
| mit #-Operator, mit beiden Optimierungen | 310 kB |

Beim single sweep-Verfahren werden zusätzlich 600 kB Speicher für die Repräsentation des Ableitungsbaums benötigt.

Der für diese Optimierungen notwendige Aufwand ist folglich berechtigt und führt zu mit praktisch eingesetzten Übersetzern bezüglich Laufzeit und Speicherbedarf vergleichbaren Compilern. Die nächstlohnende Optimierung wäre wahrscheinlich der Einsatz eines echten Scannergenerators. Dadurch könnte die Laufzeit noch weiter gesenkt werden, da Parameterberechnungen für Token nur einmal durchgeführt werden müßten. Wichtiger ist aber noch, daß die statischen Ableitungsbäume für Compiler mit mächtigeren Auswertungsverfahren deutlich kleiner würden.

# Anhang A

# Syntax der Eingabesprache Epsilon

Zur Beschreibung der mehrdeutigen Syntax von Epsilon-Spezifikationen wird ein erweiterter Bakkus-Naur-Formalismus (EBNF) verwendet.

Terminale werden durch Zeichenketten repräsentiert, die aus nichtleeren Zeichenfolgen bestehen, eingeschlossen in Anführungszeichen oder Apostrophe (wie in Modula-2). Terminale sind gleich, wenn ihre Zeichenfolgen gleich sind. Um einer irreführenden Fehlerbehandlung vorzubeugen, dürfen Zeichenketten nicht über das Zeilenende hinausgehen. Nichtterminale bestehen aus Folgen von Klein- und Großbuchstaben beliebiger Länge. Zahlen werden durch eine Ziffernfolge dargestellt und werden für eine Systemunabhängigkeit auf die Werte zwischen 0 und 9999 beschränkt.

Ein einfacher Kommentar beginnt (wie in Ada) mit einem Ausrufungszeichen (statt „-“ und erstreckt sich bis zum Zeilenende. Zur Ausblendung von Teilen der Spezifikation gibt es weiterhin Kommentare (wie in Modula-2), die aus einer öffnenden Klammer „(*“, gefolgt von einer beliebigen Zeichenfolge und der zugehörigen schließenden Klammer „*)“ bestehen. Diese Kommentare dürfen geschachtelt werden. Einfache Kommentare blenden Zeilen auch in geschachtelten Kommentare aus.

Die exakten Kombinationen von Parametrisierungen werden durch die Tabellen 2.1 und 2.2 erklärt.

In formalen Parametern kann abkürzend auf die Angabe des Wertebereichssymbols verzichtet werden, falls die Affixform aus einer Variablen zum Wertebereichssymbol besteht. Formale Parameter auf der linken Regelseite werden als abkürzende Schreibweise dafür erkannt, daß die Affixformen der formalen Parameter der nachfolgenden Alternativen identisch sind.

```
letter:  "a" | ...  | "z" | "A" | ...  | "Z".
digit:  "0" | ...  | "9".
ident:  letter {letter}.
number:  digit {digit}.
string:  "'" char {char} "'" | """ char {char} """.

MetaTerm:
  {ident| string}.
MetaExpr:
  MetaTerm {"|" MetaTerm}.
MetaRule:
  ident ["*"] "=" MetaExpr ".".
AffixForm:
  {["#"] ident [number] | string}.
ActualParams:
  "<" AffixForm {"," AffixForm} ">".
FormalParams:
  "<" ("+" | "-" ) AffixForm ":" ident
  {"," ("+" | "-" ) AffixForm ":" ident} ">".
HyperTerm:
  {ident [ActualParams] | string
  | [ActualParams]
    ( "(" | "[" | "{" ) HyperExpr ( ")" | "]" | "}" )
    [FormalParams] }.
HyperExpr:
  [FormalParams] HyperTerm [ActualParams]
  {"|" [FormalParams] HyperTerm [ActualParams] }.
HyperRule:
  ident ["*"] ":" HyperExpr "." .
Specification:
  (MetaRule | HyperRule) {MetaRule | HyperRule}.
```

# Anhang B

# Ein reduziertes Beispiel

Das nachfolgende Beispiel zeigt die Spezifikation einer einfachen Sprache sowie die Überprüfung einer typischen Kontextbedingung von Programmiersprachen. Daneben verdeutlicht es die oben definierte Syntax der Eingabesprache von Epsilon und zeigt eine übersichtliche Verwendung der EBNF-Operatoren. Zur Unterstützung der Lesbarkeit werden in den Spezifikationstexten die Bestandteile der Grundgrammatik fett gedruckt und Affixformen auf definierenden Positionen kursiv hervorgehoben.

Der aus der gezeigten EAG generierte Compiler akzeptiert als Eingabe eine Folge von Deklarationen und Applikationen von Bezeichnern, die ihrerseits aus Folgen der Buchstaben „`a`“ und „`b`“ bestehen. Jeder Bezeichner darf nur einmal deklariert und erst nach seiner Deklaration appliziert werden. Zur Überprüfung dieser Kontextbedingung wird auf der Meta-Ebene eine Symboltabelle aufgebaut, in der mit dem Prädikat `Find` nach Bezeichnern gesucht werden kann. Dieses Prädikat sucht den im ersten Parameter angegebenen Bezeichner in der im zweiten Parameter angegebenen Tabelle durch deren sukzessive Zerlegung in den jeweils ersten Eintrag und eine Resttabelle. Das Ergebnis der Suche wird im dritten Parameter zurückgereicht. Kontextfehler werden in der für das Nichtterminal `DeclAppl` erzeugten Prozedur durch Scheitern einer Analyse der Affixform „`FALSE`“ bzw. „`TRUE`“ entdeckt, die jeweils auf der dritten Parameterposition des Prädikataufrufs von `Find` steht. Zuletzt wird als „Übersetzung“ einer Eingabe die Symboltabelle ausgegeben.

```
! DeclAppl

Tab = | id ";" Tab.

DeclAppl <+ Tab: Tab>:
  <, Tab>
    { <- Tab: Tab, + Tab1: Tab>
        "DECL" id <id> Find <id, Tab, "FALSE">
        <id ";" Tab, Tab1>
    | <- Tab: Tab, + Tab1: Tab>
        "APPL" id <id> Find <id, Tab, "TRUE">
        <Tab, Tab1>
    } <- Tab: Tab, + Tab: Tab>.

x = "a" | "b".
id* = x | id x.

x <+ "a": x>: "a".
x <+ "b": x>: "b".

id* <+ id: id>:
  x <x>
  <x, id>
    { <- id: id, + id1: id>
        x <x>  <id x, id1>
    } <- id: id, + id: id>.

Bool = "TRUE" | "FALSE".

Find <- id: id, - : Tab, + "FALSE": Bool>: .
Find <- id: id, - id ";" Tab: Tab, + "TRUE": Bool>: .
Find <- id: id, - #id ";" Tab: Tab, + Bool: Bool>:
  Find <id, Tab, Bool>.
```

