ÜBERSETZERBAU

Manfred Brockhaus
Anton Ertl
Alexander Forst-Rakoczy

Institut für Computersprachen
Abteilung für Übersetzerbau

Vorwort

Dieses Skriptum behandelt grundlegende Begriffe und Methoden des Compilerbaus (Übersetzerbau). Der Inhalt entspricht dem Stoff der Vorlesung. Wir setzen Kenntnisse in höheren Programmiersprachen und Assembler voraus. Grundlegende Kenntnisse über Automaten und Formale Sprachen sind für das Verständnis von Vorteil.

Der Compilerbau ist ein klassisches Gebiet der Informatik. Er gehört seit den Anfängen der Informatik bzw. Computer Science zu den Kernfächern und hat seine Aktualität bis heute nicht verloren. Dafür sind unter anderem folgende Gründe maßgebend:

- Compiler verarbeiten formale Sprachen und können daher weitgehend formal spezifiziert werden. Dabei lassen sich Ergebnisse aus der Theorie der Automaten und Formalen Sprachen praktisch anwenden. Dies hat insbesondere zur Entwicklung von Compiler-Generatoren geführt. Die Methoden und Werkzeuge des Compilerbaus sind aber nicht nur zur Konstruktion von Compilern, sondern auch zur Implementierung von Editoren, Textsystemen, Netzwerkprotokollen etc. anwendbar. Ganz allgemein eignen sie sich zur Entwicklung von Programmen, deren Operationen stark an die Struktur der Eingabedaten gebunden sind. Solche Programme können analog zu Compilern formal spezifiziert und generiert werden, indem man die Eingabe als Sprache und die Verarbeitung als Semantik der Sprache auffaßt.

- Ein Compiler ist ein Programm, das über andere Programme „nachdenkt", d.h. diese analysiert und transformiert. Die Methoden des Compilerbaus kann man daher auch einsetzen, um universelle Programme in spezialisierte und effizientere Programme zu transformieren, sofern ein Teil der Eingabedaten bekannt ist. Diese Technik der „partiellen Auswertung" ist im Bereich der funktionalen und logischen Sprachen von aktueller Bedeutung.

- Bei neuen Rechnerarchitekturen geht man zunehmend davon aus, daß der Compiler die Erzeugung des Maschinencodes genau auf die Anforderungen der Hardware abstimmt, und zwar nicht nur im Hinblick auf die Effizienz, sondern auch im Hinblick auf den korrekten zeitlichen (parallelen) Ablauf der Befehle. Es handelt sich also um integrierte Hardware/Software-Entwicklungen mit aktuellen Anforderungen an den Compilerbau.

Das Skriptum ist wie folgt aufgebaut: Im Kapitel 1 werden die Grundbegriffe Compiler und Interpreter definiert. Kapitel 2 gibt einen Überblick über die Teilaufgaben eines Compilers und zeigt an einem einfachen Beispiel, wie ein Programm schrittweise übersetzt wird. Die restlichen Kapitel behandeln die einzelnen Teilaufgaben des Compilers. Jedes Kapitel behandelt eine Teilaufgabe, wobei die Folge der Kapitel der logischen Folge der Teilaufgaben entspricht.

Algorithmen werden in einer Java-ähnlichen Notation angegeben. Übergeordnete Klassendefinition oder Befehle zum Importieren von *Packages* werden weggelassen.

Eine umfassende Behandlung der globalen Optimierung und der automatisierten Code-Erzeugung (Code-Generator-Generatoren) ist im gegebenen Rahmen nicht möglich. Wir verweisen auf entsprechende Speziallehrveranstaltungen und die wissenschaftliche Literatur.

Der Anhang des Skriptums enthält zwei verschiedene Implementationen von Übersetzer-Interpreter-Systemen für die Programmersprache Mini, eine kleine Teilmenge von Java/C. Sie demonstrieren typische Formen von Zwischensprachen und Interpretern. Die erste Version ist in Java programmiert. Als Zwischensprache wird ein Postfix-Code verwendet, der einer Teilmenge des standardisierten Java VM Codes entspricht. Der Interpreter simuliert eine Stackmaschine und entspricht einer Java VM. Die zweite Version ist in Prolog mit DCGs programmiert. Als Zwischensprache wird die abstrakte Syntax verwendet und der Interpreter arbeitet rekursiv.

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Compiler

Ein **Compiler** (*translator*) oder **Übersetzer** ist ein Programm, das Programme einer **Quellsprache** (*source language*) in semantisch äquivalente Programme einer **Zielsprache** (*target language*) umwandelt.

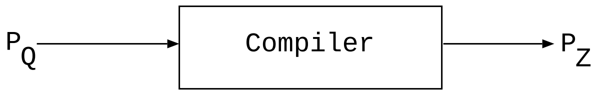


Abbildung 1.1: Blockdiagramm eines Compilers

Ein Compiler als Programm ist in einer Sprache formuliert. Diese Sprache nennen wir **Implementierungssprache** oder Systemsprache. Sie ist unabhängig von Quell- und Zielsprache. Mit einem **T-Diagramm** kann eine Übersetzung dargestellt werden. Abb. 1.2 beschreibt die Übersetzung eines Programms mit Bedeutung P von Q nach Z durch einen Compiler mit der Implementierungssprache S.

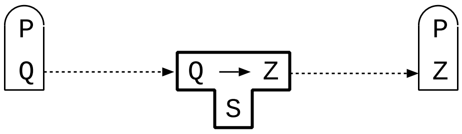


Abbildung 1.2: T-Diagramm eines Compilers

Als Implementierungssprache S eignet sich jede universelle (niedere oder höhere) Programmiersprache. Ein Compiler kann auf jedem Computer verwendet werden, der seine Implementierungssprache 'versteht'.
Beispiel 1.3: Gegeben sei ein Computer (eine Maschine) M, auf dem ein C-Compiler verfügbar ist. Gesucht sei ein Ada-Compiler für die Maschine M. Wir schreiben den

Ada-Compiler in C und übersetzen ihn wie ein gewöhnliches C-Programm mit dem verfügbaren C-Compiler. Danach ist der Ada-Compiler auf M verwendbar, um Ada-Programme zu übersetzen. Die T-Diagramme in Abb. 1.4 stellen den Zusammenhang bildlich dar.

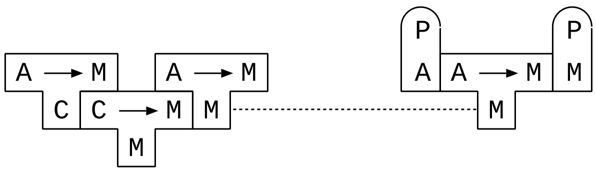


Abbildung 1.4: Übersetzung eines Compilers

Es wäre auch möglich, den Ada-Compiler von Bsp. 1.3 so zu schreiben, daß er nicht M, sondern C als Zielsprache hat. Dann müßte aber jedes Ada-Programm in zwei Stufen übersetzt werden (A→C→M).

Wenn ein Compiler in seiner Quellsprache Q geschrieben wird (Q = S), so kann er *sich selbst* übersetzen. Diese Möglichkeit läßt sich ausnutzen, um in ein oder mehreren Schritten modifizierte Versionen eines Compilers zu erstellen. Das Prinzip wird als **Bootstrapping** bezeichnet. Das folgende Beispiel zeigt eine typische Anwendung.

Beispiel 1.5: Gegeben sei ein Ada-Compiler auf einer Maschine M. Gesucht sei ein Ada-Compiler für eine andere Maschine N. Der Compiler A→N wird in Ada geschrieben und auf der Maschine M mit dem dort vorhandenen Compiler übersetzt. Er kann auf M bereits als Cross-Compiler verwendet werden, um Ada-Programme für die Maschine N zu übersetzen. Er kann aber auch sich selbst nach N übersetzen und dann auf der Maschine N verwendet werden (Abb. 1.6) .

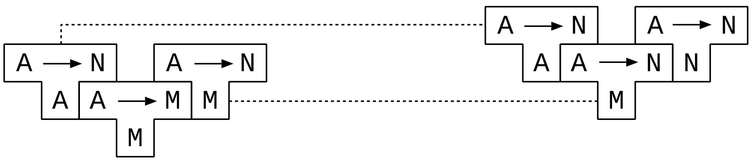


Abbildung 1.6: Selbst-Übersetzung eines Compilers

Der Compiler A→N mit der Implementierungssprache N wird erzeugt, ohne daß dazu die Maschine N selbst benötigt wird. N kann also eine beliebige, insbesondere auch eine abstrakte Maschine sein. Darauf wird im Kapitel 1.2 näher eingegangen. Die Erstellung von Compilern wird durch die Verwendung von höheren Programmiersprachen erleichtert. Eine noch weitergehende Unterstützung bieten **Compiler-Generatoren**. Compiler-Generatoren sind Werkzeuge zur Konstruktion von Compilern, die den Konstrukteur von Routinearbeit befreien.

Die Eingabe für einen Compilergenerator besteht aus einer formalen Definition der gewünschten Übersetzung X→Y. Der Generator erzeugt aus der Definition einen Compiler, indem er die Definition in ein ablauffähiges Programm übersetzt. Das Prinzip wird in Abb. 1.7 durch T-Diagramme dargestellt.

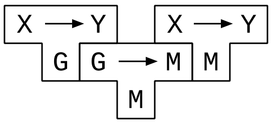


Abbildung 1.7: Compiler-Generator

Bei Verwendung eines Compiler-Generators wird der Compiler in einer **Generatorsprache G** geschrieben bzw. definiert. Das theoretische Modell für praktische Generatorsprachen ist die **Attributierte Grammatik**. Dabei wird die Syntax der zu übersetzenden Sprache X durch eine kontextfreie Grammatik definiert und die Übersetzung, d.h. die Zuordnung X→Y wird als Berechnung von Attributen der Grammatiksymbole spezifiziert (siehe Kapitel 5).

Besonders bekannt in der Praxis ist der Compiler-Generator **YACC** unter Unix. Die Generatorsprache ist hier eine Mischung bestehend aus einer speziellen Notation der Grammatik und aus der Programmiersprache C zur Berechnung der Attribute.

Die Logik-orientierte Programmiersprache **PROLOG** ist eine Generatorsprache, in der man sowohl die Syntax als auch die Attributberechnung, d.h. die Übersetzung einheitlich beschreiben kann.

1.2 Interpreter

Ein **Interpreter** ist ein Programm, das Programme ausführen kann, die in seiner **Interpretersprache** I formuliert sind. Ein Interpreter verhält sich wie eine Maschine bzw. ein Prozessor. Man bezeichnet ihn daher auch als **virtuelle** oder **abstrakte Maschine**.

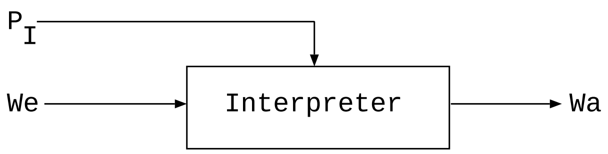


Abbildung 1.8: Blockdiagramm eines Interpreters

Der Interpreter (Abb. 1.8) führt das Programm P_I aus. Bei seiner Ausführung benötigt P_I im allgemeinen Eingangswerte We und liefert Ausgangswerte Wa.

Zum Vergleich von Compiler und Interpreter betrachten wir ein Quellprogramm, das eine lineare Folge von k Schritten beschreibt. Jeder Programmschritt i erfordert einen Decodierungsschritt D_i und einen Ausführungsschritt A_i.

- Bei der **Übersetzung** ergibt sich der Ablauf D_1, D_2, ..., D_k, Optimieren, Assemblieren, Linken, A_1, A_2, ..., A_k.

- Bei der **Interpretation** ergibt sich der Ablauf D_1, A_1, D_2, A_2, ..., D_k, A_k.

Bei einem linearen Programm (jeder Schritt wird genau einmal ausgeführt) ist der gesamte Zeitbedarf bei beiden Verfahren etwa gleich. Falls das Programm eine Schleife enthält (Schritt i wird n-mal ausgeführt), so ist der Zeitbedarf beim Interpreter um $(n-1) * D_i$ größer als beim Compiler. Falls das Programm eine Auswahl enthält (Schritt i besteht aus n Alternativen), so ist der Zeitbedarf beim Interpreter um $(n-1) * D_i$ kleiner als beim Compiler.

Interpretation ist zweckmäßig, wenn ein Programm oft geändert wird, z.B. beim Debugging.

Übersetzung ist generell zweckmäßig, wenn ein Programm mehrmals ausgeführt werden soll.

Übersetzung ist notwendig, wenn die statische Korrektheit eines Programms vor der Ausführung sichergestellt werden soll.

Interpretation ist notwendig, wenn das Programm selbst während der Laufzeit manipuliert werden soll.

1.3 Compiler-Interpreter

Oft werden Compiler und Interpreter kombiniert. Ein einfaches Beispiel ist ein Compiler, der Programme für einen Interpreter (abstrakte Maschine) erzeugt (siehe Abb. 1.9).

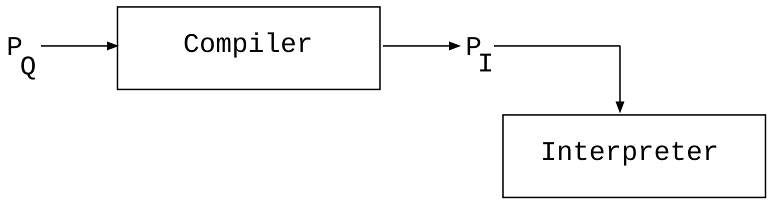


Abbildung 1.9: Blockdiagramm eines Compiler-Interpreter Systems

Diese Kombination kann man z.B. verwenden, um eine Quellsprache Q auf einer Maschine M zu implementieren, ohne daß man die Maschinenbefehle von M kennen und beachten muß. Als Zielsprache des Compilers wird eine Interpretersprache I entwickelt und ein Interpreter dafür geschrieben. Zweckmäßig wählt man für I eine Sprache, die einfach und effizient decodiert werden kann (z.B. Postfix-Code). Der

Compiler und der Interpreter können in einer beliebigen auf M verfügbaren Sprache X geschrieben werden. Abb. 1.10 zeigt ein solches System, wobei neben dem T-Diagramm auch ein I-Diagramm (für Interpreter) verwendet wird.