# Anhang C

# Spezifikation eines Oberon-0-Compilers

Zur Einführung in die „Grundlagen und Techniken des Übersetzerbaus“ präsentiert Wirth in seinem Lehrbuch [Wirth] die komplette Implementierung eines effizienten und kompakten Compilers. Die Quellsprache ist dabei eine substantielle Untermenge der Programmiersprache Oberon, und der hypothetischen Zielmaschine liegt das Konzept der RISC-Architekturen zugrunde. Die nachfolgende Spezifikation [Kröplin] ist eng an den von Hand geschriebenen Compiler angelehnt, um so exemplarisch einen Vergleich der beiden Ansätze zu ermöglichen. Allerdings wird hier zweckmäßig symbolischer Assemblercode erzeugt, der anschließend noch in das eigentliche Maschinenprogramm transformiert werden muß. Auf den erforderlichen Assemblierer kann dann auch die Berechnung arithmetischer Ausdrücke abgeschoben werden, die sonst im Compiler umständliche Prädikate erfordern würde. Die von Wirth für die Generierung angezweifelte Transparenz scheint uns wesentlich an den Stellen problematisch zu sein, die schon durch seinen Mustercompiler unsauber behandelt werden.

```
! Oberon-0 (Compiler)  MK 12.96

OberonO <+ Code>: Module <Code>.

letter =
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
  | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
  | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
  | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".

letter:
    <+ "a": letter> "a" | <+ "b": letter> "b" | <+ "c": letter> "c" | <+ "d": letter> "d"
  | <+ "e": letter> "e" | <+ "f": letter> "f" | <+ "g": letter> "g" | <+ "h": letter> "h"
  | <+ "i": letter> "i" | <+ "j": letter> "j" | <+ "k": letter> "k" | <+ "l": letter> "l"
  | <+ "m": letter> "m" | <+ "n": letter> "n" | <+ "o": letter> "o" | <+ "p": letter> "p"
  | <+ "q": letter> "q" | <+ "r": letter> "r" | <+ "s": letter> "s" | <+ "t": letter> "t"
  | <+ "u": letter> "u" | <+ "v": letter> "v" | <+ "w": letter> "w" | <+ "x": letter> "x"
  | <+ "y": letter> "y" | <+ "z": letter> "z"
  | <+ "A": letter> "A" | <+ "B": letter> "B" | <+ "C": letter> "C" | <+ "D": letter> "D"
  | <+ "E": letter> "E" | <+ "F": letter> "F" | <+ "G": letter> "G" | <+ "H": letter> "H"
  | <+ "I": letter> "I" | <+ "J": letter> "J" | <+ "K": letter> "K" | <+ "L": letter> "L"
  | <+ "M": letter> "M" | <+ "N": letter> "N" | <+ "O": letter> "O" | <+ "P": letter> "P"
  | <+ "Q": letter> "Q" | <+ "R": letter> "R" | <+ "S": letter> "S" | <+ "T": letter> "T"
  | <+ "U": letter> "U" | <+ "V": letter> "V" | <+ "W": letter> "W" | <+ "X": letter> "X"
  | <+ "Y": letter> "Y" | <+ "Z": letter> "Z".

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

digit:
    <+ "0": digit> "0" | <+ "1": digit> "1" | <+ "2": digit> "2" | <+ "3": digit> "3"
  | <+ "4": digit> "4" | <+ "5": digit> "5" | <+ "6": digit> "6" | <+ "7": digit> "7"
  | <+ "8": digit> "8" | <+ "9": digit> "9".

ident* = letter | ident letter | ident digit.

ident* <+ ident>:
  letter <letter>
  <letter, ident>
    { <- ident, + ident1>
        letter <letter>  <ident letter, ident1>
    | <- ident, + ident1>
        digit <digit>  <ident digit, ident1>
    } <- ident, + ident>.

N* = digit | N digit.

integer* <+ N>:
  digit <digit>
  <digit, N>
    { <- N, + N1>
        digit <digit>  <N digit, N1>
    } <- N, + N>.
```

```
Selector:
  { <- Table, - Item, - "RECORD" Scope: Type, + Item2, + Type2,
     - Code, - Regs, - N, + Code2, + Regs2, + N2>
      "." ident <ident> FindObject <Scope, ident, "FIELD" E Type1>
      Field <Item, E, Item1, Code, Regs, Code1, Regs1>
      <Table, Item1, Type1, Item2, Type2, Code1, Regs1, N, Code2, Regs2, N2>
  | <- Table, - Item, - "ARRAY" E1 "*" E2 Type2: Type, + Item3, + Type3,
     - Code, - Regs, - N, + Code3, + Regs3, + N3>
      "[" Expression <Table, Item1, "INT", Code, Regs, N, Code1, Regs1, N1> "]"
      Index <Item, E1, E2, Item1, Item2, Code1, Regs1, Code2, Regs2>
      <Table, Item2, Type2, Item3, Type3, Code2, Regs2, N1, Code3, Regs3, N3>
  } <- Table, - Item, - Type, + Item, + Type,
     - Code, - Regs, - N, + Code, + Regs, + N>.

Op =
    "+" | "-" | "*" | "DIV" | "MOD"
  | "NOT" | "OR" | "AND"
  | "=" | "#" | "<" | ">=" | ">" | "<=".

Factor:
    <- Table, + Item1, + Type1,
     - Code, - Regs, - N, + Code2, + Regs2, + N2>
      ident <ident> Find <Table, ident, Object, Lev>
      MakeItem <Object, Lev, Item, Type, Code, Regs, Code1, Regs1>
      Selector <Table, Item, Type, Item1, Type1, Code1, Regs1, N, Code2, Regs2, N2>
  | <- Table, + "CONST" N1: Item, + "INT": Type,
     - Code, - Regs, - N, + Code, + Regs, + N>
      integer <N1>
  | <- Table, + Item, + Type,
     - Code, - Regs, - N, + Code1, + Regs1, + N1>
      "(" Expression <Table, Item, Type, Code, Regs, N, Code1, Regs1, N1> ")"
  | <- Table, + Item1, + "BOOL": Type,
     - Code, - Regs, - N, + Code2, + Regs2, + N1>
      "~" Factor <Table, Item, "BOOL", Code, Regs, N, Code1, Regs1, N1>
      UnOp <"NOT", Item, Item1, Code1, Regs1, Code2, Regs2>.

Term <- Table, + Item1, + Type,
      - Code, - Regs, - N, + Code2, + Regs2, + N2>:
  Factor <Table, Item, Type, Code, Regs, N, Code1, Regs1, N1>
  <Table, Item, Type, Item1, Code1, Regs1, N1, Code2, Regs2, N2>
    { <- Table, - Item, - Type, + Item4,
       - Code, - Regs, - N, + Code4, + Regs4, + N4>
        <Item, Type, Op, Item1, Code, Regs, N, Code1, Regs1, N1>
          ( <- Item, - "INT": Type, + Op, + Item,
             - Code, - Regs, - N, + Code, + Regs, + N>
              <Op>
                ( <+ "*": Op> "*" | <+ "DIV": Op> "DIV" | <+ "MOD": Op> "MOD" )
          | <- Item, - "BOOL": Type, + "AND": Op, + Item1,
             - Code, - Regs, - N, + Code1, + Regs1, + N1>
              "&"
              CondJump <Item, Item1, Code, Regs, N, Code1, Regs1, N1>
          )
        Factor <Table, Item2, Type, Code1, Regs1, N1, Code2, Regs2, N2>
        Op <Op, Item1, Item2, Item3, Code2, Regs2, Code3, Regs3>
        <Table, Item3, Type, Item4, Code3, Regs3, N2, Code4, Regs4, N4>
    } <- Table, - Item, - Type, + Item,
       - Code, - Regs, - N, + Code, + Regs, + N>.
```

```
Sign = "+" | "-" | .

SimpleExpression <- Table, + Item2, + Type,
                    - Code, - Regs, - N, + Code3, + Regs3, + N3>:
  <Sign>
    [ <+ "+": Sign> "+" | <+ "-": Sign> "-" ] <+ : Sign>
  Term <Table, Item, Type, Item1, Code1, Regs1, Code2, Regs2>
  <Sign, Item, Type, Item1, Code1, Regs1, Code2, Regs2>
    [ <- "+": Sign, - Item, - "INT": Type, + Item,
       - Code, - Regs, + Code, + Regs>
    | <- "-": Sign, - Item, - "INT": Type, + Item1,
       - Code, - Regs, + Code1, + Regs1>
        UnOp <"-", Item, Item1, Code, Regs, Code1, Regs1>
    ] <- : Sign, - Item, - Type, + Item,
       - Code, - Regs, + Code, + Regs>
  <Table, Item1, Type, Item2, Code2, Regs2, N1, Code3, Regs3, N3>
    { <- Table, - Item, - Type, + Item4,
       - Code, - Regs, - N, + Code4, + Regs4, + N4>
        <Item, Type, Op, Item1, Code, Regs, N, Code1, Regs1, N1>
          ( <- Item, - "INT": Type, + Op, + Item,
             - Code, - Regs, - N, + Code, + Regs, + N>
              <Op>
                ( <+ "+": Op> "+" | <+ "-": Op> "-" )
          | <- Item, - "BOOL": Type, + "OR": Op, + Item2,
             - Code, - Regs, - N, + Code2, + Regs2, + N2>
              "OR" UnOp <"NOT", Item, Item1, Code, Regs, Code1, Regs1>
              CondJump <Item1, Item2, Code1, Regs1, N, Code2, Regs2, N2>
          )
        Term <Table, Item2, Type, Code1, Regs1, N1, Code2, Regs2, N2>
        Op <Op, Item1, Item2, Item3, Code2, Regs2, Code3, Regs3>
        <Table, Item3, Type, Item4, Code3, Regs3, N2, Code4, Regs4, N4>
    } <- Table, - Item, - Type, + Item,
       - Code, - Regs, - N, + Code, + Regs, + N>.

Expression <- Table, + Item1, + Type1,
              - Code, - Regs, - N, + Code2, + Regs2, + N2>:
  SimpleExpression <Table, Item, Type, Code, Regs, N, Code1, Regs1, N1>
  <Table, Item, Type, Item1, Type1, Code1, Regs1, N1, Code2, Regs2, N2>
    [ <- Table, - Item1, - "INT": Type, + Item3, + "BOOL": Type,
       - Code, - Regs, - N, + Code2, + Regs2, + N1>
        <Op>
          ( <+ "=": Op> "=" | <+ "#": Op> "#"
          | <+ "<": Op> "<" | <+ ">=": Op> ">="
          | <+ ">": Op> ">" | <+ "<=": Op> "<="
          )
        SimpleExpression <Table, Item2, "INT", Code, Regs, N, Code1, Regs1, N1>
        Relation <Op, Item1, Item2, Item3, Code1, Regs1, Code2, Regs2>
    ] <- Table, - Item, - Type, + Item, + Type,
       - Code, - Regs, - N, + Code, + Regs, + N>.

ActualParameters:
  [ <- Table, - Parameters, + Item,
     - Code, - Regs, - N, + Code1, + Regs1, + N1>
      "("
      <Table, Parameters, Item, Code, Regs, N, Code1, Regs1, N1>
        [ <- Table, - Kind Type Parameters: Parameters, + Item,
           - Code, - Regs, - N, + Code3, + Regs3, + N3>
            Expression <Table, Item, Type, Code, Regs, N, Code1, Regs1, N1>
```

```
            Parameter <Kind, Item, Code1, Regs1, Code2, Regs2>
            <Table, Parameters, Code2, Regs2, N1, Code3, Regs3, N3>
              { <- Table, - Kind Type Parameters: Parameters,
                 - Code, - Regs, - N, + Code3, + Regs3, + N3>
                  "," Expression <Table, Item, Type, Code, Regs, N, Code1, Regs1, N1>
                  Parameter <Kind, Item, Code1, Regs1, Code2, Regs2>
                  <Table, Parameters, Code2, Regs2, N1, Code3, Regs3, N3>
              } <- Table, - : Parameters,
                 - Code, - Regs, - N, + Code, + Regs, + N>
       ] <- Table, - : Parameters, + "VAR" "0" "," "0": Item,
          - Code, - Regs, - N, + Code, + Regs, + N>
     ")"
  ] <- Table, - : Parameters, + "VAR" "0" "," "0": Item,
     - Code, - Regs, - N, + Code, + Regs, + N>.

AssignmentOrProcedureCall <- Table, - N, + Code, + N1>:
  ident <ident> Find <Table, ident, Object, Lev>
  <Table, Object, Lev, N, Code, N1>
    ( <- Table, - Object, - Lev, - N, + Code4, + N3>
        available <Regs>
        MakeItem <Object, Lev, Item, Type, , Regs, Code1, Regs1>
        Selector <Table, Item, Type, Item1, BasicType, Code1, Regs1, N, Code2, Regs2, N2>
        ":=" Expression <Table, Item2, BasicType, Code2, Regs2, N2, Code3, Regs3, N3>
        Store <Item1, Item2, Code3, Regs3, Code4, Regs4>
    | <- Table, - Proc Parameters: Object, - Lev, - N, + Code2, + N1>
        available <Regs>
        ActualParameters <Table, Parameters, Item, , Regs, N, Code1, Regs1, N1>
        <Proc, Item, Code1, Regs1, Code2, Regs2>
          ( <- "LPROC" E: Proc, - Item,
             - Code, - Regs, + Code Code1: Code, + Regs>
              Call <E, Code1>
          | <- SProc: Proc, - Item,
             - Code, - Regs, + Code1, + Regs1>
              IOCall <SProc, Item, Code, Regs, Code1, Regs1>
          )
    ).

Condition <- Table, + Item1, - N, + Code2, + N2>:
  available <Regs>
  Expression <Table, Item, "BOOL", , Regs, N, Code1, Regs1, N1>
  CondJump <Item, Item1, Code1, Regs1, N1, Code2, Regs2, N2>.

IfStatement <- Table, - N, + Code4 Code5: Code, + N4>:
  Label <E, N, Code5, N5>
  "IF" Condition <Table, Item, N5, Code1, N1>
  "THEN" StatementSequence <Table, N1, Code2, N2>
  <Table, E, Item, Item1, Code1 Code2, N2, Code3, N3>
    { <- Table, - E, - Item, + Item2, - Code, - N, + Code5, + N5>
        Jump <E, Code1> Fix <Item, Code2>
        "ELSIF" Condition <Table, Item1, N, Code3, N3>
        "THEN" StatementSequence <Table, N3, Code4, N4>
        <Table, E, Item1, Item2, Code Code1 Code2 Code3 Code4, N4, Code5, N5>
    } <- Table, - E, - Item, + Item, - Code, - N, + Code, + N>
```

```
  <Table, E, Item1, Code3, N3, Code4, N4>
    ( <- Table, - E, - Item, - Code, - N, + Code Code1 Code2 Code3: Code, + N3>
        Jump <E, Code1> Fix <Item, Code2>
        "ELSE" StatementSequence <Table, N, Code3, N3>
    | <- Table, - E, - Item, - Code, - N, + Code Code1: Code, + N>
        Fix <Item, Code1>
    )
  "END".

WhileStatement <- Table, - N, + Code1 Code2 Code3 Code4 Code5: Code, + N3>:
  Label <E, N, Code1, N1>
  "WHILE" Condition <Table, Item, N1, Code2, N2>
  "DO" StatementSequence <Table, N2, Code3, N3> Jump <E, Code4>
  "END" Fix <Item, Code5>.

Statement <- Table, - N, + Code, + N1>:
    AssignmentOrProcedureCall <Table, N, Code, N1>
  | IfStatement <Table, N, Code, N1>
  | WhileStatement <Table, N, Code, N1>
  | <N, Code, N1>
      ( <- N, + : Code, + N> ).

StatementSequence <- Table, - N, + Code2, + N2>:
  Statement <Table, N, Code1, N1>
  <Table, Code1, N1, Code2, N2>
    { <- Table, - Code1, - N1, + Code3, + N3>
        ";" Statement <Table, N1, Code2, N2>
        <Table, Code1 Code2, N2, Code3, N3>
    } <- Table, - Code, - N, + Code, + N>.

IdentList = | ident ";" IdentList.

IdentList <+ ident ";" IdentList: IdentList, - Scope, + Scope1>:
  ident <ident> FindObject <Scope, ident, "NIL">
  <IdentList, Scope ident "UNDEF", Scope1>
    { <+ ident ";" IdentList: IdentList, - Scope, + Scope1>
        "," ident <ident> FindObject <Scope, ident, "NIL">
        <IdentList, Scope ident "UNDEF", Scope1>
    } <+ : IdentList, - Scope, + Scope>.

BasicType = "INT" | "BOOL".
Type = BasicType | "ARRAY" E "*" E Type | "RECORD" Scope.

ArrayType <- Table, + "ARRAY" E1 "*" E2 Type: Type, + "(" E1 "*" E2 ")": E>:
  available <Regs>
  "ARRAY" Expression <Table, "CONST" E1, "INT", , Regs, "0", Code1, Regs1, N1>
  "OF" Type <Table, Type, E2>.

FieldList:
  [ <- Table, - Scope, - E, + Scope2, + E2>
      IdentList <IdentList, Scope, Scope1>
      ":" Type <Table "UNDEF" Scope1, Type, E1>
      <IdentList, Type, E1, Scope, E, Scope2, E2>
        { <- ident ";" IdentList: IdentList, - Type, - E1, - Scope, - E, + Scope2, + E2>
           <IdentList, Type, E1, Scope ident "FIELD" E Type, "(" E "+" E1 ")", Scope2, E2>
        } <- : IdentList, - Type, - E1, - Scope, - E, + Scope, + E>
  ] <- Table, - Scope, - E, + Scope, + E>.
```

```
RecordType <- Table, + "RECORD" Scope2: Type, + E2>:
  "RECORD" FieldList <Table, , "0", Scope1, E1>
  <Table, Scope1, E1, Scope2, E2>
    { <- Table, - Scope, - E, + Scope2, + E2>
        ";" FieldList <Table, Scope, E, Scope1, E1>
        <Table, Scope1, E1, Scope2, E2>
    } <- Table, - Scope, - E, + Scope, + E>
  "END".

Type <- Table, + Type, + E>:
    ident <ident> Find <Table, ident, "TYPE" E Type, Lev>
  | ArrayType <Table, Type, E>
  | RecordType <Table, Type, E>.

Kind = "VAR" | "REF" | "PAR".
SProc = "READ" | "WRITE" | "WRITEHEX" | "WRITELN".
Proc = "LPROC" E | SProc.
Parameters = | Kind Type Parameters.
Object =
    "UNDEF"
  | "CONST" E Type | "TYPE" E Type | "FIELD" E Type | Kind E Type | Proc Parameters.
Scope = | Scope ident Object.

ObjNil = Object | "NIL".

FindObject:
    <- : Scope, - ident, + "NIL": ObjNil>
  | <- Scope ident Object: Scope, - ident, + Object: ObjNil>