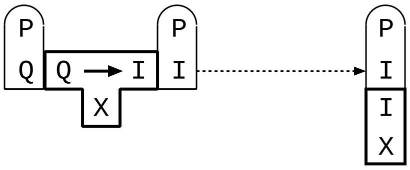


Abbildung 1.10: Implementierung von Q mittels Compiler-Interpreter

Der Nachteil dieser Implementierung ist, daß die erzeugten Programme durch den Interpreter langsamer als durch einen Hardware-Prozessor ausgeführt werden. Ein Vorteil liegt aber z.B. darin, daß das Zielprogramm in der Sprache I wesentlich kompakter dargestellt werden kann als in einem realen Maschinencode und daß die Übersetzung meist wesentlich schneller ist. Die Beispiele im Anhang sind Compiler-Interpreter Systeme nach dem Muster von Abb. 1.10 . Eine Compiler-Interpreter Kombination kann man insbesondere verwenden, um die **Portabilität** eines Compilers zu erreichen. Dazu muß der Compiler Q→I in seiner Zielsprache I dargestellt werden. Er muß deswegen nicht in I geschrieben werden, sondern kann – falls er in der Quellsprache Q geschrieben wird – nach dem Prinzip von Abb. 1.6 durch Selbst-Übersetzung nach I übersetzt werden. Danach ist der Compiler auf allen Maschinen einsetzbar, auf denen die abstrakte Maschine I als Interpreter implementiert ist. In Abb. 1.11 wird der Einsatz des Systems auf einer Maschine X dargestellt.

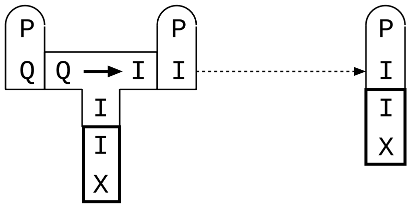


Abbildung 1.11: Portabler Compiler mittels Interpreter

Um das System auf einen neuen Computer X zu übertragen, muß also lediglich der Interpreter neu geschrieben werden. Der Nachteil dieses Verfahrens ist aber, daß der Compiler *und* die erzeugten Programme interpretiert werden (Bsp. UCSD-Pascal).

Das aktuelle Beipiel für Compiler-Interpreter Systeme ist Java. Der Zwischencode (Java VM Code) bringt nicht nur den Vorteil der Maschinenunabhängigkeit, sondern er verkürzt durch seine Kompaktheit auch die Übertragungszeiten im Internet.

Kapitel 2

Struktur von Compilern

Ein Compiler wandelt das Quellprogramm in mehreren **Phasen** (logischen Teilaufgaben) in das Zielprogramm um. Jede Phase bearbeitet eine abgeschlossene Teilaufgabe. Die Phasen sind durch den Datenfluß voneinander abhängig. In vielen Fällen können sie zeitlich verzahnt durchgeführt werden.

Phasen, die beim Ablauf nicht mit anderen Phasen verzahnt sind, bezeichnet man als einen „Durchlauf“ oder **Pass**. Es gibt logische und technische Gründe, den Ablauf eines Compilers in einen oder mehrere Pässe zu gliedern. Man beachte aber, daß die Phasen - die logischen Teilaufgaben – immer die gleichen sind.

2.1 Phasen

Jede Übersetzung hat zwei grundlegende Phasen: Analyse und Synthese.

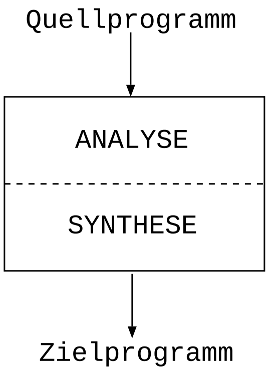


Abbildung 2.1: Grundstruktur eines Compilers

Die **Analyse** überprüft die **Syntax** des Quellprogramms und ermittelt seine **Struktur**. Außerdem wird die **statische Semantik** überprüft und ausgewertet. Die statische Semantik umfaßt semantische Eigenschaften des Programms, die zur Übersetzungszeit festliegen. Dies sind insbesondere die **Typen** von Objekten.

Die **Synthese** baut das Zielprogramm auf. Besondere Aspekte dabei sind einerseits **Optimierungen** zur Erhöhung der Effizienz des Zielprogramms und andererseits der Einbau **dynamischer Kontrollen** (z.B. Indexüberprüfung) zur Erhöhung der Sicherheit des Zielprogramms. Die Grundstruktur läßt sich weiter verfeinern:

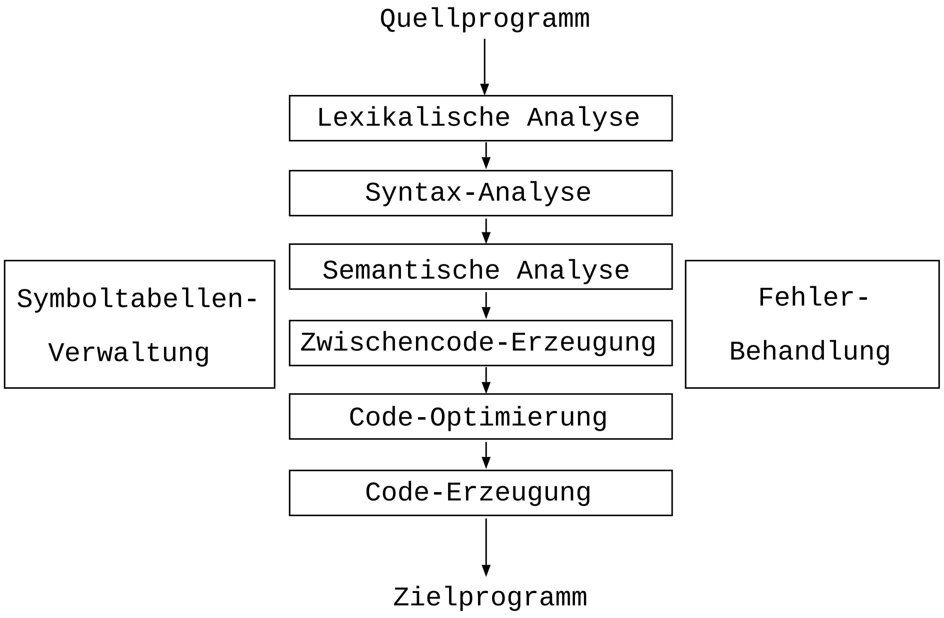


Abbildung 2.2: Verfeinerte Phasenstruktur

Gemäß Abb. 2.2 lassen sich Analyse und Synthese jeweils in drei Phasen gliedern. Dabei sind einzelne Phasen je nach Programmiersprache und Anspruch an die Qualität des erzeugten Codes mehr oder weniger stark ausgeprägt.

Die Pfeile geben die logische Abhängigkeit (Datenfluß) der Phasen an. Die seitlich dargestellten „Phasen“ stellen Teilaufgaben dar, die nicht in diesem Datenfluß liegen, sondern im wesentlichen mit allen Phasen Informationen austauschen.

Wir werden im folgenden anhand eines Programm-Beispiels einen Überblick über die Phasen einer Übersetzung geben. Als Quellsprache wird hier Modula-2 verwendet.

In Abb. 2.3 und Abb. 2.4 sind die Zwischenergebnisse der einzelnen Phasen dargestellt. Man beachte, daß es sich dabei um konzeptuelle Zwischendarstellungen handelt. Sie zeichnen die „Geschichte“ der erzeugten Informationen auf. Tatsächlich werden die Informationen oft nur in Teilen erzeugt. Es folgt nun eine kurze Beschreibung der Phasen.

Lexikalische Analyse

Die Lexikalische Analyse untersucht die Zeichenfolge des Quellprogramms und teilt sie in **Grundsymbole** ein. Diese werden durch Paare (Token, Attribut) dargestellt.

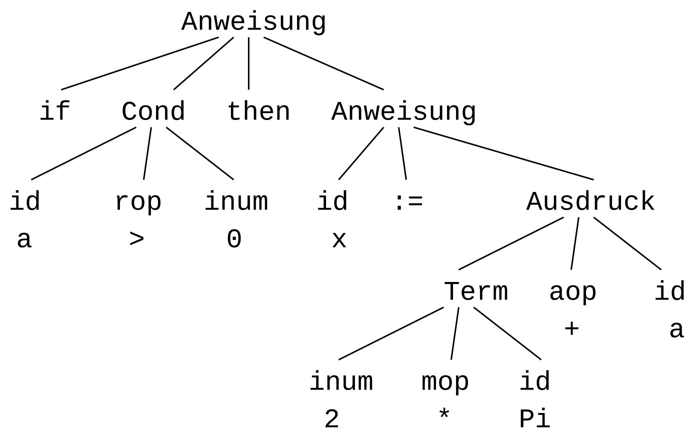
Ausschnitt eines Modula-2 Quellprogramms:

```
... CONST Pi=3.14;
    VAR a:INTEGER; x:REAL;
... IF a > 0 THEN
    x:=2*Pi+a ...
```

Folge von Symbolen:

```
... (const)(id,Pi)(=)(rnum,3.14)(;)
    (var)(id,a)(:)(int)(;)(id,x)(:)(real)(;)
... (if)(id,a)(rop,>)(inum,0)(then)
    (id,x)(:=)(inum,2)(mop,*)(id,Pi)(aop,+)(id,a) ...
```

Ableitungsbaum der IF-Anweisung:



Abstrakter Syntaxbaum:

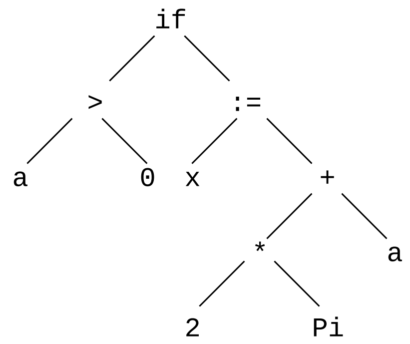
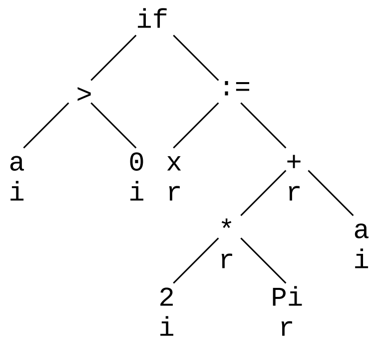


Abbildung 2.3: Vom Quellprogramm zum Syntaxbaum

Attributierter Syntaxbaum und Symboltabelle:



name	art	typ	wert/(adr)
Pi	c	r	3.14
a	v	i	(0)
x	v	r	(4)

Linearer Zwischencode:

    if a $>_i$ 0 goto L1 *$>_i$ vergleicht integer-Werte *)
    goto L2
L1: t1 $=_r$ intreal(2) $=_r$ bedeutet Zuweisung von real auf real *)
    t2 $=_r$ t1 $*_r$ 3.14 (* $*_r$ multipliziert real-Werte *)
    t3 $=_r$ intreal(a)
    t4 $=_r$ t2 $+_r$ t3
    x $=_r$ t4
L2:

Optimierter Zwischencode:

    if a $<=_i$ 0 goto L
    t1 $=_r$ intreal(a)
    x $=_r$ 6.28 $+_r$ t1
L:

Symbolischer Maschinencode:

```
   MOVi 0(A),R1
   CMPi #0,R1
   BLE  L
   FLT  R1,F1 (* Konversionsbefehl von integer nach real *)
   ADDr #6.28,F1
   MOVr F1,4(A)
L:
```

Abbildung 2.4: Vom Syntaxbaum zum Maschinencode

Das Token kennzeichnet das Symbol und das Attribut die individuelle Ausprägung. Falls das Token eindeutig ist, bleibt das Attribut leer.

Die Lexikalische Analyse untersucht nur einfache (also keine geschachtelten) Strukturen. Diese können durch **Reguläre Ausdrücke** beschrieben werden. Die Analyse arbeitet nach dem Prinzip eines **Endlichen Automaten**. Durch die Lexikalische Analyse wird die nachfolgende Syntax-Analyse wesentlich vereinfacht.

Syntax-Analyse

Die Syntax-Analyse betrachtet die von der Lexikalischen Analyse gelieferten Token (nicht die Attribute) und erzeugt einen **Ableitungsbaum** entsprechend der Grammatik der Quellsprache, die in der Regel als **kontextfreie Grammatik** gegeben ist. Die Syntax-Analyse arbeitet nach dem Prinzip eines **Kellerautomaten**.

In den Endknoten des Ableitungsbaums stehen die Token als Terminalsymbole der Grammatik. Die Nonterminale der Grammatik dienen nur zum Aufbau der Struktur und können danach eliminiert werden. Man markiert dabei jeden Zwischenknoten durch den Operator, der den Teilbaum charakterisiert. Es entsteht ein Operatorbaum bzw. ein (**abstrakter**) **Syntaxbaum**.

Ein abstrakter Syntaxbaum kann auf verschiedene Weise implementiert werden: In imperativen Sprachen als dynamische Datenstruktur, in objektorientierten Sprachen als Instanz einer Klassenhierarchie, in funktionalen Sprachen als geschachtelte Liste und in logischen Sprachen als Term. Oft wird der Baum nicht explizit dargestellt, sondern nur implizit durch den Aufruf entsprechender Codestücke, die die semantische Analyse und die Zwischencodeerzeugung durchführen.

Semantische Analyse

Während die Syntax-Analyse nur strukturelle Beziehungen zwischen Symbolen untereinander analysiert, behandelt die Semantische Analyse die Bedeutung von Symbolen, d.h. die Beziehungen zwischen Symbolen und den von ihnen bezeichneten Objekten (z.B. Konstanten, Variablen).

Die Semantische Analyse trägt Art und Typ der von Symbolen bezeichneten Objekte in die **Symboltabelle** ein. Die zentrale Aufgabe der Semantischen Analyse ist die **Typ-Überprüfung** (*type checking*). Dabei ist zu prüfen, ob Verwendungen von Objekten mit ihren Deklarationen typkonsistent sind. Bei überladenen Operatoren (z.B. `"+"` für integer- und real-Addition) muß dabei aus dem Typ der Operanden der Typ des Operators abgeleitet werden. Man nennt dies **Operator-Identifizierung**. Im einfachsten Fall kann die Semantische Analyse den Syntaxbaum von den Blättern zur Wurzel hin untersuchen. Das Ergebnis der Analyse ist ein **attributierter Syntaxbaum**, der das syntaktisch und statisch-semantisch analysierte Programm repräsentiert.