  | <- Scope #ident Object: Scope, - ident, + ObjNil: ObjNil>
      FindObject <Scope, ident, ObjNil>.

Insert:
  { <- ident ";" IdentList: IdentList, - Kind, - E1, - Type, - E2,
     - Scope, - E, + Scope3, + E3>
      <IdentList, Kind, E1, Type, E2,
       Scope ident Kind "(" E1 "-" "(" E "+" E2 ")" ")" Type, "(" E "+" E2 ")", Scope3, E3>
  } <- : IdentList, - Kind, - E1, - Type, - E2,
     - Scope, - E, + Scope, + E>.

Lev = "GLOBAL" | "UNDEF" | "LOCAL".
Table = | Table Lev Scope.

Find <- Table Lev Scope: Table, - ident, + Object, + Lev1>:
  FindObject <Scope, ident, ObjNil>
  <Table, ident, ObjNil, Lev, Object, Lev1>
    { <- Table Lev Scope: Table, - ident, - "NIL": ObjNil, - Lev3, + Object, + Lev2>
        FindObject <Scope, ident, ObjNil>
        <Lev, Lev1>
          ( <- "GLOBAL": Lev, + "GLOBAL": Lev>
          | <- "UNDEF": Lev, + "UNDEF": Lev>
          | <- "LOCAL": Lev, + "UNDEF": Lev>
          )
        <Table, ident, ObjNil, Lev1, Object, Lev2>
    } <- Table, - ident, - Object: ObjNil, - Lev, + Object, + Lev>.
```

```
FPSection <- Table, - Scope, - E, + Scope2, + E2>:
  <Kind>
    [ <+ "REF": Kind> "VAR" ] <+ "VAR": Kind>
  IdentList <IdentList, Scope, Scope1>
  ":" ident <ident> Find <Table "UNDEF" Scope1, ident, "TYPE" E1 BasicType, Lev>
  Insert <IdentList, Kind, "0", BasicType, E1, Scope, E, Scope2, E2>.

FormalParameters:
  [ <- Table, + Scope, + E, + Parameters>
      "("
      <Table,Scope, E, Parameters>
        [ <- Table, + Scope3, + E2, + Parameters>
            FPSection <Table, , "0", Scope1, E1>
            <Table, Scope1, E1, Scope2, E2>
              { <- Table, - Scope, - E, + Scope2, + E2>
                  ";" FPSection <Table, Scope, E, Scope1, E1>
                  <Table, Scope1, E1, Scope2, E2>
              } <- Table, - Scope, - E, + Scope, + E>
            markSize <E3>
            <"(" E2 "+" E3 ")", Scope2, Scope3, , Parameters>
              { <- E,
                 - Scope ident Kind E1 Type: Scope,
                 + Scope1 ident Kind "(" E "+" E1 ")" Type: Scope,
                 - Parameters, + Parameters1>
                  <E, Scope, Scope1, Kind Type Parameters, Parameters1>
              } <- E, - : Scope, + : Scope, - Parameters, + Parameters>
        ] <- Table, + : Scope, + "0": E, + : Parameters>
      ")"
  ] <- Table, + : Scope, + "0": E, + : Parameters>.

Body <- Table, - N, + Code, + N1>:
  <Table, N, Code, N1>
    [ <- Table, - N, + Code, + N1>
        "BEGIN" StatementSequence <Table, N, Code, N1>
    ] <- Table, - N, + : Code, + N>
  "END".

ProcedureDeclaration <- Table, - Lev, - Scope, + Scope ident "LPROC" E Parameters: Scope,
                        - N, + Code1 Code2 Code3 Code4 Code5: Code, + N4>:
  Label <E, N, Code2, N2>
  "PROCEDURE" ident <ident> FindObject <Scope, ident, "NIL">
  FormalParameters <Table Lev Scope, Scope1, E1, Parameters>
  ";"
  Declarations <Table Lev Scope ident "LPROC" E Parameters, "LOCAL", Scope1, Table1, E2,
                N2, Code1, N1>
  Enter <E2, Code3>
  Body <Table1, N1, Code4, N4> ident <ident>
  Return <E1, Code5>.

Declarations <- Table, - Lev, - Scope, + Table Lev Scope4: Table, + E, - N, + Code1, + N1>:
  <Table, Scope, Scope1>
    [ <- Table, - Scope, + Scope1>
        "CONST"
        <Table, Scope, Scope1>
          { <- Table, - Scope, + Scope1>
              ident <ident> FindObject <Scope, ident, "NIL">
              available <Regs>
```

```
            "=" Expression <Table "UNDEF" Scope ident "UNDEF", "CONST" E, "INT",
                           , Regs, "0", Code1, Regs1, N1>
            ";"
            <Table, Scope ident "CONST" E "INT", Scope1>
        } <- Table, - Scope, + Scope>
    ] <- Table, - Scope, + Scope>
  <Table, Scope1, Scope2>
    [ <- Table, - Scope, + Scope1>
        "TYPE"
        <Table, Scope, Scope1>
          { <- Table, - Scope, + Scope1>
            ident <ident> FindObject <Scope, ident, "NIL">
            "=" Type <Table "UNDEF" Scope ident "UNDEF", Type, E>
            ";"
            <Table, Scope ident "TYPE" E Type, Scope1>
        } <- Table, - Scope, + Scope>
    ] <- Table, - Scope, + Scope>
  <Table, Lev, Scope2, Scope3, E>
    [ <- Table, - Lev, - Scope, + Scope1, + E1>
        <Lev, E>
          ( <- "GLOBAL": Lev, + E>
            memSize <E>
          | <- "LOCAL": Lev, + "0": E>
          )
        "VAR"
        <Table, E, Scope, "0", Scope1, E1>
          { <- Table, - E1, - Scope, - E, + Scope3, + E3>
            IdentList <IdentList, Scope, Scope1>
            ":" Type <Table "UNDEF" Scope1, Type, E4>
            Insert <IdentList, "VAR", E1, Type, E4, Scope, E, Scope2, E2>
            ";"
            <Table, E1, Scope2, E2, Scope3, E3>
        } <- Table, - E1, - Scope, - E, + Scope, + E>
    ] <- Table, - Lev, - Scope, + Scope, + "0": E>
  <Table, Lev, Scope3, Scope4, , N, Code1, N1>
    { <- Table, - Lev, - Scope, + Scope2, - Code, - N, + Code2, + N2>
        ProcedureDeclaration <Table, Lev, Scope, Scope1, N, Code1, N1>
        ";"
        <Table, Lev, Scope1, Scope2, Code Code1, N1, Code2, N2>
    } <- Table, - Lev, - Scope, + Scope, - Code, - N, + Code, + N>.

standard <+ "F" "A" "L" "S" "E"              "CONST" "0" "BOOL"
            "T" "R" "U" "E"                  "CONST" "1" "BOOL"
            "I" "N" "T" "E" "G" "E" "R"      "TYPE" "1" "INT"
            "B" "O" "O" "L" "E" "A" "N"      "TYPE" "1" "BOOL"
            "R" "e" "a" "d"                  "READ"     "PAR" "INT"
            "W" "r" "i" "t" "e"              "WRITE"    "PAR" "INT"
            "W" "r" "i" "t" "e" "H" "e" "x"  "WRITEHEX" "PAR" "INT"
            "W" "r" "i" "t" "e" "L" "n"      "WRITELN"
          : Scope>: .

Module <+ Code1 Code2 Code3 Code4: Code>:
  standard <Scope>
  "MODULE" ident <ident>
  ";" Declarations <"UNDEF" Scope, "GLOBAL", , Table, E, "1", Code1, N1>
  Header <E, Code2>
  Body <Table, N1, Code3, N3> ident <ident>
  "." close <Code4>.
```

```
! RISC-Assembler

label* = "L" N.
Ls = | Ls Ls
  | label.

Inc:
    <- N digit: N, + N digit1: N>
      Inc <digit, digit1>
  | <- N "9": N, + N1 "0": N>
      Inc <N, N1>
  | <- "0": N, + "1": N>
  | <- "1": N, + "2": N>
  | <- "2": N, + "3": N>
  | <- "3": N, + "4": N>
  | <- "4": N, + "5": N>
  | <- "5": N, + "6": N>
  | <- "6": N, + "7": N>
  | <- "7": N, + "8": N>
  | <- "8": N, + "9": N>
  | <- "9": N, + "1" "0": N>.

E =
    N | label | "PC"
  | "(" E "+" E ")" | "(" E "-" E ")" | "(" E "*" E ")" | "(" E "/" E ")" | "(" E "%" E ")".

eqO <- "0": E>: .

uneqO <- E>:
  <#E>
    ( <+ "0": E> ).

CC = "BEQ" | "BNE" | "BLT" | "BGE" | "BGT" | "BLE".

Negated:
    <- "BEQ": CC, + "BNE": CC> | <- "BNE": CC, + "BEQ": CC>
  | <- "BLT": CC, + "BGE": CC> | <- "BGE": CC, + "BLT": CC>
  | <- "BGT": CC, + "BLE": CC> | <- "BLE": CC, + "BGT": CC>.

Code = | Code Code
  | label "*" | Ls !
  | "ADD"  N "," N "," N
  | "SUB"  N "," N "," N
  | "MUL"  N "," N "," N
  | "DIV"  N "," N "," N
  | "MOD"  N "," N "," N
  | "CMP"  N "," N "," N
  | "ADDI" N "," N "," E
  | "SUBI" N "," N "," E
  | "MULI" N "," N "," E
  | "DIVI" N "," N "," E
  | "MODI" N "," N "," E
  | "CMPI" N "," N "," E
  | "CHKI" N "," E
  | "LDW"  N "," N "," E
  | "STW"  N "," N "," E
  | "POP"  N "," N "," E
  | "PSH"  N "," N "," E
```

```
  |  CC    N "," E
  | "BSR"  E
  | "RET"  N
  | "RD"   N
  | "WD"   N
  | "WH"   N
  | "WL"   N.
```