Die Semantische Analyse behandelt **Kontextbeziehungen**, die nicht durch eine kontextfreie Grammatik definiert werden können - formal ist auch sie ein Teil der Syntax-Analyse, nämlich der nicht-kontextfreien. Als Basis für die Semantische Analyse eignet sich die **Attributierte Grammatik**.

Zwischencode-Erzeugung

Der attributierte Syntaxbaum kann bereits als eine Darstellung des Programms in einem Zwischencode angesehen werden, der von einer abstrakten Maschine ausgeführt werden könnte. Im Hinblick auf reale Maschinen muß der Baum aber linearisiert werden. Dabei sind insbesondere Sprungbefehle zu erzeugen.

In Abb. 2.4 ist für den linearen Zwischencode eine abstrakte Drei-Adreß-Maschine gewählt. Zur Linearisierung wird der Syntaxbaum in einem Tiefendurchlauf von links nach rechts abgearbeitet. Für den bedingten Ausdruck werden Sprungbefehle und Marken erzeugt. Bei der Zuweisungs-Anweisung wird für jeden Operatorknoten ein Befehl generiert. Zur Typ-Konversion werden zusätzliche Befehle eingefügt (z.B. `intreal`). Für jedes Zwischenergebnis wird eine temporäre Hilfsvariable $\mathtt{t}_i$ verwendet.

Alle bisherigen Phasen sind prinzipiell noch unabhängig von einer konkreten Zielmaschine. Diese ersten Phasen bezeichnet man auch als **front-end** des Compilers und die anschließenden Phasen als **back-end**.

Code-Optimierung

Die Code-Optimierung versucht, das Programm zu vereinfachen, um Speicherplatz und/oder Laufzeit einzusparen. Grundsätzlich kann man unterscheiden zwischen maschinenunabhängigen und maschinenabhängigen Optimierungen. Typische maschinenunabhängige Optimierungen sind z.B. die Vereinfachung von Indexberechnungen in Schleifen und die Auswertung konstanter Ausdrücke. Bei der maschinenabhängigen Optimierung dagegen werden vor allem spezielle Befehle der Zielmaschine ausgenutzt.

In Abb. 2.4 werden einfache maschinenunabhängige Optimierungen gezeigt: Der Sprungbefehl `goto L2` kann durch Umkehrung der Vergleichsoperation eliminiert werden. Die Konversionsoperation `intreal(2)` muß nicht erzeugt werden, sondern kann zur Übersetzungszeit ausgeführt werden. Dann läßt sich auch die Multiplikation `2.0*3.14` berechnen. Insgesamt ergibt sich eine Optimierung von 7 auf 3 Befehle.

Code-Erzeugung

Die Code-Erzeugung generiert ein Programm, das auf einer realen Maschine ablaufen kann. Dieses Programm wird auch **Objektprogramm** genannt. Die in Abb. 2.4 gewählte Maschine entspricht einem 68000-Prozessor, der über Register `Ri` verfügt und um einen Koprozessor für real-Operationen mit Registern `Fi` erweitert ist. Wir nehmen an, daß ein integer-Wert 4 Bytes benötigt. Wenn `a` im Variablenspeicher die relative Adresse `d=0` hat, bekommt `x` die relative Adresse `d=4`. Diese relativen Adressen werden in die Symboltabelle eingetragen. Das Adreßregister `A` enthält zur Laufzeit des Objektprogramms die Anfangsadresse des Variablenspeichers. Dann können die Variablen durch indizierte Adressierung `d(A)` angesprochen werden.

In Abb. 2.4 ist der erzeugte Code als Assemblercode angegeben, um den Code lesbar zu machen. Dieser Code muß durch den **Assembler** in ausführbaren Maschinencode übersetzt werden.

Binder und Lader

Programme werden im allgemeinen als verschiebbarer (*relocatable*) Code erzeugt. D.h., daß sie nicht in einen bestimmten Speicherbereich geladen werden müssen. In vielen Programmiersprachen gibt es außerdem die Möglichkeit, ein Programm auf verschiedene Module (Dateien) aufzuteilen und getrennt zu übersetzen. Das Überprüfen der Schnittstellen ist Aufgabe des Compilers, während das Zusammenfügen von Programmteilen (Auflösen der Referenzen) und das Laden in den Speicher vom Betriebssystem, d.h. vom **Binder** (*linker*) und **Lader** (*loader*) durchgeführt werden. Dabei können auch Bibliotheksprozeduren (*libraries*), die schon in Objektcode vorliegen, eingebunden werden. Grundsätzlich kann Binden und Laden von Programmteilen vor oder während der Laufzeit des Objektprogramms erfolgen.

Laufzeitsystem

Die Zielmaschine eines Compilers ist im allgemeinen eine um Software erweiterte Hardware gemäß Abb. 2.5. Das Objektprogramm verwendet Funktionen der Hardware, Funktionen des Betriebssystems (z.B. E/A Routinen) und Funktionen des **Laufzeitsystems**. Das Laufzeitsystem umfaßt Unterprogramme, die zum Ablauf von Objektprogrammen generell benötigt werden (z.B. Speicherverwaltung). Es bildet eine Compiler-spezifische Softwareschicht über dem Betriebssystem und muß daher auf das verwendete Betriebssystem genau abgestimmt werden.

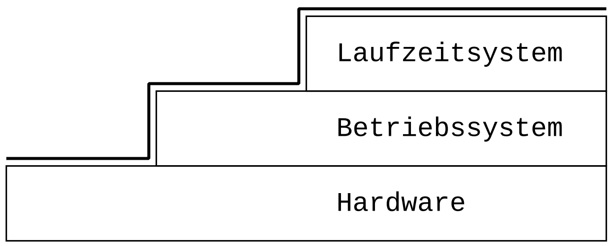


Abbildung 2.5: Schichtenmodell der Zielmaschine

Ein Interpreter kann als ein Laufzeitsystem angesehen werden, das Hardware und Betriebssystem voll überdeckt und einen eigenen Maschinencode interpretiert.

2.2 Ablaufsteuerung

Alle Phasen (Teilaufgaben) eines Compilers können unter bestimmten Umständen zeitlich verzahnt durchgeführt werden, d.h. das Quellprogramm kann in einem Pass (Durchlauf) übersetzt werden. Einen solchen Compiler nennt man daher einen Ein-Pass-Compiler. Wenn das Quellprogramm oder Zwischendarstellungen davon mehrmals von einem Compiler gelesen werden müssen, spricht man von einem Mehr-Pass-Compiler. Wir betrachten im folgenden einige typische Strukturen.

Ein-Pass-Compiler

Ein-Pass-Compiler sind besonders schnell, da das Programm nur einmal gelesen wird und keine Zwischendarstellungen des Programms erzeugt und weitergegeben werden müssen. Ein-Pass-Übersetzung ist aber nur möglich, wenn es die Programmiersprache zuläßt. Insbesondere müssen alle Deklarationen textlich *vor* ihrer Verwendung stehen, damit der Compiler zu jedem Zeitpunkt über vollständige Informationen verfügt. Optimierungen sind nur sehr beschränkt möglich, da keine globalen Informationen gesammelt und ausgewertet werden können. Abb. 2.6 zeigt die typische Ablaufstruktur.

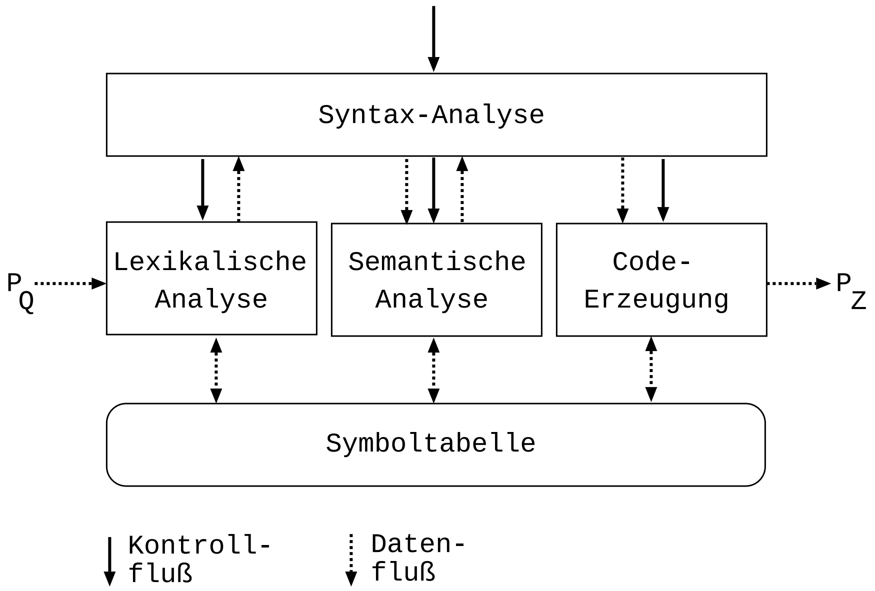


Abbildung 2.6: Ablaufstruktur eines Ein-Pass-Compilers

Der Ein-Pass-Compiler wird von der Syntax-Analyse gesteuert. Die anderen Teilaufgaben (Lexikalische Analyse, Semantische Analyse und Code-Erzeugung) werden als Prozeduren aufgerufen.

Mehr-Pass-Compiler

Falls die Quellsprache erlaubt, daß Deklarationen auch textlich *nach* ihrer Verwendung stehen können, muß der Compiler mit mindestens zwei Pässen arbeiten. Der erste Pass analysiert zunächst alle Deklarationen und der zweite Pass baut das Zielprogramm auf. Abb. 2.7 zeigt die typische Struktur.

Der erste Pass hat im Modell von Abb. 2.7 nur Analyseaufgaben. Er sammelt alle Deklarationen in der Symboltabelle und wandelt das Programm in eine geeignete Zwischendarstellung (Zwischencode) um. Nach Ablauf des ersten Passes wird der zweite Pass aufgerufen. Er verwendet die erzeugten Informationen und baut das Zielprogramm auf. Zwei-Pass-Übersetzung wird häufig für Assembler verwendet,

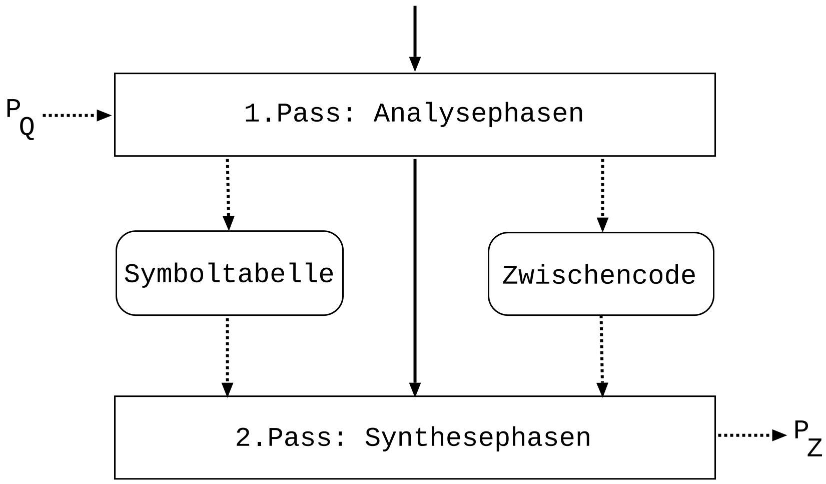


Abbildung 2.7: Ablaufstruktur eines Zwei-Pass-Compilers

wobei der erste Pass die Adressen der symbolischen Marken bestimmt und der zweite Pass den Maschinencode erzeugt.

Im Prinzip kann jede Phase eines Compilers als ein Pass implementiert werden. Der Steuerfluß der Pässe entspricht dann genau dem Datenfluß von Abb. 2.2. Der Compiler startet mit der Lexikalischen Analyse. Jeder Pass ruft seinen logischen Nachfolger auf. Zwischen den Pässen wird jeweils die Zwischendarstellung des Programms als Ganzes übergeben.

Ein Mehr-Pass-Compiler hat den Vorteil, daß die einzelnen Pässe klare, explizite Schnittstellen haben. Außerdem benötigt der Compiler wenig Hauptspeicher, da die Pässe nacheinander geladen werden können. Der Nachteil liegt darin, daß das Erzeugen und Verarbeiten von Zwischendarstellungen zeitaufwendig ist. Heutige Compiler können nicht so einfach eingeordnet werden. Oft zeigen sie im Back-End zwar im Prinzip eine Mehr-Pass-Struktur, nicht aber auf der Ebene von Eingabedateien. Die Zwischendarstellungen werden im Speicher übergeben.

Kapitel 3

Lexikalische Analyse

Die Lexikalische Analyse (*scanning*) liest das Quellprogramm als Zeichenfolge (z.B. ASCII) und teilt es in Symbole ein. Für den Compiler bedeutungslose Zeichen (z.B. Leerzeichen, Kommentare) werden dabei entfernt.

3.1 Grundsymbole

Ein oder mehrere aufeinanderfolgende Zeichen im Quellprogramm bilden ein **Grundsymbol** (auch kurz: Symbol). Nach ihrer Schreibweise im Quellprogramm kann man folgende Arten von Grundsymbolen unterscheiden:

- Wortsymbole (*keywords*), z.B. `if`, `while`
- Bezeichner (*identifier*) als Namen für definierte Objekte, z.B. `x1`
- Literale (*literals*) als Notationen für Werte, z.B. `3.14`
- Spezialsymbole (*delimiter*), z.B. `=`, `+`, `==`

Die Codierung der Symbole in der Ausgabe der Lexikalischen Analyse wird durch die weitere Verarbeitung im Compiler bestimmt. Die nachfolgende Syntax-Analyse basiert auf einer kontextfreien Grammatik, welche die Grundsymbole als *Terminale* verwendet. Entsprechend klassifiziert man die Symbole. Jedes Wortsymbol wird auf ein eigenes Terminal, alle Bezeichner werden auf ein einziges Terminal abgebildet. Literale werden nach dem Typ des Wertes (z.B. integer, real, string) auf Terminale abgebildet. Spezialsymbole entsprechen grundsätzlich jeweils einem Terminal. Man kann aber Operatoren auch zusammenfassen und entsprechend ihrer syntaktischen Rolle (z.B. relational, arithmetisch) auf Terminale abbilden.

Für die Syntax-Analyse ist allein die Abbildung der Symbole auf Terminale relevant, aber für weitere Phasen des Compilers ist auch die individuelle Ausprägung, z.B. die Identität der Bezeichner und der Wert von Literalen von Bedeutung. Zur Codierung eines Symbols benötigt man daher ein Tupel (Token, Attribut).

Token ist der Terminal-Code für die Syntax-Analyse, während das **Attribut** die Ausprägung des Symbols beschreibt. Als Attribut kann man im Prinzip das **Lexem**

(d.h. die Zeichenfolge des Symbols) aus dem Quellprogramm übernehmen. Abb. 2.3 in Kapitel 2 zeigt eine entsprechende Codierung von Symbolen. Meistens wird für die Fehlerbehandlung in späteren Phasen auch die Stelle, an der ein Symbol bzw. Lexem im Quellprogramm steht (Zeile, Spalte), als weitere Komponente codiert.

Spezifikation von Symbolen

Der Aufbau von Symbolen aus Zeichen kann durch **Reguläre Ausdrücke** oder **Reguläre Definitionen** exakt spezifiziert werden. Man bezeichnet in diesem Zusammenhang die Symbole auch als lexikalische Konstrukte.

```
[0-9]+ | [0-9]* ([0-9]"." | "."[0-9]) [0-9]*
```

Beispiel 3.1: Regulärer Ausdruck zur Beschreibung von Zahlen. Wir betrachten integer- und real-Zahlen ohne Exponent, wobei vor oder nach dem Dezimalpunkt mindestens eine Ziffer stehen soll. (Die meisten Programmiersprachen verlangen, daß mindestens eine Ziffer vor dem Punkt steht.)

Ein Regulärer Ausdruck (Bsp. 3.1) beschreibt eine Menge von Zeichenfolgen. Er wird aus Zeichen und Operatoren (Metasymbolen) aufgebaut. Die Zeichen werden zur Unterscheidung von den Metasymbolen oder zur Verdeutlichung bei Bedarf zwischen Anführungsstriche (`"t"`) gestellt.

Als Operatoren (Metasymbole) stehen zur Verfügung:

- Bereich von Eingabezeichen ($[t_1\text{-}t_n]$ für Zeichen zwischen t_1 und t_n),
- Iteration (`r*` beliebige Vorkommnisse, `r+` mindestens ein Vorkommnis von `r`),
- Option (`r?`),
- Verkettung ($r_1\ r_2$) und
- Alternative ($r_1 \mid r_2$)

Die Reihenfolge entspricht den Operator-Prioritäten (Alternative bindet am schwächsten), auch Klammerung ist möglich.

`[0-9]` spezifiziert z.B. eine Ziffer, `[0-9]+` eine oder mehrere Ziffern.

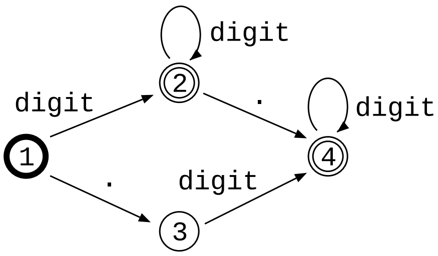
Reguläre Definitionen dienen zur Bennenung und übersichtlichen Schreibweise von Regulären Ausdrücken (siehe Bsp. 3.2). Auf der rechten Seite der Definitionen stehen Reguläre Ausdrücke, die *Namen* von Regulären Ausdrücken enthalten können. Rekursionen sind dabei nicht erlaubt, d.h. rechts dürfen nur *vorher* definierte Namen stehen.

```
digit       = [0-9]
digits      = digit+
pointgroup  = digit "."| "."digit
integer     = digits
real        = digits? pointgroup digits?
number      = integer | real
```

Beispiel 3.2: Die reguläre Definition der Zahlen von Bsp. 3.1.

Erkennen von Symbolen

Zum Erkennen von Symbolen werden **Endliche Automaten** verwendet: Sie untersuchen Zeichen für Zeichen, ob der Anfang der vorliegenden Zeichenfolge einem Symbol entspricht. In Abhängigkeit von ihrem Zustand und dem zuletzt gelesenen Eingabezeichen gehen sie in einen neuen Zustand über, bis sie einen Endzustand erreichen und damit ein Symbol erkannt haben. Endliche Automaten können als **Zustandsdiagramm** (Bsp. 3.3) oder als **Zustandsmatrix** (Bsp. 3.4) dargestellt werden.



Beispiel 3.3: Zustandsdiagramm für die Zahlen-Analyse (Bsp. 3.1)

Die Zustände entsprechen den Knoten des Graphen (Anfangszustand: fetter Rand, Endzustände: doppelter Rand). An den Kanten stehen die Eingabezeichen, bzw. Zeichenmengen, mit denen ein Übergang zwischen den Zuständen stattfindet.

Jede Zeile entspricht einem Zustand, jede Spalte einem Eingabezeichen. In der Matrix stehen die Folgezustände. Leereinträge (.) zeigen lexikalische Fehler an.

Der bisher betrachtete Automat ist ein deterministischer Endlicher Automat (DFA, *deterministic finite automaton*). Bei einem DFA gibt es für jede Kombination aus Zustand und Eingabezeichen nur maximal einen Folgezustand. Für einfache Reguläre Ausdrücke kann man den zugehörigen DFA intuitiv konstruieren, aber für komplexere Ausdrücke benötigt man ein algorithmisches Verfahren.

Im folgenden wird ein Verfahren vorgestellt, das Reguläre Definitionen in minimale DFA umgeformt. Die meisten Scanner-Generatoren (z.B. LEX unter Unix) arbeiten nach diesem Verfahren. Zunächst wird ein nichtdeterministischer Endlicher

	Eingabezeichen	
Zustand	`digit`	`"."`
1	2	3
∘ 2	2	4
3	4	.
∘ 4	4	.

Beispiel 3.4: Zustandsmatrix für die Zahlen-Analyse (vgl. Bsp. 3.3). Anfangszustand ist Zustand 1, Endzustände sind durch ∘ markiert. Die Zustandsmatrix ist hier optimiert: Durch Bilden der Zeichenmenge `digit` wurden 9 Spalten eingespart.

Automat (NFA) erzeugt, der dann in einen DFA umgewandelt wird. Wir beschreiben im folgenden das Verfahren in vier Schritten ausgehend von Bsp. 3.2.

(1) Umformung der Regulären Definition in Reguläre Ausdrücke

Durch wiederholte Textersetzung wird die Definition jedes Symbols in einen Regulären Ausdruck umgewandelt. Vereinfachungen sind dabei möglich, z.B. Ersetzen von `r+?` durch `r*` oder von `r* r` durch `r+`. Angewandt auf Bsp. 3.2 ergeben sich für `integer` und `real` die folgenden Regulären Ausdrücke, wobei zur besseren Lesbarkeit `digit` nicht ersetzt wurde.

```
digit+ |                                  (* integer *)
digit* (digit "."| "."digit) digit*       (* real *)
```

(2) Umformung der Regulären Ausdrücke in einen NFA

Bei einem NFA kann es für einen Zustand und ein Eingabezeichen mehrere Folgezustände geben. Er kann auch ε -Kanten enthalten. Der NFA erkennt ein Eingabewort (Symbol), wenn es einen Weg vom Start- zu einem Endzustand gibt, an dessen Kanten die Zeichen des Eingabeworts in entsprechender Folge stehen.

Für jeden Regulären Ausdruck, der einem Symbol entspricht, wird aus den Elementen von Abb. 3.5 ein NFA konstruiert (Bsp. 3.6). Die Anfangszustände der gewonnenen NFAs werden durch ε -Kanten mit einem gemeinsamen Startzustand verbunden (Bsp. 3.7). Es entsteht ein NFA mit einem Startzustand und mehreren Endzuständen, von denen jeder dem Erkennen eines bestimmten Symbols entspricht.

(3) Umformung des NFA in einen DFA

Der NFA läßt sich direkt zur Analyse einsetzen, indem bei mehreren möglichen Nachfolgezuständen alle Alternativen durchprobiert werden. Bei Anwendungen, die mit wechselnden Regulären Ausdrücken jeweils relativ wenig Text verarbeiten (z.B. Editoren), wird dies auch praktiziert. Für die Lexikalische Analyse im Compiler ist es effizienter, einen DFA zu verwenden. Jeder NFA kann in einen äquivalenten DFA (d.h. einen DFA, der die gleiche Sprache akzeptiert) umgewandelt werden. Diese

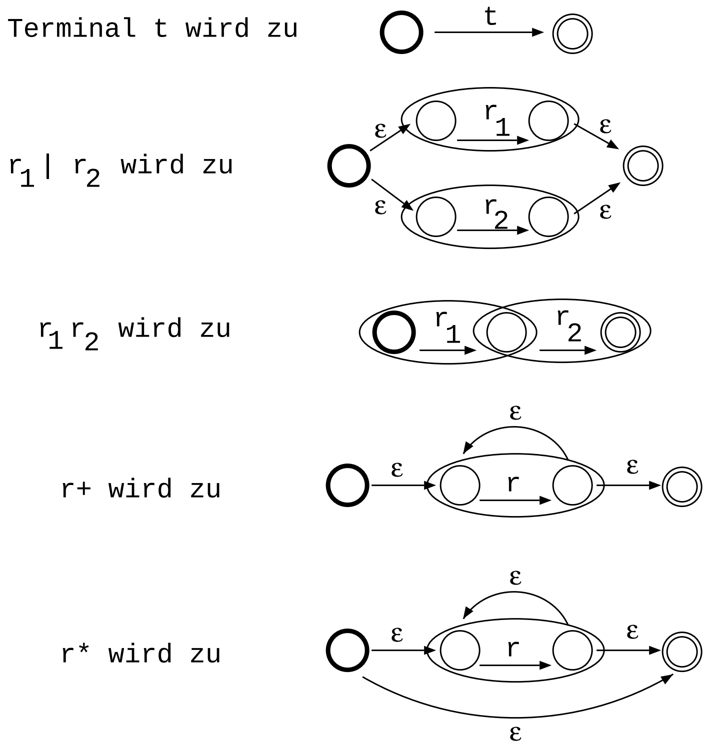
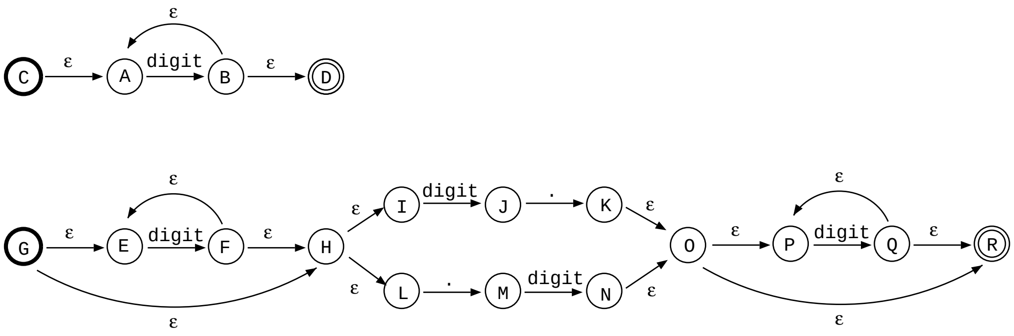


Abbildung 3.5: Konstruktion eines NFA aus einem regulären Ausdruck



Beispiel 3.6: Umformung der Regulären Ausdrücke für integer und real

Zustand	digit	"."	ε
S	.	.	CG
A	B	.	.
B	.	.	AD
C	.	.	A
○ D	.	.	.
E	F	.	.
F	.	.	EH
G	.	.	EH
H	.	.	IL
I	J	.	.
J	.	K	.
K	.	.	O
L	.	M	.
M	N	.	.
N	.	.	O
O	.	.	PR
P	Q	.	.
Q	.	.	PR
○ R	.	.	.

Man erkennt an den Doppeleinträgen und ε-Übergängen, daß der Automat nichtdeterministisch ist.

Beispiel 3.7: Die Zustandsmatrix des erzeugten NFA

Umwandlung ist ein häufig angewandtes Verfahren, das uns in diesem Skriptum noch mehrfach begegnen wird (z.B. in Kap. 4). Das Grundprinzip dabei ist, mehrere mögliche Nachfolgezustände zu einem einzigen neuen Zustand zusammenzufassen und Entscheidungen aufzuschieben, bis mehr von der Eingabe bekannt ist. Der Alg. 3.8 erzeugt nach diesem Prinzip aus einem NFA die Zustandsmatrix `dTransTab` des äquivalenten DFA.

Jedes Element in `dStates`, das einen NFA-Endzustand enthält, wird Endzustand des DFA (Bsp. 3.9). Der DFA kann durch Zusammenfassen von Zuständen verkleinert werden - eine derartige Minimierung ist aus Effizienzgründen immer sinnvoll.

(4) Minimierung eines DFA

Die Anzahl der Zustände wird minimiert, indem äquivalente Zustände zusammengefaßt werden (zwei Zustände sind äquivalent, wenn sie durch die gleiche Menge von Eingabeworten in einen Endzustand führen). Praktisch geht man so vor, daß man die Menge der Zustände iterativ in nicht äquivalente Teilmengen zerlegt (Alg. 3.10 und Bsp. 3.11).

`public NStates move(NStates nStates, char input)` liefert die Menge aller NFA-Zustände, zu denen Übergänge von einem Zustand aus der Menge `nStates` mittels des Eingabezeichens `input` führen;

`public NStates epsclosure(NStates nStates)` liefert die Menge aller NFA-Zustände, die über eine Folge von ε-Übergängen von einem Zustand in `nStates` erreicht werden können (d.h. die ε-Hülle). Darin sind alle Zustände aus `nStates` selbst enthalten

```
dStates = epsclosure( InitialState )                    // Startzustand
do {
  Für jeden DFA-Zustand dState in dStates
  und jedes Eingabezeichen input
  newState = epsclosure(move(dState, input))
    if( !empty(newState) && !dStates.member(newState) )
      // neuer Zustand noch nicht enthalten
      füge newState zu dStates hinzu
    dTransTab[dState][input] = newState
    } while(dStates verändert sich noch immer)
```

Algorithmus 3.8: Erzeugung der Zustandsmatrix eines äquivalenten DFA aus einem NFA.