**! Registerzuteilung**

```
Regs = | N ";" Regs.

rO  <+     "0": N>: .

fp  <+ "2" "9": N>: .

sp  <+ "3" "0": N>: .

lnk <+ "3" "1": N>: .

available <+     "1" ";"     "2" ";"     "3" ";"     "4" ";"     "5" ";"
                 "6" ";"     "7" ";"     "8" ";"     "9" ";" "1" "0" ";"
             "1" "1" ";" "1" "2" ";" "1" "3" ";" "1" "4" ";" "1" "5" ";"
             "1" "6" ";" "1" "7" ";" "1" "8" ";" "1" "9" ";" "2" "0" ";"
             "2" "1" ";" "2" "2" ";" "2" "3" ";" "2" "4" ";" "2" "5" ";"
             "2" "6" ";" "2" "7" ";" "2" "8" ";"
           : Regs>: .

FreeReg:
    <- N, - Regs, + Regs>
      rO <N>
  | <- N, - Regs, + Regs>
      fp <N>
  | <- N, - Regs, + N ";" Regs: Regs>
      <N, N>
        ( <- #N1, - #N2>
            rO <N1> fp <N2>
        ).
```

**! Code-Auswahl**

```
Item = "VAR" N "," E | "REG" N | "CONST" E | "COND" CC N Ls "," Ls | "PEND" Ls.

MakeItem:
    <- "CONST" E Type: Object, - Lev, + "CONST" E: Item, + Type,
     - Code, - Regs, + Code, + Regs>
  | <- "VAR" E Type: Object, - "GLOBAL": Lev, + "VAR" N "," E: Item, + Type,
     - Code, - Regs, + Code, + Regs>
      rO <N>
  | <- "VAR" E Type: Object, - "LOCAL": Lev, + "VAR" N "," E: Item, + Type,
     - Code, - Regs, + Code, + Regs>
      fp <N>
  | <- "REF" E Type: Object, - "LOCAL": Lev, + "VAR" N1 "," "0": Item, + Type,
     - Code, - N1 ";" Regs: Regs, + Code "LDW" N1 "," N "," E: Code, + Regs>
      fp <N>.
```

```
MakeCond <- Op, - N, + "COND" CC N ",": Item>:
  <Op, CC>
    ( <- "=": Op, + "BEQ": CC> | <- "#": Op, + "BNE": CC>
    | <- "<": Op, + "BLT": CC> | <- ">=": Op, + "BGE": CC>
    | <- ">": Op, + "BGT": CC> | <- "<=": Op, + "BLE": CC>
    ).

Load:
    <- "VAR" N "," E: Item, + "REG" N1: Item,
     - Code, - Regs, + Code "LDW" N1 "," N "," E: Code, + Regs1>
      FreeReg <N, Regs, N1 ";" Regs1>
  | <- "REG" N: Item, + "REG" N: Item,
     - Code, - Regs, + Code, + Regs>
  | <- "CONST" E: Item, + "REG" N1: Item,
     - Code, - N1 ";" Regs: Regs, + Code "ADDI" N1 "," N "," E: Code, + Regs>
      uneqO <E>
      rO <N>
  | <- "CONST" E: Item, + "REG" N: Item,
     - Code, - Regs, + Code, + Regs>
      eqO <E>
      rO <N>
  | <- "COND" CC N1 Ls1 "," Ls2: Item, + "REG" N2: Item,
     - Code, - Regs, + Code  CC    N1 "," "3"
                         Ls1 "ADDI" N2 "," N "," "0"
                             "BEQ"  N "," "2"
                         Ls2 "ADDI" N2 "," N "," "1": Code, + Regs1>
      rO <N> FreeReg <N1, Regs, N2 ";" Regs1>.

LoadBool:
    <- "REG" N: Item, + Item>
      MakeCond <"#", N, Item>
  | <- "CONST" E: Item, + Item>
      uneqO <E>
      rO <N> MakeCond <"=", N, Item>
  | <- "CONST" E: Item, + Item>
      eqO <E>
      rO <N> MakeCond <"#", N, Item>
  | <- "COND" CC N Ls1 "," Ls2: Item, + "COND" CC N Ls1 "," Ls2: Item>.

Field <- "VAR" N "," E: Item, - E1, + "VAR" N "," "(" E "+" E1 ")": Item,
       - Code, - Regs, + Code, + Regs>: .

Index:
    <- Item, - E1, - E2, - "VAR" N "," E: Item, + Item2,
     - Code, - Regs, + Code2, + Regs2>
      Load <"VAR" N "," E, Item1, Code, Regs, Code1, Regs1>
      Index <Item, E1, E2, Item1, Item2, Code1, Regs1, Code2, Regs2>
  | <- "VAR" N "," E: Item, - E1, - E2, - "REG" N1: Item, + "VAR" N2 "," E: Item,
     - Code, - Regs, + Code "CHKI" N1 "," E1
                           "MULI" N2 "," N1 "," E2: Code, + Regs1>
      rO <N>
      FreeReg <N1, Regs, N2 ";" Regs1>
  | <- "VAR" N "," E: Item, - E1, - E2, - "REG" N1: Item, + "VAR" N2 "," E: Item,
     - Code, - Regs, + Code "CHKI" N1 "," E1
                           "MULI" N2 "," N1 "," E2
                           "ADD"  N2 "," N "," N2: Code, + Regs2>
      rO <#N>
      FreeReg <N1, Regs, N2 ";" Regs1> FreeReg <N, Regs1, Regs2>
```

```
  | <- "VAR" N "," E: Item, - E1, - E2, - "CONST" E3: Item,
     + "VAR" N "," "(" E "+" "(" E2 "*" E3 ")" ")": Item,
     - Code, - Regs, + Code, + Regs>.

CondJump:
    <- "VAR" N1 "," E: Item, + Item2,
     - Code, - Regs, - N, + Code2, + Regs2, + N2>
      Load <"VAR" N1 "," E, Item1, Code, Regs, Code1, Regs1>
      CondJump <Item1, Item2, Code1, Regs1, N, Code2, Regs2, N2>
  | <- Item, + "PEND" Ls1 "L" N: Item,
     - Code, - Regs, - N, + Code  CC1 N2 "," "(" "L" N "-" "PC" ")"
                            Ls2: Code, + Regs1, + N1>
      LoadBool <Item, "COND" CC N2 Ls1 "," Ls2> Negated <CC, CC1>
      FreeReg <N2, Regs, Regs1> Inc <N, N1>.

Fix <- "PEND" Ls: Item, + Ls: Code>: .

UnOp:
    <- Op, - "VAR" N "," E: Item, + Item2,
     - Code, - Regs, + Code2, + Regs2>
      Load <"VAR" N "," E, Item1, Code, Regs, Code1, Regs1>
      UnOp <Op, Item1, Item2, Code1, Regs1, Code2, Regs2>
  | <- "-": Op, - "REG" N1: Item, + "REG" N2: Item,
     - Code, - Regs, + Code "SUB" N2 "," N "," N1: Code, + Regs1>
      rO <N> FreeReg <N1, Regs, N2 ";" Regs1>
  | <- "-": Op, - "CONST" E: Item, + "CONST" "(" "0" "-" E ")": Item,
     - Code, - Regs, + Code, + Regs>
  | <- "NOT": Op, - Item, + "COND" CC1 N Ls2 "," Ls1: Item,
     - Code, - Regs, + Code, + Regs>
      LoadBool <Item, "COND" CC N Ls1 "," Ls2> Negated <CC, CC1>.

Op:
    <- Op, - Item1, - "VAR" N "," E: Item, + Item3,
     - Code, - Regs, + Code2, + Regs2>
      Load <"VAR" N "," E, Item2, Code, Regs, Code1, Regs1>
      Op <Op, Item1, Item2, Item3, Code1, Regs1, Code2, Regs2>
  | <- Op, - Item, - "REG" N2: Item, + "REG" N3: Item,
     - Code, - Regs, + Code1 Code2: Code, + Regs3>
      Load <Item, "REG" N1, Code, Regs, Code1, Regs1>
      FreeReg <N2, Regs1, Regs2> FreeReg <N1, Regs2, N3 ";" Regs3>
      <Op, N3, N1, N2, Code2>
        ( <- "+": Op, - N, - N1, - N2, + "ADD" N "," N1 "," N2: Code>
        | <- "-": Op, - N, - N1, - N2, + "SUB" N "," N1 "," N2: Code>
        | <- "*": Op, - N, - N1, - N2, + "MUL" N "," N1 "," N2: Code>
        | <- "DIV": Op, - N, - N1, - N2, + "DIV" N "," N1 "," N2: Code>
        | <- "MOD": Op, - N, - N1, - N2, + "MOD" N "," N1 "," N2: Code>
        )
  | <- Op, - Item, - "CONST" E: Item, + "REG" N2: Item,
     - Code, - Regs, + Code1 Code2: Code, + Regs2>
      <Item, "REG" N1, Code, Regs, Code1, Regs1>
        ( <- "VAR" N "," E: Item, + Item,
           - Code, - Regs, + Code1, + Regs1>
            Load <"VAR" N "," E, Item, Code, Regs, Code1, Regs1>
        | <- "REG" N: Item, + "REG" N: Item,
           - Code, - Regs, + Code, + Regs>
        )
      FreeReg <N1, Regs1, N2 ";" Regs2>
      <Op, N2, N1, E, Code2>
```

```
        ( <- "+": Op, - N, - N1, - E, + "ADDI" N "," N1 "," E: Code>
        | <- "-": Op, - N, - N1, - E, + "SUBI" N "," N1 "," E: Code>
        | <- "*": Op, - N, - N1, - E, + "MULI" N "," N1 "," E: Code>
        | <- "DIV": Op, - N, - N1, - E, + "DIVI" N "," N1 "," E: Code>
        | <- "MOD": Op, - N, - N1, - E, + "MODI" N "," N1 "," E: Code>
        )
  | <- Op, - "CONST" E1: Item, - "CONST" E2: Item, + "CONST" E3: Item,
     - Code, - Regs, + Code, + Regs>
      <Op, E1, E2, E3>
        ( <- "+": Op, - E1, - E2, + "(" E1 "+" E2 ")": E>
        | <- "-": Op, - E1, - E2, + "(" E1 "-" E2 ")": E>
        | <- "*": Op, - E1, - E2, + "(" E1 "*" E2 ")": E>
        | <- "DIV": Op, - E1, - E2, + "(" E1 "/" E2 ")": E>
        | <- "MOD": Op, - E1, - E2, + "(" E1 "%" E2 ")": E>
        )
  | <- "OR": Op, - "PEND" Ls: Item, - Item, + "COND" CC N Ls1 "," Ls Ls2: Item,
     - Code, - Regs, + Code, + Regs>
      LoadBool <Item, "COND" CC N Ls1 "," Ls2>
  | <- "AND": Op, - "PEND" Ls: Item, - Item, + "COND" CC N Ls Ls1 "," Ls2: Item,
     - Code, - Regs, + Code, + Regs>
      LoadBool <Item, "COND" CC N Ls1 "," Ls2>.

Relation:
    <- Op, - Item1, - "VAR" N "," E: Item, + Item3,
     - Code, - Regs, + Code2, + Regs2>
      Load <"VAR" N "," E, Item2, Code, Regs, Code1, Regs1>
      Relation <Op, Item1, Item2, Item3, Code1, Regs1, Code2, Regs2>
  | <- Op, - Item1, - "REG" N2: Item, + Item,
     - Code, - Regs, + Code1 "CMP" N3 "," N1 "," N2: Code, + Regs3>
      Load <Item1, "REG" N1, Code, Regs, Code1, Regs1>
      FreeReg <N2, Regs1, Regs2> FreeReg <N1, Regs2, N3 ";" Regs3>
      MakeCond <Op, N3, Item>
  | <- Op, - Item1, - "CONST" E: Item, + Item,
     - Code, - Regs, + Code1 "CMPI" N2 "," N1 "," E: Code, + Regs2>
      uneqO <E>
      Load <Item1, "REG" N1, Code, Regs, Code1, Regs1>
      FreeReg <N1, Regs1, N2 ";" Regs2>
      MakeCond <Op, N2, Item>
  | <- Op, - Item1, - "CONST" E: Item, + Item,
     - Code, - Regs, + Code1, + Regs1>
      eqO <E>
      Load <Item1, "REG" N, Code, Regs, Code1, Regs1>
      MakeCond <Op, N, Item>.

Parameter:
    <- "VAR": Kind, - Item,
     - Code, - Regs, + Code1 "PSH" N1 "," N "," "1": Code, + Regs2>
      Load <Item, "REG" N1, Code, Regs, Code1, Regs1>
      sp <N> FreeReg <N1, Regs1, Regs2>
  | <- "REF": Kind, - "VAR" N1 "," E: Item,
     - Code, - Regs, + Code "ADDI" N2 "," N1 "," E
                             "PSH"  N2 "," N "," "1": Code, + N2 ";" Regs1: Regs>
      uneqO <E>
      sp <N> FreeReg <N1, Regs, N2 ";" Regs1>
  | <- "REF": Kind, - "VAR" N1 "," E: Item,
     - Code, - Regs, + Code "PSH"  N1 "," N "," "1": Code, + Regs1>
      eqO <E>
      sp <N> FreeReg <N1, Regs, Regs1>
```

```
  | <- "PAR": Kind, - Item,
     - Code, - Regs, + Code, + Regs>.

Store <- "VAR" N "," E: Item, - Item,
       - Code, - Regs, + Code1 "STW" N1 "," N "," E: Code, + Regs3>:
  Load <Item, "REG" N1, Code, Regs, Code1, Regs1>
  FreeReg <N1, Regs1, Regs2> FreeReg <N, Regs2, Regs3>.

Label <+ "L" N: E, - N, + "L" N: Code, + N1>:
  Inc <N, N1>.

Jump <- E, + "BEQ" N "," "(" E "-" "PC" ")": Code>:
  rO <N>.

Call <- E, + "BSR" "(" E "-" "PC" ")": Code>: .

IOCall:
    <- "READ": SProc, - Item,
     - Code, - N ";" Regs: Regs, + Code1, + Regs1>
      Store <Item, "REG" N, Code "RD" N, Regs, Code1, Regs1>
  | <- "WRITE": SProc, - Item,
     - Code, - Regs, + Code1 "WD" N: Code, + Regs2>
      Load <Item, "REG" N, Code, Regs, Code1, Regs1>
      FreeReg <N, Regs1, Regs2>
  | <- "WRITEHEX": SProc, - Item,
     - Code, - Regs, + Code1 "WH" N: Code, + Regs2>
      Load <Item, "REG" N, Code, Regs, Code1, Regs1>
      FreeReg <N, Regs1, Regs2>
  | <- "WRITELN": SProc, - Item,
     - Code, - Regs, + Code "WL" N: Code, + Regs>
      rO <N>.

memSize <+ "1" "0" "2" "4": E>: .

Header <- E, + "L" "0" "*" "ADDI" N2 "," N "," "(" E1 "-" E ")": Code>:
  rO <N> sp <N2> memSize <E1>.

close <+ "RET" N: Code>:
  rO <N>.

markSize <+ "2": E>: .

Enter <- E, + "PSH"  N3 "," N2 "," "1"
              "PSH"  N1 "," N2 "," "1"
              "ADD"  N1 "," N "," N2
              "SUBI" N2 "," N2 "," E: Code>:
  rO <N> fp <N1> sp <N2> lnk <N3>.

Return <- E, + "ADD" N2 "," N "," N1
               "POP" N1 "," N2 "," "1"
               "POP" N3 "," N2 "," "(" E "+" "1" ")"
               "RET" N3: Code>:
  rO <N> fp <N1> sp <N2> lnk <N3>.
```

# Anhang D

# Ungleichheit im reinen Kalkül

Im reinen Kalkül der Zweistufengrammatiken (ohne Ungleichheit #) ist die naive Formulierung der Ungleichheit gerade einzelner Buchstaben theoretisch zwar trivial, aber praktisch verbietet sich das Auflisten von Hunderten von Regeln für die in der Anzahl der Buchstaben quadratisch vielen Fälle. Die einfache Idee, daß nur verschiedene Buchstaben nacheinander aus dem Alphabet entfernt werden können, führt systematisch auf ein Hilfsprädikat zum Löschen eines Zeichens aus einer Zeichenfolge. Wenn dieses jedoch allgemein als Funktion formuliert werden soll, ist beispielsweise für die Festlegung, das erste Vorkommen aus Zeichenfolgen mit Wiederholung zu löschen, schon die Ungleichheit von Zeichen erforderlich; allerdings „funktioniert“ hier für die tatsächlichen Aufrufe die Einschränkung auf Zeichenfolgen ohne Wiederholung. Werden in der Spezifikation des Oberon-0-Compilers zur Überprüfung der Ungleichheit von Bezeichnern und Zahlen die nachfolgend angegebenen Prädikate eingesetzt, so bedingt dies für die dauernde Bereitstellung der Zeichenfolgen ohne Optimierung terminaler Affixformen eine enorme Speicherplatzverschwendung, und in jedem Fall wird der generierte Compiler erheblich gebremst.

```
! unequal

letgit = letter | digit.
letgits = | letgit letgits.

excl:
    <- letgit letgits: letgits, - letgit, + letgits>
  | <- letgit1 letgits: letgits, - letgit, + letgit1 letgits1: letgits>
      excl <letgits, letgit, letgits1>.

ident* = letter | ident letgit.

uneqID:
    <- ident1 letgit1: ident, - ident2 letgit2: ident>
      uneqID <ident1, ident2>
  | <- ident1 letgit1: ident, - ident2 letgit2: ident>
      excl <"a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
            "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
            "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
            "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
            "0" "1" "2" "3" "4" "5" "6" "7" "8" "9",
                    letgit1, letgits1>
      excl <letgits1, letgit2, letgits2>
  | <- ident letgit: ident, - letter: ident>
  | <- letter: ident, - ident letgit: ident>
  | <- letter1: ident, - letter2: ident>
      excl <"a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
            "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
            "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
            "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z",
                    letter1, letgits1>
      excl <letgits1, letter2, letgits2>.

N* = digit | N digit.

uneqN:
    <- N1 digit1: N, - N2 digit2: N>
      uneqN <N1, N2>
  | <- N1 digit1: N, - N2 digit2: N>
      uneqN <digit1, digit2>
  | <- N digit1: N, - digit2: N>
  | <- digit1: N, - N digit2: N>
  | <- digit1: N, - digit2: N>
      excl <"0" "1" "2" "3" "4" "5" "6" "7" "8" "9",
                    digit1, letgits1>
      excl <letgits1, digit2, letgits2>.
```