3.2 Ausgabe-Aktionen

Das vorige Kapitel behandelte die Spezifikation bzw. Analyse von Symbolen (lexikalischen Konstrukten). Die eingelesene Zeichenfolge soll aber nicht nur geprüft, sondern auch umcodiert werden. Dazu erweitert man die Analyse um geeignete (Ausgabe-)Aktionen.

Aktionen werden prinzipiell erst nach dem Erkennen eines Symbols ausgeführt. Das kann zu Redundanz führen, wenn das Lexem nochmals durchsucht werden muß (z.B. bei der Behandlung von doppelten Hochkommas in Strings). Im allgemeinen aber kann der Scanner erst am Schluß entscheiden, welche Aktion(en) auszuführen sind.

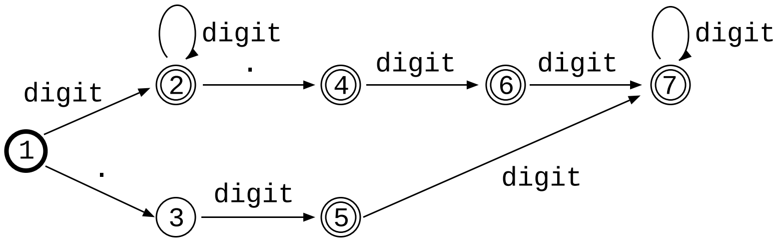
Bei Regulären Definitionen wird jedem Symbol eine Aktionsfolge oder eine Prozedur zugeordnet (Bsp. 3.12).

Bei Ausführung der Aktion muß die als Symbol erkannte Zeichenfolge (Lexem) des Quellprogramms verfügbar sein. Dazu kann man die Zeichen während der Analyse in einem Array sammeln, oder man verwendet zwei Zeiger in den Eingabepuffer: Der erste wird auf den Anfang des Lexems gesetzt und der zweite wird bei der Analyse fortgeschaltet, so daß er nach Erkennen des Symbols das Ende des Lexems anzeigt. Folgende Aktionen treten bei der Analyse üblicher Programmiersprachen auf:

Man beginnt mit der ε-Hülle (siehe Prozedur `epsclosure`) von `S`: Sie enthält `S` selbst; von `S` führen ε-Kanten nach `C` und `G`, von da nach `A` bzw. nach `E` und `H` und von dort nach `I` und `L`. Von keinem dieser Zustände führen ε-Kanten weg. Der Startzustand des DFA ist also `SACEGHIL` (kurz für {`S,A,C,E,G,H,I,L`}). `move(SACEGHIL,digit)`= `BFJ` (wegen der Übergänge von `A`, `E` und `I`). Die Hülle davon (`ABDEFHIJL`) bildet den zweiten DFA-Zustand. Die erste Zeile der Zustandsmatrix wird mit der Hülle von `move(SACEGHIL,".")`= `M` fertiggestellt. Mit den neu gewonnenen DFA-Zuständen wird die Matrix Zeile für Zeile erstellt:

		DFA-Zustand	digit	"."
(1)		SACEGHIL	ABDEFHIJL	M
(2)	○	ABDEFHIJL	ABDEFHIJL	KMOPR
(3)		M	NOPR	.
(4)	○	KMOPR	NOPQR	.
(5)	○	NOPR	QPR	.
(6)	○	NOPQR	QPR	.
(7)	○	QPR	QPR	.

Zustandsdiagramm des generierten DFA:



Beispiel 3.9: Determinisierung des NFA

Zurückliefern des Token. Token werden als integer-Werte in Form benannter Konstanten oder durch einen Aufzähltyp repräsentiert.

Eintragungen in Tabellen. Bezeichner und Konstanten werden aus Effizienzgründen oft in Tabellen gehalten. Bei ihnen muß also überprüft werden, ob das Lexem bereits in der entsprechenden Tabelle enthalten ist. Sollte das nicht der Fall sein, wird es eingetragen. In jedem Fall wird ein Verweis auf den Eintrag (Index) als Attribut zurückgeliefert. In Bsp. 3.12 erledigen das die Prozeduren `installInum` bzw. `installRnum`.

Behandlung von Wortsymbolen. Nach dem Erkennen eines Namens muß überprüft werden, ob dieser Name ein Wortsymbol ist. Das entsprechende Token soll ausgegeben werden (z.B. `if`, `then`, ... im Gegensatz zu `id`). Dazu wird entweder eine eigene Tabelle für Wortsymbole geführt oder die Bezeichner-Tabelle wird am Anfang

---

Anfangsaufteilung herstellen: teile die Zustandsmenge `dStates` des DFA in 2 Teilmengen: alle Endzustände, alle anderen Zustände

```
rest = dStates - finalStates
partition = SetOf( finalStates,rest )
```

endgültige Aufteilung iterativ generieren

```
do {
   teile jede Menge in partition so in Untermengen, daß zwei
   Zustände genau dann in dieselbe Untermenge kommen, wenn die
   Übergänge mittels der gleichen Eingabezeichen jeweils in dieselben
   ursprünglichen Mengen führen
   partition = die neuen Untermengen
   } while(partition hat sich geändert)
```

Zustandsmatrix des minimalen Automaten erzeugen: Die neuen Zustände sind die Mengen der endgültigen Aufteilung. Das sind maximal soviele wie beim ursprünglichen DFA, wenn jede Menge einelementig ist.

```
states = partition
```

Übergänge (`transTab`) zwischen den neuen Zuständen (Mengen) genau dann generieren, wenn es Übergänge zwischen ihren Elementen gibt.

Algorithmus 3.10: Minimierung eines DFA durch Zerlegung der Menge der Zustände in nicht äquivalente Teilmengen.

mit den Wortsymbolen initialisiert. Indizes kleiner als ein bestimmter Grenzwert sind dann für Wortsymbole reserviert, größere zeigen auf einen Bezeichner.

Ausblenden von Zeichen. In den meisten Programmen gibt es viele Zeichen, die nur die Lesbarkeit erhöhen (Zwischenraum, Tabulator, Zeilenwechsel, ...), für den Compiler also unwichtig sind. Manche benötigt der Scanner, um das Ende eines Lexems zu erkennen (Trennzeichen). Alle derartigen Leerzeichen können von der Lexikalischen Analyse ausgeblendet werden, d.h. ihre Ausgabeaktion ist leer. Das Mitzählen von Zeilenwechseln (und Spalten) ist jedoch für genaue Fehlermeldungen nötig. Ähnlich wie die Leerzeichen sind Kommentare für den Compiler bedeutungslos und können ausgeblendet werden. Geschachtelte Kommentare (wie z.B. in Modula erlaubt) lassen sich mit normalen regulären Methoden nicht behandeln.

3.3 Analyse-Verfahren

Die Analyse soll den jeweils längsten passenden Anfang der Eingabefolge als Symbol erkennen (*longest input match*). Beim Automaten in Bsp. 3.3 führen von den Endzuständen noch Kanten weg. Daher sind auch in einem Endzustand weitere Zeichen zu lesen, solange es Übergänge für sie gibt.

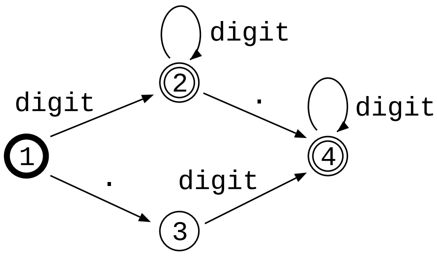
Anfangsaufteilung: `24567` (Endzustände), `13` (restliche Zustände)

1. Iterationsschritt:

1. Menge:	2	(digit→ $\underline{2}$4567,"."→2$\underline{4}$567)
	4	(digit→245$\underline{6}$7,"."→{})
	5	(digit→2456$\underline{7}$,"."→{})
	6	(digit→2456$\underline{7}$,"."→{})
	7	(digit→2456$\underline{7}$,"."→{})
2. Menge:	1	(digit→ $\underline{2}$4567,"."→1$\underline{3}$)
	3	(digit→24$\underline{5}$67,"."→{})
1. Aufteilung:	2,	4567, 1, 3

2. Iterationschritt: `2`, `1` und `3` sind einelementig – es ist keine weitere Aufteilung möglich. Bei der Menge `4567` führen bei jedem Element Übergänge mit `digit` nach `4567` selbst, Übergänge für `"."` existieren nicht. Es ist keine weitere Aufteilung nötig. In der Folge wird kurz `4` statt `4567` geschrieben.

2. Aufteilung = 1. Aufteilung = endgültige Aufteilung
Der minimale DFA hat das folgende Aussehen (vgl. Bsp. 3.3):



Beispiel 3.11: Minimierung des DFA: Zustandsmengen werden durch Ziffernfolgen dargestellt. Bei den Iterationsschritten stehen die möglichen Übergänge in Klammern, wobei der ursprüngliche Folgezustand innerhalb der Mengen unterstrichen ist.

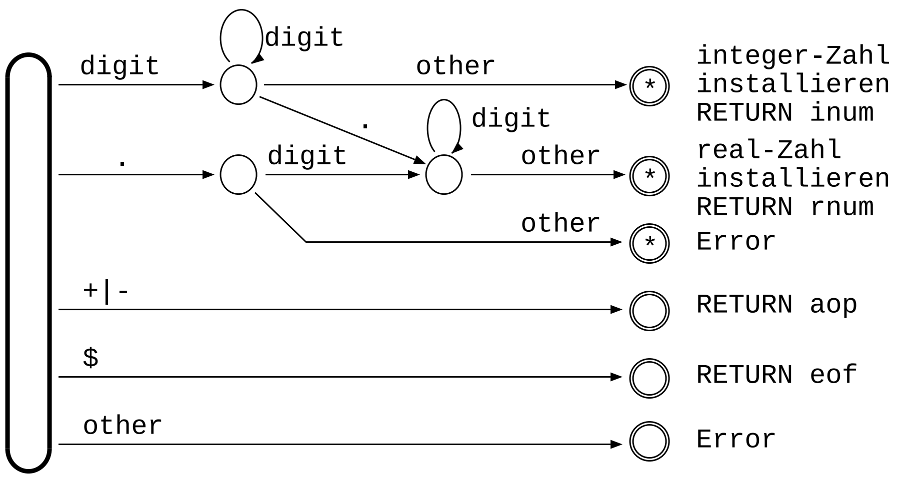
```
          /* Definition von Symbolen */
integer   = digits
real      = digits? pointgroup digits?

          /* Aktionen */
integer   : installInum(index); return( inum, index)
real      : installRnum(index); return( rnum, index)
```

	Eingabezeichen		
Zustand	`digit`	`"."`	Aktionen
1	2	3	.
∘ 2	2	4	`install...; return( inum,...)`
3	4	.	.
∘ 4	4	.	`install...; return( rnum,...)`

Beispiel 3.12: Bsp. 3.2 und Bsp. 3.4 werden um Aktionen erweitert

Die Analyse wird einfacher, wenn von Endzuständen keine Kanten wegführen. Bei Lexemen ohne erkennbare Endezeichen (z.B. Zahlen) muß man das erste Zeichen, das nicht mehr zum Lexem gehört, als Endezeichen (*other*) verwenden. Damit wird aber ein Zeichen „zuviel" gelesen, das für das nächste Symbol wieder berücksichtigt werden muß. Im Bsp. 3.13 sind solche Endzustände mit `*` markiert.



Beispiel 3.13: Ein DFA zur Analyse von Zahlen (wie in Bsp. 3.3) und Additionsoperatoren. Der Automat kann z.B einen Ausdruck `17+3.14-.32$` lexikalisch analysieren.

Bei einem erkannten Fehler (`Error`) wird im einfachsten Fall ein „Fehlertoken" `err` zurückgeliefert und eventuell im Attribut zusätzliche Information übergeben.

Eine aufwendige Fehlerbehandlung in der Lexikalischen Analyse selbst ist nicht besonders sinnvoll, da der jeweilige Kontext nicht bekannt ist.

`$` wird als eof-Zeichen verwendet; es darf in der Sprachdefinition nicht vorkommen.

Tabellengesteuerte Analyse

Zugrunde liegt ein Kontroll-Algorithmus, der für alle Automaten gleichbleibt. Mittels der jeweiligen Zustandsmatrix (Tabelle) wird die Analyse gesteuert. Dieses Prinzip ist insbesondere für Scanner-Generatoren geeignet. Das Problem des „Zuweit-Lesens“ kann man dabei auf zwei Arten lösen:

- Das überschüssige Zeichen wird *zurückgeschrieben* (siehe Alg. 3.14).
- Es wird grundsätzlich ein Zeichen *vorausgelesen* (siehe Alg. 3.15).

```
public void scan1() {
  state.set(initialState); // Ausgangszustand
  while( !finalStates.member(state.current) ) {
    nextchar(); // nächstes Zeichen lesen
    state.set(transTab[state.current][input]); // neuer Zustand
  }
  if( state.marked ) writeback( input ); // zurückschreiben
}
```

Algorithmus 3.14: Ein überschüssiges Zeichen wird zurückgeschrieben.

```
public void scan2() {
  state.set(initialState);
  while( !finalStates.member(state.current) ) {
    state.set(transTab[state.current][input]); // neuer Zustand
    if( !state.marked ) nextchar(); // nächstes Zeichen
  }
}
```

Algorithmus 3.15: Ein Zeichen wird grundsätzlich vorausgelesen.

Die Hilfsprozedur `nextchar` liest das nächste Eingabezeichen in die Variable `input`. Die Prozedur in Alg. 3.15 erwartet im Gegensatz zu Alg. 3.14, daß vor ihrem Aufruf bereits ein Zeichen gelesen wurde.

Die angegebenen Prozeduren erkennen jeweils ein Symbol. Am Ende der Prozeduren kann daher die dem Endzustand `state` zugeordnete Aktionsfolge ausgeführt werden. Zur vollständigen Lexikalischen Analyse eines Quellprogramms sind die Prozeduren wiederholt aufzurufen. Bei Alg. 3.15 muß dann vor dem ersten Aufruf ein Zeichen gelesen werden.

Ausprogrammieren des Automaten

Aus einem Zustandsdiagramm (Tabelle) kann man ein Programm erzeugen, wobei Zustände durch Programmstellen repräsentiert werden, auf die in Abhängigkeit vom nächsten Eingabezeichen verzweigt wird. An denjenigen Stellen, die Endzuständen entsprechen, setzt man die zugehörigen Aktionen ein (Bsp. 3.16).

```
public Token scan3() {
  switch( input ) {
    case "0": ... case "9":
      while( isDigit( input )) nextchar();
      if( input == "." {
        nextchar();
        while( isDigit( input )) nextchar();
        return "rnum";
      } else return "inum";
      break;
    case ".":
      nextchar();
      if( isDigit( input ) {
        nextchar();
        while( isDigit( input )) nextchar();
        return "rnum";
      } else return "err";
      break;
    case "+": nextchar(); return "aop"; break;
    case "-": nextchar(); return "aop"; break;
    case "$": return "eof"; break;
    default: nextchar(); return "err";
  }
}
```

Beispiel 3.16: Aus dem Diagramm von Bsp. 3.13 programmierter Scanner. Zur besseren Übersicht ist hier nur die Ausgabe der Token (ohne Attribute) eingetragen.

Vor dem ersten Aufruf von `scan3` muß `nextchar()` aufgerufen werden, um das erste Zeichen bereitzustellen. Auch nach Erkennen der Operatoren (`+`,`-`) und beim letzten Fehlerzustand wird `nextchar()` aufgerufen, da diese Endzustände nicht markiert sind.

Eine vollständige ausprogrammierte Lexikalische Analyse nach dem obigen Muster findet man im Anhang A.

Kapitel 4

Syntax-Analyse

Die Syntax-Analyse (*parsing*) untersucht, ob die von der Lexikalischen Analyse gelieferte Symbolfolge der Grammatik der Programmiersprache entspricht und liefert Informationen über die Struktur des analysierten Programms. Die Attribute werden dabei nicht benötigt, sondern an die Semantische Analyse weitergereicht.

4.1 Grundlagen

Die Syntax von Programmiersprachen wird durch **kontextfreie Grammatiken** beschrieben. Die Grammatik spezifiziert nicht nur die Menge der syntaktisch richtigen Programme, sondern auch die *Struktur* dieser Programme.

```
(1) F → id
(2) F → id ( L )
(3) L → L , L
(4) L → num
(5) L → F
```

Beispiel 4.1: Grammatik für Funktionsaufrufe in Backus-Naurform (BNF).

Eine kontextfreie Grammatik G = (T,N,P,S) besteht aus:

Terminalen T (z.B. `id`, `(`, `)`, ...). Sie können nicht weiter abgeleitet werden (daher der Name).

Nonterminalen N (z.B. `F` und `L`), mit großem Anfangsbuchstaben. Bei gut konstruierten Grammatiken entspricht jedes Nonterminal einer syntaktischen Klasse (in Bsp. 4.1 Funktionsaufruf bzw. Parameterliste).

Produktionen P (z.B. `L` → `L,L`), auf ihrer linken Seite steht genau ein Nonterminal, die rechte Seite besteht aus einem Wort. Mehrere Produktionen für dasselbe Nonterminal können auch zusammengefaßt werden (z.B. `L` → `L,L | num | id`).

Startsymbol S Wird das Startsymbol nicht explizit angegeben, so ist das Nonterminal der linken Seite der *ersten* Produktion das Startsymbol (im Bsp. 4.1: `F`).

Ein **Wort** ist eine beliebige Folge (Verkettung) von Nonterminalen und Terminalen, z.B. `id L, num`. Es wird mit griechischen Kleinbuchstaben (α, β, $\ldots$) bezeichnet. Die leere Folge (ε) nennt man *Leerwort*. Eine **Ableitung** ist die Anwendung von Produktionen auf ein Wort derart, daß ein Nonterminal im Wort, das der linken Seite einer Produktion entspricht, durch deren rechte Seite ersetzt wird. Bei **Linksableitungen** wird jeweils das am weitesten links stehende Nonterminal des Wortes ersetzt.

Umgekehrt ist eine **Reduktion** die hintereinander ausgeführte Ersetzung von Teilworten, die rechten Seiten von Produktionen entsprechen, durch die zugehörigen Nonterminale. Die Schritte einer Linksreduktion entsprechen den Schritten einer Rechtsableitung in umgekehrter Folge.

$$\begin{array}{ll}
\underline{\texttt{F}} \Longrightarrow & \texttt{id (} \underline{\texttt{num}} \texttt{ , id)} \Longleftarrow \\
\texttt{id (} \underline{\texttt{L}} \texttt{)} \Longrightarrow & \texttt{id (L , } \underline{\texttt{id}} \texttt{)} \Longleftarrow \\
\texttt{id (} \underline{\texttt{L}} \texttt{ , L)} \Longrightarrow & \texttt{id (L , } \underline{\texttt{F}} \texttt{)} \Longleftarrow \\
\texttt{id (num , } \underline{\texttt{L}} \texttt{)} \Longrightarrow & \texttt{id (} \underline{\texttt{L , L}} \texttt{)} \Longleftarrow \\
\texttt{id (num , } \underline{\texttt{F}} \texttt{)} \Longrightarrow & \underline{\texttt{id (L)}} \Longleftarrow \\
\texttt{id (num , id)} & \texttt{F}
\end{array}$$

Beispiel 4.2: Linksableitung $\texttt{F} \stackrel{*}{\Longrightarrow} \texttt{id (num , id)}$ und Linksreduktion $\texttt{id (num , id)} \stackrel{*}{\Longleftarrow} \texttt{F}$ mit der Grammatik aus Bsp. 4.1. Die Teilworte, die im nächsten Schritt ersetzt werden, sind unterstrichen.

Das Ergebnis von Ableitungen bzw. Reduktionen kann durch einen **Ableitungsbaum** (*parse tree*) strukturiert dargestellt werden (siehe Abb. 4.3).

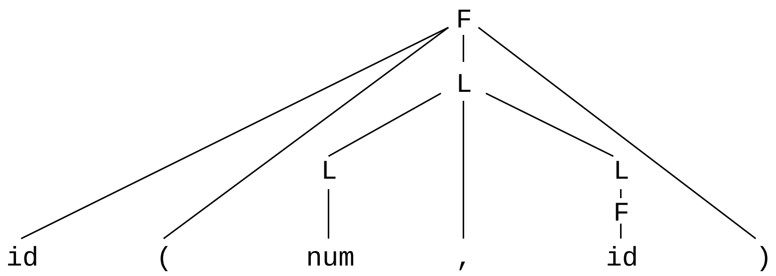


Abbildung 4.3: Ableitungsbaum zu Bsp. 4.2

Die Innenknoten entsprechen den Nonterminalen (Wurzel = Startsymbol). Ein Innenknoten mit seinen Nachfolgern entspricht der Anwendung einer Produktion.

Von links nach rechts gelesen ergeben die Blätter das abgeleitete Wort. Eine **Satzform** ist ein Wort α , das vom Startsymbol `S` aus ableitbar ist ($\mathtt{S} \overset{*}{\Longrightarrow} \alpha$), z.B. `id ( L , L )` . Ein **Satz** ist eine *Satzform*, die nur aus Terminalen besteht, also das 'Endergebnis' einer Ableitung, z.B. `id ( num , id )`. Bei der Syntax-Analyse sind die Sätze die syntaktisch richtigen Programme. Die von einer Grammatik erzeugte **Sprache** L(G) ist die Menge all ihrer Sätze. Eine Programmiersprache ist damit die Menge aller ableitbaren Programme.

Äquivalenz von Grammatiken

Zwei verschiedene Grammatiken G_1 und G_2, die dieselbe Sprache L beschreiben, nennt man **äquivalent**. Aufgrund derartiger Äquivalenzen können mittels *Umformungen* bestimmte, für das gewählte Analyseverfahren (siehe Kap. 4.2) oder für nachfolgende Phasen (z.B. Semantische Analyse) ungünstige Eigenschaften von Grammatiken entfernt werden.

Alle Grammatiken in Bsp. 4.4 außer G1 sind **eindeutig** (d.h. für jede Satzform gibt es genau einen Ableitungsbaum). G1 ist **mehrdeutig**: es gibt Satzformen (z.B. `num,num,num`) mit verschiedenen Ableitungsbäumen (Abb. 4.5).

Die Grammatik G2 erzeugt „linksrekursive" Bäume (Abb. 4.6 links), G3 jedoch „rechtsrekursive" Bäume (Abb. 4.6 rechts). Beide eignen sich zum Beschreiben von *listen*artigen Strukturen, bei denen nur die *Reihenfolge* der Blätter wichtig ist. Bei *Ausdrücken* mit links- bzw. rechtsassoziativen Operatoren soll die Links- bzw. Rechtsrekursion verwendet werden – der Baum bildet dann die Struktur richtig ab.

Die Grammatik G4 (Abb. 4.7 verwendet Iterationen, dargestellt mit regulären Ausdrücken. Sie ist anschaulich, jedoch kann im 'Ableitungsbaum' die Anzahl der Nachfolger eines Knoten statisch nicht bestimmt und die Struktur von Ausdrücken nicht erkannt werden.

Bei der Grammatik G5 wird die Iteration mit einer Rekursion in einem zusätzlichen Nonterminal („Restliste") aufgelöst – das erste Terminal (`num` bei `L`) bekommt eine Sonderstellung. In G6 kommt es zu einer indirekten Rekursion. Mit beiden können Iterationen durch Knoten mit fester Nachfolgerzahl dargestellt werden (Abb. 4.8).

Regular Right Part Grammars (RRPG)

Die Syntax vieler Programmiersprachen wird unter Verwendung von Regulären Ausdrücken auf den rechten Seiten sehr einfach und klar beschrieben. Die syntaktischen Klassen sind deutlich erkennbar (in Bsp. 4.9 Ausdruck, Term, Faktor). Im Unterschied zur Regulären Definition sind Rekursionen erlaubt. Es folgt eine Gegenüberstellung dieser Konstrukte und gleichwertiger BNF-Konstrukte. Die Umformung ist automatisch durchführbar; die Iterationen werden dabei in Rechtsrekursionen mit neuen Zusatz-Nonterminalen aufgelöst.

Iteration	$\ldots\alpha * \ldots$	$\Longrightarrow$	$\ldots\mathtt{N}\ldots$	mit neuer Prod. $\mathtt{N}\rightarrow\alpha\mathtt{N} \mid \varepsilon$
Iteration	$\ldots\alpha\mathtt{+}\ldots$	$\Longrightarrow$	$\ldots\alpha\mathtt{N}\ldots$	mit neuer Prod. $\mathtt{N}\rightarrow\alpha\mathtt{N} \mid \varepsilon$
Option	$\ldots\alpha\,?\ldots$	$\Longrightarrow$	$\ldots\mathtt{N}\ldots$	mit neuer Prod. $\mathtt{N}\rightarrow\alpha \mid \varepsilon$

G1 (doppelte Rekursion):	`L` $\rightarrow$	`L , L` $\mid$ `num`
G2 (Linksrekursion):	`L` $\rightarrow$	`L , num` $\mid$ `num`
G3 (Rechtsrekursion):	`L` $\rightarrow$	`num , L` $\mid$ `num`
G4 (reguläre Schreibweise):	`L` $\rightarrow$	`num (, num)*`
G5 (Restlistenrekursion):	`L` $\rightarrow$	`num` $\mathtt{L}_r$
	$\mathtt{L}_r$ $\rightarrow$	`, num` $\mathtt{L}_r$ $\mid \varepsilon$
G6 (alternierende Rekursion):	`L` $\rightarrow$	`num` $\mathtt{L}_r$
	$\mathtt{L}_r$ $\rightarrow$	`, L` $\mid \varepsilon$

Beispiel 4.4: Äquivalente Grammatiken (Zeilen (3) und (4) aus Bsp. 4.1)

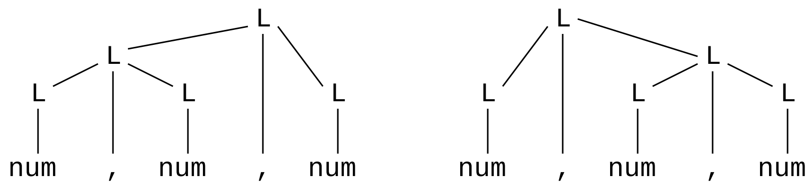


Abbildung 4.5: Ableitungsbäume für G1

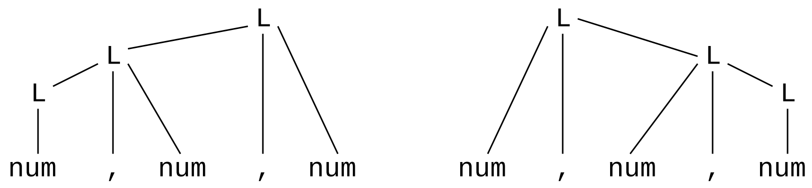


Abbildung 4.6: Ableitungsbäume für G2 und G3

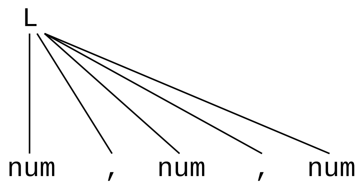


Abbildung 4.7: Ableitungsbaum für G4

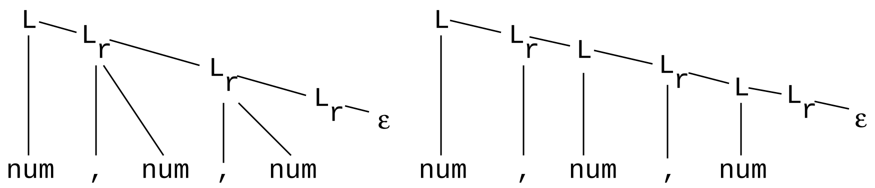


Abbildung 4.8: Ableitungsbäume für G5 und G6

Dabei besteht α aus einem Symbol oder einer geklammerten Folge von Symbolen, `N` ist jeweils ein neues Nonterminal.

$$
\begin{array}{lcl}
\mathtt{E} \rightarrow \mathtt{T}\ (\mathtt{+}\ \mathtt{T})^* & \Longrightarrow & \mathtt{E} \rightarrow \mathtt{T}\ \mathtt{E}_r \\
 & & \mathtt{E}_r \rightarrow \mathtt{+}\ \mathtt{T}\ \mathtt{E}_r \mid \varepsilon \\
\mathtt{T} \rightarrow \mathtt{F}\ (\mathtt{*}\ \mathtt{F})^* & \Longrightarrow & \mathtt{T} \rightarrow \mathtt{F}\ \mathtt{T}_r \\
 & & \mathtt{T}_r \rightarrow \mathtt{*}\ \mathtt{F}\ \mathtt{T}_r \mid \varepsilon \\
\mathtt{F} \rightarrow \mathtt{(}\ \mathtt{E}\ \mathtt{)} \mid \mathtt{id} & \Longrightarrow & \mathtt{F} \rightarrow \mathtt{(}\ \mathtt{E}\ \mathtt{)} \mid \mathtt{id}
\end{array}
$$

Beispiel 4.9: Automatische Umformung einer RRPG in eine BNF

In Bsp. 4.9 treten Klammern als Terminale und als Meta-Operatoren auf. Zur eindeutigen Unterscheidung werden Terminale daher oft zwischen Hochkommas (z.B. `"+"`,`"*"`,`"("`, etc.) geschrieben.