# Anhang E

# Spezifikation eines Eta-nach-Epsilon-Konverters

Um vorhandene Eta-Spezifikationen nicht von Hand in die neue Spezifikationssprache übertragen zu müssen, wird ein entsprechendes Werkzeug benötigt, das selbst als einfacher Compiler angesehen werden kann. Für die nachfolgende Spezifikation unverzichtbar ist die Sonderbehandlung der Token-Nichtterminale im Parsergenerator, damit auch nur durch Zwischenraum getrennte Bezeichner richtig erkannt werden. Da bei der Transformation außer der Syntaxanalyse fast keine Überprüfung stattfindet und anschließend Meldungen sich auf die erzeugte Spezifikation beziehen, spielt hier die Lesbarkeit der ausgegebenen Übersetzung eine zentrale Rolle.

```
! Eta (Konverter)  MK 08.96

Eta <+ Spec>:
  Specification <Spec>.

letter =
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
  | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
  | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
  | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".

letter:
    <+ "a": letter> "a" | <+ "b": letter> "b" | <+ "c": letter> "c" | <+ "d": letter> "d"
  | <+ "e": letter> "e" | <+ "f": letter> "f" | <+ "g": letter> "g" | <+ "h": letter> "h"
  | <+ "i": letter> "i" | <+ "j": letter> "j" | <+ "k": letter> "k" | <+ "l": letter> "l"
  | <+ "m": letter> "m" | <+ "n": letter> "n" | <+ "o": letter> "o" | <+ "p": letter> "p"
  | <+ "q": letter> "q" | <+ "r": letter> "r" | <+ "s": letter> "s" | <+ "t": letter> "t"
  | <+ "u": letter> "u" | <+ "v": letter> "v" | <+ "w": letter> "w" | <+ "x": letter> "x"
  | <+ "y": letter> "y" | <+ "z": letter> "z".

LETTER:
    <+ "A": letter> "A" | <+ "B": letter> "B" | <+ "C": letter> "C" | <+ "D": letter> "D"
  | <+ "E": letter> "E" | <+ "F": letter> "F" | <+ "G": letter> "G" | <+ "H": letter> "H"
  | <+ "I": letter> "I" | <+ "J": letter> "J" | <+ "K": letter> "K" | <+ "L": letter> "L"
  | <+ "M": letter> "M" | <+ "N": letter> "N" | <+ "O": letter> "O" | <+ "P": letter> "P"
  | <+ "Q": letter> "Q" | <+ "R": letter> "R" | <+ "S": letter> "S" | <+ "T": letter> "T"
  | <+ "U": letter> "U" | <+ "V": letter> "V" | <+ "W": letter> "W" | <+ "X": letter> "X"
  | <+ "Y": letter> "Y" | <+ "Z": letter> "Z".

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

digit:
    <+ "0": digit> "0" | <+ "1": digit> "1" | <+ "2": digit> "2" | <+ "3": digit> "3"
  | <+ "4": digit> "4" | <+ "5": digit> "5" | <+ "6": digit> "6" | <+ "7": digit> "7"
  | <+ "8": digit> "8" | <+ "9": digit> "9".

char = letter | digit | "'"
  | " " | "!" | '"' | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" | ","
  | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "\\"
  | "]" | "^" | "_" | "`" | "{" | "|" | "}" | "~".

char:
    <+ letter: char>
      letter <letter>
  | <+ letter: char>
      LETTER <letter>
  | <+ digit: char>
      digit <digit>
  | <+ " ": char> " " | <+ "!": char> "!" | <+ '"': char> '"' | <+ "#": char> "#"
  | <+ "$": char> "$" | <+ "%": char> "%" | <+ "&": char> "&" | <+ "(": char> "("
  | <+ ")": char> ")" | <+ "*": char> "*" | <+ "+": char> "+" | <+ ",": char> ","
  | <+ "-": char> "-" | <+ ".": char> "." | <+ "/": char> "/" | <+ ":": char> ":"
  | <+ ";": char> ";" | <+ "<": char> "<" | <+ "=": char> "=" | <+ ">": char> ">"
  | <+ "?": char> "?" | <+ "@": char> "@" | <+ "[": char> "[" | <+ "\\": char> "\"
  | <+ "]": char> "]" | <+ "^": char> "^" | <+ "_": char> "_" | <+ "`": char> "`"
  | <+ "{": char> "{" | <+ "|": char> "|" | <+ "}": char> "}" | <+ "~": char> "~".
```

```
chars = | char chars.

ident* <+ letter chars: chars>:
  letter <letter>
  <chars>
    { <+ letter chars: chars>
        letter <letter>  <chars>
    | <+ digit chars: chars>
        digit <digit>  <chars>
    | <+ "_" chars: chars>
        "_"  <chars>
    } <+ : chars>.

ident* = letter | ident letter.
leftIdent* = "\n" ident.

IDENT* <+ ident>:
  LETTER <letter>
  <letter, ident>
    { <- ident1, + ident>
        LETTER <letter>  <ident1 letter, ident>
    } <- ident, + ident>.

digits = | digit digits.

number* <+ digit digits: digits>:
  digit <digit>
  <digits>
    { <+ digit digits: digits>
        digit <digit>  <digits>
    } <+ : digits>.

string* = "'" chars "'" | '"' chars '"'.

string* <+ string>:
  "'"
  <string>
    ( <+ "'" char chars "'": string>
        char <char>
        <chars>
          { <+ char chars: chars>
              char <char>  <chars>
          } <+ : chars>
    | <+ '"' "'" '"': string>
        "'" "'"
    )
  "'".

MetaT:
    <+ '"' chars '"': string>
      ident <chars>
  | <+ string>
      string <string>.

MetaN <+ ident, + digits>:
  IDENT <ident>
  <digits>
```

```
    [ <+ digits>
        number <digits>
    ] <+ : digits>.

HyperT <+ string>:
  string <string>.

HyperN <+ ident>:
  ident <letter chars>
  <letter, chars, ident>
    { <- ident1, - letter chars: chars, + ident>
        <ident1 letter, chars, ident>
    | <- ident1, - "0" chars: chars, + ident>
        <ident1 "O" "h", chars, ident>
    | <- ident1, - "1" chars: chars, + ident>
        <ident1 "O" "n" "e", chars, ident>
    | <- ident1, - "2" chars: chars, + ident>
        <ident1 "T" "w" "o", chars, ident>
    | <- ident1, - "3" chars: chars, + ident>
        <ident1 "T" "h" "r" "e" "e", chars, ident>
    | <- ident1, - "4" chars: chars, + ident>
        <ident1 "F" "o" "u" "r", chars, ident>
    | <- ident1, - "5" chars: chars, + ident>
        <ident1 "F" "i" "v" "e", chars, ident>
    | <- ident1, - "6" chars: chars, + ident>
        <ident1 "S" "i" "x", chars, ident>
    | <- ident1, - "7" chars: chars, + ident>
        <ident1 "S" "e" "v" "e" "n", chars, ident>
    | <- ident1, - "8" chars: chars, + ident>
        <ident1 "E" "i" "g" "h" "t", chars, ident>
    | <- ident1, - "9" chars: chars, + ident>
        <ident1 "N" "i" "n" "e", chars, ident>
    | <- ident1, - "_" chars: chars, + ident>
        <ident1 "X", chars, ident>
    } <- ident, - : chars, + ident>.

Dir = "+" | "-".
Signature = "NIL" | Dir ident Signature.

FormalParameter <+ Dir, + ident>:
  <Dir>
    ( <+ "+": Dir> "+"
    | <+ "-": Dir> "-"
    )
  MetaN <ident, digits>.

Definition <+ ident, + Signature>:
  HyperN <ident>
  <Signature>
    [ <+ Dir ident Signature: Signature>
        "(" FormalParameter <Dir, ident>
        <Signature>
          { <+ Dir ident Signature: Signature>
              "," FormalParameter <Dir, ident>
              <Signature>
          } <+ "NIL": Signature>
        ")"
    ] <+ "NIL": Signature>.
```

```
Kind = "ROOT" | "NONT" | "PRED".
Table = | Kind ident Signature Table.

Find:
    <- Kind ident Signature Table: Table, + Kind, - ident, + Signature>
  | <- Kind1 #ident Signature1 Table: Table, + Kind, - ident, + Signature>
      Find <Table, Kind, ident, Signature>.

Heading <+ Table>:
  HyperN <ident1> "(" "+" MetaN <ident, digits> ")"
  <"ROOT" ident1 "+" ident "NIL", Table1>
    { <- Table1, + Table>
        Definition <ident, Signature>
        <"NONT" ident Signature Table1, Table>
    } <- Table, + Table>
  <Table1, Table>
    [ <- Table1, + Table>
        ";" Definition <ident, Signature>
        <"PRED" ident Signature Table1, Table>
          { <- Table1, + Table>
              Definition <ident, Signature>
              <"PRED" ident Signature Table1, Table>
          } <- Table, + Table>
    ] <- Table, + Table>
  ".".
Bool = "TRUE" | "FALSE".

and:
    <- "TRUE": Bool, - Bool, + Bool>
  | <- "FALSE": Bool, - Bool, + "FALSE": Bool>.

MetaTerm = | MetaTerm MetaTerm
  | string
  | ident.
MetaExpr = MetaExpr "\n " "|" MetaExpr
  | MetaTerm .
Enum = Enum "|" Enum |
  MetaTerm.
MetaRule =
    leftIdent "=" "\n   " MetaExpr "."
  | leftIdent "=" "\n " Enum ".".

MetaAlt:
  [ <+ MetaTerm1 MetaTerm2: MetaTerm, + Bool>
      <MetaTerm1, Bool1>
        ( <+ string: MetaTerm, + "TRUE": Bool>
            MetaT <string>
        | <+ ident: MetaTerm, + "FALSE": Bool>
            MetaN <ident, digits>
        )
      <MetaTerm2, Bool2>
        { <+ string MetaTerm: MetaTerm, + "FALSE": Bool>
            MetaT <string>
            <MetaTerm, Bool>
        | <+ ident MetaTerm: MetaTerm, + "FALSE": Bool>
            MetaN <ident, digits>
            <MetaTerm, Bool>
        } <+ : MetaTerm, + "TRUE": Bool>
```

```
      and <Bool1, Bool2, Bool>
  ] <+ : MetaTerm, + "FALSE": Bool>.

MetaRule <+ MetaRule>:
  MetaN <ident, digits> ":"
  MetaAlt <MetaTerm, Bool1>
  <MetaTerm, MetaTerm, Bool1, MetaExpr, Enum, Bool>
    { <- MetaExpr1, - Enum1, - Bool1, + MetaExpr, + Enum, + Bool>
        ";" MetaAlt <MetaTerm, Bool2>
        and <Bool1, Bool2, Bool3>
        <MetaExpr1 "\n " "|" MetaTerm, Enum1 "|" MetaTerm, Bool3, MetaExpr, Enum, Bool>
    } <- MetaExpr, - Enum, - Bool, + MetaExpr, + Enum, + Bool>
  <ident, MetaExpr, Enum, Bool, MetaRule>
    ( <- ident, - MetaExpr, - Enum, - "TRUE": Bool,
       + "\n" ident "=" "\n " Enum ".": MetaRule>
    | <- ident, - MetaExpr, - Enum, - "FALSE": Bool,
       + "\n" ident "=" "\n   " MetaExpr ".": MetaRule>
    )
  ".".

Var* = ident digits.
AffixForm = | AffixForm AffixForm
  | string
  | Var.

AffixForm:
  [ <+ AffixForm1 AffixForm2: AffixForm>
      <AffixForm1>
        ( <+ string: AffixForm>
            MetaT <string>
        | <+ ident digits: AffixForm>
            MetaN <ident, digits>
        )
      <AffixForm2>
        { <+ string AffixForm: AffixForm>
            MetaT <string>
            <AffixForm>
        | <+ ident digits AffixForm: AffixForm>
            MetaN <ident, digits>
            <AffixForm>
        } <+ : AffixForm>
  ] <+ : AffixForm>.

FPList = FPList "," FPList
  | Dir AffixForm ":" ident.
FormalParams = | "<" FPList ">".

LeftHyperNotion <- Table, + Kind, + ident, + FormalParams>:
  HyperN <ident>
  Find <Table, Kind, ident, Signature>
  <Signature, FormalParams>
    [ <- Dir ident Signature: Signature, + "<" FPList ">": FormalParams>
        "(" AffixForm <AffixForm>
        <Dir AffixForm ":" ident, Signature, FPList>
          { <- FPList1, - Dir ident Signature: Signature, + FPList>
              "," AffixForm <AffixForm>
              <FPList1 "," Dir AffixForm ":" ident, Signature, FPList>
          } <- FPList, - "NIL": Signature, + FPList>
```

```
        ")"
    ] <- "NIL": Signature, + : FormalParams>.

APList = APList "," APList
  | AffixForm.
ActualParams = | "<" APList ">".

RightHyperNotion <+ ident, + ActualParams>:
  HyperN <ident>
  <ActualParams>
    [ <+ "<" APList ">": ActualParams>
        "(" AffixForm <AffixForm>
        <AffixForm, APList>
          { <- APList1, + APList>
              "," AffixForm <AffixForm>
              <APList1 "," AffixForm, APList>
          } <- APList, + APList>
        ")"
    ] <+ : ActualParams>.

HyperFactor =
    string
  | ident ActualParams.
HyperTerm = | "\n " HyperFactor HyperTerm.
HyperExpr =
    HyperTerm
  | string.
HyperRule = leftIdent FormalParams ":" HyperExpr ".".

HyperRule <- Table, + Kind, + "\n" ident FormalParams ":" HyperExpr ".": HyperRule>:
  LeftHyperNotion <Table, Kind, ident, FormalParams> ":"
  <HyperExpr>
    [ <+ HyperExpr>
        <HyperFactor>
          ( <+ string: HyperFactor>
              HyperT <string>
          | <+ ident ActualParams: HyperFactor>
              RightHyperNotion <ident, ActualParams>
          )
        <HyperTerm>
          { <+ "\n " string HyperTerm: HyperTerm>
              HyperT <string>
              <HyperTerm>
          | <+ "\n " ident ActualParams HyperTerm: HyperTerm>
              RightHyperNotion <ident, ActualParams>
              <HyperTerm>
          } <+ : HyperTerm>
        <HyperFactor, HyperTerm, HyperExpr>
          ( <- string: HyperFactor, - : HyperTerm,
             + string: HyperExpr>
          | <- ident ActualParams: HyperFactor, - : HyperTerm,
             + "\n " ident ActualParams: HyperExpr>
          | <- HyperFactor1, - "\n " HyperFactor HyperTerm: HyperTerm,
             + "\n " HyperFactor1 "\n " HyperFactor HyperTerm: HyperExpr>
          )
    ] <+ : HyperExpr>
  ".".
```

```
Spec = | Spec Spec
  | MetaRule
  | HyperRule.

append:
    <- "ROOT": Kind, - HyperRule, - Spec1, + Spec1 HyperRule: Spec, - Spec2, + Spec2>
  | <- "NONT": Kind, - HyperRule, - Spec1, + Spec1, - Spec2, + Spec2 HyperRule: Spec>
  | <- "PRED": Kind, - HyperRule, - Spec1, + Spec1, - Spec2, + Spec2 HyperRule: Spec>.

Specification <+ Spec1 Spec2 Spec3: Spec>:
  Heading <Table>
  <Spec1>
    ( <+ MetaRule Spec: Spec>
        MetaRule <MetaRule>
        <Spec>
          { <+ MetaRule Spec: Spec>
              MetaRule <MetaRule>
              <Spec>
          } <+ : Spec>
    )
  <Table, Spec2, Spec3>
    ( <- Table, + Spec2, + Spec4>
        HyperRule <Table, Kind, HyperRule>
        append <Kind, HyperRule, , Spec1, , Spec3>
        <Table, Spec1, Spec2, Spec3, Spec4>
          { <- Table, - Spec1, + Spec3, - Spec4, + Spec6>
              HyperRule <Table, Kind, HyperRule>
              append <Kind, HyperRule, Spec1, Spec2, Spec4, Spec5>
              <Table, Spec2, Spec3, Spec5, Spec6>
          } <- Table, - Spec1, + Spec1, - Spec2, + Spec2>
    ).
```

# Anhang F

# Benutzung des Epsilon-Compilergenerators

Die Generierung eines Compilers erfolgt in mehreren Schritten, die durch Aufruf entsprechender Kommandos eingeleitet werden. Ein Ein-Pass-Compiler wird auf folgende Weise erstellt:

1. Internalisierung einer Spezifikation
2. Generierung des Scanners
3. Bestimmung der Prädikate
4. Test der SLEAG-Auswertbarkeit
5. Generierung des zum Ein-Pass-Compiler erweiterten Parsers

Soll ein Compiler mit separatem single sweep-Evaluator erstellt werden, so entfallen die Schritte 4 und 5. Stattdessen wird zuerst der Evaluator generiert, anschließend der vorgeschaltete Parser.