Eine andere Schreibweise von Regulären Ausdrücken auf der rechten Seite einer Produktion wird in der **EBNF** (erweiterte BNF) verwirklicht: Iterationen werden mit {}, Optionale mit [ ] beschrieben. Bekannt ist diese Schreibweise von der Modula-Spezifikation.

Operatoren und Trenner

In fast allen Programmiersprachen gibt es **Ausdrücke**, deren *Struktur*, beruhend auf der Assoziativität und Priorität ihrer **Operatoren**, wichtig ist (spätestens bei der Code-Erzeugung). Für solche Ausdrücke sollten die passenden Produktionen gewählt werden (z.B. Linksrekursion, s.o.).

Andererseits gibt es **Listen** und listenähnliche Konstrukte (z.B. Parameterlisten, Anweisungsfolgen), bei denen nur die *Reihenfolge* der einzelnen Elemente wichtig ist. Dabei werden die 'Operatoren' als **Trenner** bezeichnet, da sie nur zur Trennung der Elemente dienen, z.B. `S;S;S`.

Analyseverfahren

Die Grammatiken, die in erster Linie für die *Sprachbeschreibung* gedacht sind, können auch für die *Sprachanalyse* verwendet werden. Dabei wird untersucht, ob ein gegebenes Wort in einer bestimmten Sprache enthalten ist, d.h. ob das Wort mit der Grammatik ableitbar ist; man sagt, das Wort wird akzeptiert. Ähnlich wie bei der Lexikalischen Analyse wird diese Aufgabe meist durch Automaten erledigt. Man kann sie als Interpreter sehen, die als (Steuer-)Programm eine Grammatik verwenden. Außerdem benötigen sie einen Keller zum Speichern der Geschichte der Ableitung (daher auch der Name **Keller-Automat**).

Um die Analyse zu vereinfachen, wird die Grammatik erweitert: Den Produktionen wird `S` $\rightarrow$ Startsymbol `$` hinzugefügt (z.B. `(0) S` $\rightarrow$ `F $` für Bsp. 4.1). `S` ist ein neues Startsymbol, das in der Grammatik sonst nicht vorkommt. `$` ist ein spezielles *eof-Symbol*, mit dem jedes Eingabewort abgeschlossen ist und das als zusätzliches neues Terminalsymbol gesehen werden kann. Diese einfache Erweiterung findet bei jedem Analysator statt, wird in der Folge aber nur dann explizit angeführt, wenn das zum besseren Verständnis nötig ist.

Für die im folgenden behandelten Analyseverfahren werden zwei grundlegende Funktionen benötigt:

`First(`α`)` gibt für eine Grammatik an, mit welchen Terminalsymbolen die aus α ableitbaren Satzformen beginnen können, bzw. ob α nach ε ableitbar ist. Die `First`-Mengen aller Grammatiksymbole (Terminale und Nonterminale) können gemeinsam iterativ berechnet werden (siehe Alg. 4.10).

`First(`$s_1\ s_2 \ldots s_k$`)` wird äquivalent zur Produktion `N` $\rightarrow s_1\ s_2 \ldots s_k$ gebildet, wobei `First(N)` durch `First(`$s_1\ s_2 \ldots s_k$`)` zu ersetzen ist.

`Follow(N)` gibt für eine Grammatik an, welche Terminalsymbole dem Nonterminal `N` in einer beliebigen Satzform unmittelbar folgen können, bzw. ob `eof` (`$`) folgen kann. Die `Follow`-Mengen aller Nonterminale einer Grammatik können ebenfalls iterativ berechnet werden (siehe Alg. 4.11).

4.2 Top-Down-Analyse

Es wird versucht, mittels einer gegebenen Grammatik einen Ableitungsbaum für eine gegebene Symbolfolge zu finden. Dabei wird **„absteigend“** (*top down*) vorgegangen (Abb. 4.12): Ausgehend vom Startsymbol, das die Wurzel des Ableitungsbaumes wird, und vom linken Rand der Symbolfolge werden solange **Linksableitungen** durchgeführt, bis die ganze Symbolfolge (d.h. das Eingabewort) abgeleitet wurde oder feststeht, daß sie nicht abgeleitet werden kann.

Tabellengesteuerte Top-Down-Analyse

Die Analyse wird durch einen allgemeinen Kontroll-Algorithmus (Alg. 4.14) mithilfe von Tabellen (wie in Bsp. 4.13) durchgeführt. Jeder Ableitungsschritt hängt vom *obersten* Kellersymbol und vom *vordersten* Eingabesymbol ab. Für jedes Nonterminal gibt es eine Zeile in der Tabelle. Sie gibt an, welche Produktion in Abhängigkeit von der Eingabe anzuwenden ist. Leereinträge (.) stellen Fehler dar.

$\forall$ Terminalsymbole `t`: `First(t) =` $\{$`t`$\}$
$\forall$ Nonterminale `N`: `First(t) =` $\{\}$
`do` $\{$
  `if(` es gibt eine Produktion `N` $\rightarrow \varepsilon$ `)` füge ε zu `First(N)` hinzu
  `if(` es gibt eine Produktion `N` $\rightarrow$ s_1 s_2 … s_k `)`
    füge die *Terminalsymbole* aus `First(`s_1`)` zu `First(N)` hinzu
  `i=1;`
  `while( i<k &&` $\varepsilon \in$ `First(`s_i`) )` $\{$
    füge die *Terminalsymbole* aus `First(`s_{i+1}`)` zu `First(N)` hinzu
    `i=i+1;`
  $\}$
  `if(` s_1 s_2 … $s_k \overset{*}{\Longrightarrow} \varepsilon$ `)` füge ε zu `First(N)` hinzu
$\}$ `while( First`-Menge hat sich geändert `)`

Algorithmus 4.10: Berechnung der `First`-Mengen aller Grammatiksymbole.

$\forall$ alle Nonterminale `N`: `DO Follow(N) =` $\{\}$
`Follow`(Startsymbol) `=` $\{\$\}$
`do` $\{$
  `if(` es gibt eine Produktion `A` $\rightarrow$ α `N` oder `A` $\rightarrow$ `N )`
    füge die Elemente aus `Follow(A)` der `Follow`-Menge von `N` hinzu
  `if(` es gibt eine Produktion `A` $\rightarrow \alpha$ `N` β oder `A` $\rightarrow$ `N` β $\{$
    füge die *Terminalsymbole* aus `First(`β`)` zur `Follow`-Menge von `N`
    `if(` $\varepsilon \in$ `First(`β`) )`
      füge die Elemente aus `Follow(A)` der `Follow`-Menge von `N` hinzu
  $\}$
$\}$ `while( Follow`-Menge hat sich geändert `)`

Algorithmus 4.11: Berechnung der `Follow`-Mengen aller Nonterminale einer Grammatik.

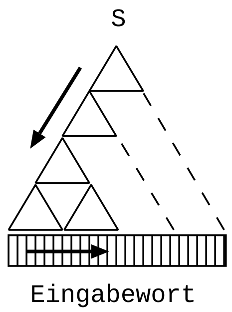


Abbildung 4.12: Top-Down-Analyse

Konstruktion der Tabellen. Die Top-Down-Tabelle wird durch die Matrix `M[N,t]` mit den Indexbereichen Nonterminale = Kellersymbole (für die Zeilen) und Eingabesymbole (für die Spalten) dargestellt. Sie wird durch den Alg. 4.15 gefüllt. Bsp. 4.16 zeigt die Anwendung des Verfahrens.

Bei der Ableitung eines Nonterminals `N` wird jene Produktion `N` $\rightarrow \alpha$ gewählt, die zum vordersten Eingabesymbol `t` hinführt. Es muß `t` $\in$ `First(`α`)` sein; läßt sich allerdings α nach ε ableiten, so muß `t` $\in$ `Follow(N)` sein.

Rekursiver Abstieg (recursive descent)

Bei der Methode des Rekursiven Abstiegs (*recursive descent*) wird für jedes Nonterminal der Grammatik eine Prozedur gebildet (siehe Bsp. 4.17).

Als Optimierung läßt sich die Rekursion von `R` durch eine *Iteration* ersetzen und statt des Aufrufs direkt in `E` eintragen. Die Prozedur `E` hat dann das Aussehen:

```
... T(); while( sym == op ) { nextsym(); T(); }
```

Sie entspricht genau der RRP-Produktion `E` = `T ( op T )`*. So lassen sich RRP-Grammatiken ohne Umweg über BNF zu Parsern ausprogrammieren. Leider lassen sich nicht alle Grammatiken direkt mit den obigen Methoden behandeln. In der Tabelle kann es zu Mehrfacheinträgen kommen; im ausprogrammierten Parser kann der Fall auftreten, daß die Verzweigung für einen Wert von `sym` nicht eindeutig ist.

LL(1)-Grammatiken

Eine Grammatik, deren Top-Down-Tabelle keine Mehrfacheinträge hat, nennt man LL(1): sie analysiert Wörter von **L**inks nach rechts mittels **L**inksableitung und einer Vorausschau in der Eingabe von **1** Symbol.

Eine Grammatik ist genau dann LL(1), wenn für die Alternativen α_i der Produktionen für jedes `N` gilt:

1. `First(`α_i`)` $\cap$ `First(`α_j`) = {}` f.a. `i,j (i`$\neq$`j)`

Grammatik:

$E \rightarrow T\ E_r$
$E_r \rightarrow +\ T\ E_r \mid \varepsilon$
$T \rightarrow F\ T_r$
$T_r \rightarrow *\ F\ T_r \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Tabelle:

oberstes Kellersymbol	vorderstes Eingabesymbol id	(	)	+	*	$
E	T E_r	T E_r	.	.	.	.
E_r	.	.	ε	+ T E_r	.	ε
T	F T_r	F T_r	.	.	.	.
T_r	.	.	ε	ε	* F T_r	ε
F	id	(E)	.	.	.	.

Analyse des Wortes `id + id * id $`:

Keller	verbleibendes Eingabewort	anzuwendende Produktion
$ E	id + id * id $	$E \rightarrow T\ E_r$
$ E_r T	id + id * id $	$T \rightarrow F\ T_r$
$ E_r T_r F	id + id * id $	$F \rightarrow$ id
$ E_r T_r id	id + id * id $	
$ E_r T_r	+ id * id $	$T_r \rightarrow \varepsilon$
$ E_r	+ id * id $	$E_r \rightarrow +\ T\ E_r$
$ E_r T +	+ id * id $	
$ E_r T	id * id $	$T \rightarrow F\ T_r$
$ E_r T_r F	id * id $	$F \rightarrow$ id
$ E_r T_r id	id * id $	
$ E_r T_r	* id $	$T_r \rightarrow *\ F\ T_r$
$ E_r T_r F *	* id $	
$ E_r T_r F	id $	$F \rightarrow$ id
$ E_r T_r id	id $	
$ E_r T_r	$	$T_r \rightarrow \varepsilon$
$ E_r	$	$E_r \rightarrow \varepsilon$
$	$	

Beispiel 4.13: Tabellengesteuerte Top-Down-Analyse

```
for(;;) {
  sym = vorderstes Eingabesymbol
  switch( oberstes Kellersymbol ) {
    case $: if( sym == $ ) accept                  // Analyse richtig beendet
      else Error ...                             // es gibt noch Eingabesymbole
      break;
    case Terminal t: if( t == sym ) skip              // verbrauche Terminal
      else Error                       // erwartetes Terminal nicht in der Eingabe
      break;
    case Nonterminal N: if( table[N][sym] == α )
        lösche N vom Keller
        schreibe α in umgekehrter Reihenfolge auf den Keller
      else Error ...                        // es gibt keine passende Produktion
    }
  }
```

Algorithmus 4.14: Allgemeiner Kontroll-Algorithmus mit Tabellen.

```
Setze alle Einträge M[N][t] auf error
Für jede Produktion N → α und jedes Terminalsymbol t:
  if( Terminalsymbol t ∈ First(α) ) füge α zu M[N][t] hinzu
  if( ε ∈ First(α) && t ∈ Follow(N) ) füge α zu M[N][t] hinzu
```

Algorithmus 4.15: Füllen der Top-Down-Tabelle.

2. höchstens ein α_i läßt sich nach ε ableiten

3. falls $\alpha_i \stackrel{*}{\Longrightarrow} \varepsilon$: First(α_j) $\cap$ Follow(N)={} f.a. j$\neq$i

Die dritte Bedingung besagt: Läßt sich eine der Alternativen eines Nonterminals N zum Leerwort (ε) ableiten, so darf keine der anderen Alternativen für N mit einem Element aus Follow(N) anfangen (siehe Bsp. 4.18).

Bei der Ableitung von op id ist nicht eindeutig, welche Produktion von $\mathtt{Opd}_0$ anzuwenden ist, wenn id das vorderste Eingabesymbol ist.