Die Namen der einzelnen Kommandos sowie die jeweils möglichen Optionen sind nachstehendem Abdruck des auch elektronisch zur Verfügung stehenden Tool-Files zu entnehmen.

```
Epsilon
Compiler-Generator for Oberon Version 1.02

Edit.Open DeclAppl.Eps

eAnalyser.Analyse *
eScanGen.Generate
ePredicates.Check
eSLEAGGen.Test  eELL1Gen.Test
eELL1Gen.Generate
eSSweep.Test  eSSweep.Generate  eELL1Gen.GenerateParser

Edit.Open DeclAppl.Mod

DeclAppl.Compile *
```

```
eErrorElems.Insert ^  eErrorElems.Remove
eErrorElems.Next      eErrorElems.Repair

Options:
  of the Epsilon compiler generator (generation commands):
    -c: disable collapsing constant trees
    -r: disable reference counting in generated compiler
    -m: modules are shown, not compiled directly
    -p: parser ignores regular token marks at hypernonterminals
    -w: open new window with compilation output as default
    -s: generated compiler uses a space instead of a newline
           as separator in compilation output
  of the generated compilers:
    -i: show heap usage information
    -v: verbose parser error messages
    -w: toggle default value for opening window

Commands of the Epsilon compiler generator:
eAnalyser.Analyse ( * | @ | ^ | filename )
  internalizes an Epsilon specification. All further commands use this
  internalized version. Errors are reported. eAnalyser.Warnings shows
  warnings.
eScanGen.Generate [ -m ]
  generates a scanner.
ePredicates.Check
  computes the predicates in the specification. ePredicates.List shows
  them. Needed by all further commands.
eSLEAGGen.Test
  checks for SLEAG evaluability. Needed for a one pass compiler.
eELL1Gen.Test [ -p ]
  checks for ELL(1) parsability.
eELL1Gen.Generate  [ -crmpws ]
  generates complete one pass compiler. Consists of ELL(1) parser and
  SLEAG evaluator. Needs a scanner.
eELL1Gen.GenerateParser  [ -mp ]
  generates a separate parser for a compiler with a
  single sweep evaluator. Needs a scanner.
eSSweep.Test
  checks for single sweep evaluability.
eSSweep.Generate [ -crmws ]
  generates a single sweep evaluator. Needs a separate parser.
  This command must be executed before eELL1Gen.GenerateParser
  (restriction of implementation).

eSplit.Split [-num] [-m] ( * | @ | ^ | filename )
  splits generated module into num MOD 100 smaller modules plus
  num DIV 100 basemodules. Needed for large modules, that can't
  be compiled with the standard Oberon compiler (error 210).

eErrorElems.Insert [ ^ ]
  reads specified error messages of a generated compiler and inserts
  eErrorElems into the marked text.
eErrorElems.Remove
  removes all eErrorElems from the marked text.
```

```
eErrorElems.Next
  shows the next eErrorElem in the marked text and sets the caret.
eErrorElems.Repair
  repairs parser defined syntax errors in the marked text with
  eErrorElems inserted. Requires compiler option -v (verbose).

Commands of the generated compilers:
Name.Compile [-ivw ] ( * | @ | ^ | filename )
  compiles the specified input.
Name.Reset
  frees allocated heap space.
```

Hier ein Protokoll der Generierung eines Ein-Pass-Compilers aus obiger Beispiel-Spezifikation sowie der anschließenden Compilierung einer kleinen Eingabe mit dem generierten Compiler:

```
Analysing ...       DeclAppl   ok
ScanGen writing DeclAppl    new symbol file   5304
Predicates in     DeclAppl:   1      ePredicates.List
Predicates in     DeclAppl:
  pos   1797 :   Find
SLEAG testing   DeclAppl   ok
ELL(1) writing   DeclAppl   +rc +ct  new symbol file   9516

DeclAppl compiler (generated with Epsilon)
DeclAppl compiler: compiling...
ba ; ab ; b ; a ;
```

# Anhang G

# Demonstration der Fehlerbehandlung

Um die Güte der Fehlererkennung, des Wiederaufsetzverhaltens und der Reparatur aufzuzeigen, geben wir hier die Fehlermeldungen in übersichtlicher Form wider, die der in Kapitel 8 angesprochene Oberon-0-Compiler bei der Übersetzung eines reichlich fehlerhaften Testprogramms ausgibt. Die Meldungstexte sind vollständig generiert und entsprechen daher nicht der gewohnten Form. Zu beachten ist, daß aufgrund der Fehlerreparatur parallel zu den Syntaxfehlern auch Kontextfehler gemeldet werden können.

```
MODULE Error;
CONST M := 10, N = 100 X = 10;
        ^  ^ ^ ^       ^
        1  2 3 4       5

VAR , a, b, c;
    ^ ^      ^
    6 7      8

PROCEDURE P;
BEGIN
  s := 0; a = 5 * (b - 1 END;
    ^       ^     ^      ^  ^
    9      10    11     12 13

BEGIN
  >  a  >  b;
  ^  ^   ^  ^
 14 15 16 17

  WHILE a DO
    BEGIN > b;  -  c := 0;
    ^       ^^   ^  ^
   18     19 20 21 22
```

```
   WHILE a > 0 BEGIN
               ^
             23

     IF ODD [a c := c * - b;
     ^       ^  ^         ^
    24     25 26       27

     b := 2 * b; a := a *2
   END;
   P := 0; P; 666;
     ^         ^  ^
    28       29 30

 END .
    ^
    31


1   syntax error, expected:  =
2       restart point
        symbol inserted:  =
3   syntax error, expected:  ;
4       restart point
        symbol inserted:  ;
5   syntax error, expected:  * DIV MOD &
        restart point
        symbol inserted:  ;
6   syntax error, expected:  a b ...  y z A B ...  Y Z
7       restart point
8   syntax error, expected:  ,
        restart point
        symbol inserted:  :
        symbol inserted:  a
    analysis in 'Type' failed
9   predicate 'Find' failed
10  syntax error, expected:  .  [ ( :=
11      restart point
    analysis in 'AssignmentOrProcedureCall' failed
12  syntax error, expected:  ,
        restart point
        symbol inserted:  )
13  syntax error, expected:  a b ...  y z A B ...  Y Z
        restart point
        symbol inserted:  a
    'ident' failed in 'ProcedureDeclaration'
14  syntax error, expected:  a b ...  y z A B ...  Y Z IF WHILE REPEAT
15      restart point
16  syntax error, expected:  .  [ ( :=
17      restart point
        symbol inserted:  :=
18  syntax error, expected:  a b ...  y z A B ...  Y Z IF WHILE REPEAT
19      restart point
```

```
20  analysis in 'AssignmentOrProcedureCall' failed
21  syntax error, expected:  a b ...  y z A B ...  Y Z IF WHILE REPEAT
22      restart point
23  syntax error, expected:  * DIV MOD &
24      restart point
        symbol inserted:  DO
25  predicate 'Find' failed
26  syntax error, expected:  .  [
        restart point
        symbol inserted:  ]
        symbol inserted:  THEN
27  syntax error, expected:  a b ...  y z A B ...  Y Z 0 1 ...  9 ( ~
        restart point
        symbol inserted:  a
28  predicate in 'AssignmentOrProcedureCall' failed
29  syntax error, expected:  a b ...  y z A B ...  Y Z IF WHILE REPEAT
30      restart point
31  syntax error, expected:  ;
        restart point
        symbol inserted:  END
        symbol inserted:  END
        symbol inserted:  a
    'ident' failed in 'Module'

    25 errors detected
```

Das fehlerhafte Programm wurde während der Parsierung in das nachstehende syntaktisch korrekte Programm transformiert, auf das sich die gemeldeten Kontextfehler beziehen:

```
MODULE Error;
CONST M  = 10 ; N = 100  ; X = 10;
VAR a, b, c : a ;

PROCEDURE P;
BEGIN
  s := 0; a (b - 1  ) END a ;

BEGIN
  a  := b;
  WHILE a DO
    b; c := 0;
    WHILE a > 0  DO IF ODD [a  ]  THEN c := c *  a - b;
      b := 2 * b; a := a *2
    END;
    P := 0; P; ;
  END  END  END  a .
```

# Literaturverzeichnis


[Earley] J. Earley:
An efficient context-free parsing algorithm
*Communications of the ACM* **13**: 94–102 (1970)

[Grosch] J. Grosch:
Efficient and Comfortable Error Recovery in Recursive Descent Parsers
*Structured Programming* **11**: 129–140 (1990)

[Kröplin] M. Kröplin:
Eta nach Epsilon (Teil II): Spezifikation eines Oberon-0-Compilers
Vorträge im Forschungskolloquium am 5. und 18. Dezember 1996,
Fachgebiet PC, Fachbereich Informatik, TU Berlin

[Reiser] M. Reiser:
*The Oberon System: User Guide and Programmer's Manual*
Addison-Wesley 1991

[ReiWi] M. Reiser, N. Wirth:
*Programming in Oberon: Steps beyond Pascal and Modula*
Addison-Wesley 1992

[Röhrich] J. Röhrich:
Methods for the Automatic Construction of Error Correcting Parsers
*Acta Informatica* **13**: 115–139 (1980)

[Schröer] F. W. Schröer:
*Eta: Ein Compiler-Generator auf der Basis zweistufiger Grammatiken*
Bericht Nr. 84-2, Fachbereich Informatik, TU Berlin, März 1984

[Watt] D. A. Watt:
*Analysis Oriented Two Level Grammars*
Ph. D. thesis, Glasgow 1974

[Wirth] N. Wirth:
*Grundlagen und Techniken des Compilerbaus*
Addison-Wesley 1996

[WiGu] N. Wirth, J. Gutknecht:
*Project Oberon – The Design of an Operating System and Compiler*
Addison-Wesley 1992

[Zimmermann] B. Zimmermann, K. Voßloh:
Compiler-Generierung II: Spezifikationskalküle und Implementierungskonzepte
Skript einer Lehrveranstaltung an der TU Berlin im WS 1994/95