LL-Grammatiken haben den Vorteil, daß bei jedem Analyseschritt klar ist, welche Aktion als nächste zu treffen ist: welche Produktion anzuwenden ist, bzw. ob ein Eingabesymbol zu verbrauchen ist, bzw. ob die Analyse richtig oder falsch beendet ist. Die Analyse läuft *deterministisch* ab und ist daher zeiteffizient. LL(1)-Analysen sind viel speichereffizienter als solche mit mehr Vorausschau (LL(k) mit k>1).

Grammatik:

```
(1) E → T Er     (4) T → F Tr       (7) F → ( E )
(2) Er→ + T Er   (5) Tr→ * F Tr     (8) F → id
(3) Er→ ε        (6) Tr→ ε
```

First- und Follow-Mengen:

```
First(E)  = {id,(}   Follow(E)  = {),$}
First(Er) = {+,ε}    Follow(Er) = {),$}
First(T)  = {id,(}   Follow(T)  = {+,),$}
First(Tr) = {*,ε}    Follow(Tr) = {+,),$}
First(F)  = {id,(}   Follow(F)  = {*,+,),$}
```

Einträge für das Nonterminal E: Zu betrachten ist nur die Produktion (1): First(T E_r) = {(,id} – unter (und unter id wird die rechte Seite der Regel (1), d.h. T E_r, eingetragen. Die anderen Einträge der Zeile E (für), +, * und $) bleiben leer – sie zeigen Syntaxfehler an.

Einträge für das Nonterminal E_r: Zuerst Produktion (2): First(+ T E_r) = {+} - deshalb wird in M[E_r,+] die rechte Seite (+ T E_r) eingetragen; dann (3): $\varepsilon \in$ First(ε) – also Follow(E_r) bestimmen: das ist {),$}. Also steht unter) und unter $ die rechte Seite: ε. Die Einträge für die restlichen Nonterminale werden analog bestimmt.

Das Erstellen der Tabelle kann maschinell (mit einem Parser-Generator) erfolgen.

Beispiel 4.16: Erstellen der Tabelle von Bsp. 4.13.

Top-Down-Analyse mit nicht-LL(1)-Grammatiken

Steht oben im Keller Stm und vorne in der Eingabe id (siehe Bsp. 4.19), so ist nicht klar, welche Produktion angewendet werden muß. Lösungsmöglichkeiten sind:

- Vorausschau um ein weiteres Eingabesymbol – also LL(2)-Analyse
- Verwenden semantischer Informationen (z.B. Symboltabelle) – das führt zu einer Vermischung mit der Semantischen Analyse
- Rücksetzen im Kontroll-Algorithmus

Rekursiver Abstieg mit Backtracking

Zum Speichern der Geschichte der bisherigen Analyse ist ein Keller nötig, der nicht nur die noch abzuleitenden Symbole (wie bei der tabellengesteuerten Analyse), sondern den *gesamten* Ableitungsbaum und in jedem Innenknoten die zuletzt versuchte

```
E → T R                                                    // expression
R → op T R | ε                                     // rest of expression
T → ( E ) | id                                                   // term

class RecursiveDescent {
  Token sym;
  public void nextsym() ...    // setzt sym auf das nächste Eingabesymbol
  public void E() {                                        // expression
    if( sym.member("id,(") ) { T(); R(); } else Error...
  }
  public void R() {                                // rest of expression
    if( sym.equals("op") ) { nextsym(); T(); R(); }
    else if( sym.member("),$")) /* ε (leeres statement) */
    else Error...
  }
  public void T() {                                              // term
    if( sym.equals("(") ) { nextsym(); E(); nextsym(); }       // ")"
    else if( sym.equals("id") ) nextsym();
    else Error...
  }
  void main() {                                   // Recursive Descent
    nextsym();
    E();
  }
```

Beispiel 4.17: Recursive-descent-Parser für einen Teil der Grammatik von Bsp. 4.16

$$\texttt{Opd} \rightarrow \texttt{op Opd}_0\ \texttt{Opd} \mid \texttt{id}$$
$$\texttt{Opd}_0 \rightarrow \texttt{Opd} \mid \varepsilon$$

$$\texttt{First(Opd)} \cap \texttt{Follow(Opd}_0\texttt{)} = \{\texttt{op,id}\} \neq \{\}$$

Beispiel 4.18: Die folgende Grammatik für Präfix-Ausdrücke mit ein- und zweistelligen Operatoren ist nicht LL(1), da die 3.Bedingung nicht erfüllt wird.

```
(1) Stm → V = E        /* Zuweisung */
(2) Stm → id ( L )     /* Prozeduraufruf */
(3) Stm → id           /* Prozeduraufruf ohne Parameter */
(4) V → id             /* Variable */
(5) V → id [ L ]       /* indizierte Variable */
    E → ...            /* Ausdruck */
    L → ...            /* Parameterliste */
```

Beispiel 4.19: Grammatik zur Beschreibung von Anweisungen

Alternative speichert. Derartige recursive-descent-Parser untersuchen prinzipiell alle Ableitungsmöglichkeiten (siehe Bsp. 4.20). Sie sind mit Sprachen, die Backtracking oder Parallelität erlauben (siehe Anhang B), besonders einfach zu schreiben, da der Keller bereits implizit in der Sprache vorhanden ist.

```
Stm  ⟹ V = E    ⟹ id = E            = paßt nicht → rücksetzen
                 ⟹ id [ L ] = E      [ paßt nicht → rücksetzen
     ⟹ id ( L )
```

Beispiel 4.20: Analyse von `id ( L ) $` mittels Backtracking.

Bestimmte Grammatiken, die die LL-Bedingung nicht erfüllen, können durch folgende Algorithmen in LL(1)-Grammatiken umgewandelt werden:

Linksfaktorisierung. Gleiche Anfänge der Alternativen eines Nonterminals werden „herausgehoben“, für den Rest wird ein neues Nonterminal eingeführt. Stehen dabei Nonterminale am Anfang, ist es manchmal nötig, sie durch ihre rechten Seiten zu ersetzen (siehe Bsp. 4.21).

Linksrekursionen. Linksrekursive Grammatiken beschreiben linksassoziative Operatoren gut, sind aber für Top-Down-Verfahren nicht geeignet: bei Verfahren mit (beliebiger) Vorausschau ist der Schnitt der `First`-Mengen nicht leer, selbst Verfahren mit Rücksetzen fallen bei fehlerhaften Eingabeworten in Endlos-Linksrekursionen.

Linksrekursionen lassen sich auf folgende Weise in LL(1)-Grammatiken umwandeln:

$$\mathtt{A} \rightarrow \mathtt{A}\ \alpha \mid \beta \Longrightarrow \quad \begin{array}{l} \mathtt{A} \rightarrow \beta\ \mathtt{A}_r \\ \mathtt{A}_r \rightarrow \alpha\ \mathtt{A}_r \mid \varepsilon \end{array}$$

Dabei können Probleme entstehen, wenn `A` auch in α oder β vorkommt.

$$\texttt{V} \rightarrow \texttt{id}\ \texttt{V}_r \qquad \texttt{V}_r \rightarrow \varepsilon$$
$$\texttt{V}_r \rightarrow \texttt{[L]}$$

Auch die rechten Seiten der Produktionen für `Stm` haben gleiche Elemente in ihren `First`-Mengen. Zuerst wird `V` ersetzt, dann wird wieder faktorisiert:

$$\texttt{Stm} \rightarrow \texttt{id}\ \texttt{Stm}_r \qquad \texttt{Stm}_r \rightarrow \texttt{V}_r\ \texttt{=}\ \texttt{E}$$
$$\texttt{Stm}_r \rightarrow \texttt{(L)}$$
$$\texttt{Stm}_r \rightarrow \varepsilon$$

Nun ist die Grammatik LL(1). Die Produktionen für `V` werden nicht mehr gebraucht.

Beispiel 4.21: Linksfaktorisierung für `V` aus Bsp. 4.19.

Fehlerbehandlung

LL-Parser erkennen einen Fehler (bzw. sein Symptom) an der frühestmöglichen Stelle, d.h. wenn sich der Rest der Eingabe nicht mehr ableiten läßt (**valid-prefix-Eigenschaft**). Natürlich kann die Fehlerursache weitaus früher liegen: z.B. verursacht das Weglassen von `(` in `id * ( id + id )` einen Fehler, der erst beim Lesen von `)` erkannt wird. Im Gegensatz dazu können Verfahren mit Rücksetzen die Fehlerstelle nicht bestimmen.

Die einfachste Fehlerbehandlung ist der Abbruch der Analyse nach einer genauen Fehlermeldung. Tabellengesteuerte Verfahren unterstützen das mit Informationen aus ihrem Keller. Er enthält die 'erwarteten' Terminalsymbole bzw. Nonterminale und ermöglicht auch *phrase level recovery*, eine bessere Methode, die alle Fehler findet und 'verbessert', aber Teile überliest bzw. Folgefehler erzeugt.

Zur Vorbereitung setzt man alle `error`-Einträge jedes Nonterminals mit einer ε-Alternative auch auf ε (siehe `(*)` in Bsp. 4.23). Diese falschen ε-Ableitungen werden später abgefangen. Bei den anderen Nonterminalen `N` ändert man die `error`-Einträge für die Folgesymbole ($\in$ `Follow(N)`) auf `stop`, die anderen auf `skip`. Die Analyse erfolgt durch Alg. 4.22 (vgl. Alg. 4.14).

Bei der ausprogrammierten LL-Analyse können die recursive-descent-Prozeduren ähnlich erweitert werden. Man kann dabei *dynamische* (d.h. vom Input abhängige) Informationen mittels Parametern weiterreichen.

Es gibt globale Strategien, die nicht nur die nächste Umgebung des Fehlers (vorderstes Eingabesymbol, oberstes Kellersymbol) betrachten. Dabei wird versucht, mit Kostenmatrizen die Fehlerbehebung zu optimieren.

4.3 Bottom-Up-Analyse

Mittels einer gegebenen Grammatik wird versucht, einen Ableitungsbaum für eine gegebene Symbolfolge zu finden. Dabei wird „**aufsteigend**“ (*bottom up*) vorgegan-

```
for(;;) {
  sym = vorderstes Eingabesymbol
  switch( oberstes Kellersymbol {
    case $: if( sym.equals("$") ) accept              // Analyse beendet
      break;
    case Terminal t: if( sym.equals(t) ) skip    // verbrauche Terminal t
      else lösche t vom Keller           // „Einfügen“ von t im Eingabewort
      break;
    case Nonterminal N: if( table[N][sym] == α ) // ersetze N im Keller durch α
      else if( table[N][sym] == stop )
        lösche N vom Keller                  // erzwingt die Ableitung von N
      else skip                             // Überlesen des Eingabesymbols
      break;
  }
}
```

Algorithmus 4.22: Analyse mit Fehlerbehandlung.

gen: Ausgehend vom Eingabewort (= die von der Lexikalischen Analyse gelieferte Symbolfolge) werden solange **Linksreduktionen** durchgeführt, bis zum Startsymbol (= Wurzel des Ableitungsbaumes) reduziert wurde oder feststeht, daß das nicht möglich ist. Dabei wird, umgekehrt wie bei der Top-Down-Analyse, jeder Knoten des Ableitungsbaumes *nach* seinen Nachfolgern erzeugt (Abb. 4.24).

Bottom-Up-Verfahren haben den Vorteil, daß die zugrundeliegende Grammatik auch Linksrekursionen und gleiche Anfänge enthalten darf. Umformungen (Faktorisierung, Elimination von Linksrekursionen) sind *nicht* nötig.

Grundbegriffe

Die Bottom-Up-Analyse heißt auch **shift-reduce-Analyse**. Die wesentlichen Aktionen des Parsers sind das „Schieben“ eines Eingabesymbols in den Keller (*shift*) und das Reduzieren der obersten Kellersymbole, die der rechten Seite einer Produktion entsprechen, zu einem Nonterminal (*reduce*). Im Keller stehen, im Gegensatz zur Top-Down-Analyse, die abgeleiteten Grammatiksymbole.

Bei jedem Analyseschritt stellen daher Kellerinhalt und verbleibendes Eingabewort aneinandergereiht (d.h. die Symbolfolge zwischen `$` und `$`) eine *Satzform* dar, die mittels Rechtsableitungen gewonnen wird. Für eine entsprechende Rechtsableitung sind die Zeilen in Bsp. 4.25 von unten nach oben zu lesen.

Die unterstrichenen Teilworte in Bsp. 4.25 bezeichnet man als **Reduktionsansätze** (*handles*). Sie sind jene Symbolfolgen oben im Keller, die der rechten Seite einer Produktion entsprechen und deren Ersetzung einen *Schritt* der *entsprechenden Rechtsableitung* des Wortes darstellt.

In einer Satzform kann es Teilworte geben, die einer rechten Seite entsprechen,

(1) $E \to T\ E_r$	(4) $T \to F\ T_r$	(7) $F \to$ (E)
(2) $E_r \to$ + $T\ E_r$	(5) $T_r \to$ * $F\ T_r$	(8) $F \to$ id
(3) $E_r \to \varepsilon$	(6) $T_r \to \varepsilon$	

Zugehörige Tabelle mit Fehler-Aktionen:

oberstes Kellersymbol	vorderstes Eingabesymbol id	(	)	+	*	$
E	T E_r	T E_r	stop	skip	skip	stop
E_r	ε(*)	ε(*)	ε	+ T E_r	ε(*)	ε
T	F T_r	F T_r	stop	stop	skip	stop
T_r	ε(*)	ε(*)	ε	ε	* F T_r	ε
F	id	(E)	stop	stop	stop	stop

Die Analyse von `( + id * + id $` liefert `( id * F + id ) $` als richtiges Wort:

Keller	verbleibendes Eingabewort	Aktion
$ E	(+ id * + id $	
$ E_r T	(+ id * + id $	
$ E_r T_r F	(+ id * + id $	
$ E_r T_r) E (	(+ id * + id $	
$ E_r T_r) E	+ id * + id $	skip über +
$ E_r T_r) E	id * + id $	id ist richtiger
$ E_r T_r) E_r T	id * + id $	Anfang von E
$ E_r T_r) E_r T_r F	id * + id $	
$ E_r T_r) E_r T_r id	id * + id $	
$ E_r T_r) E_r T_r	* + id $	
$ E_r T_r) E_r T_r F *	* + id $	
$ E_r T_r) E_r T_r F	+ id $	stop Ableitung
$ E_r T_r) E_r T_r	+ id $	von F erzwingen
...		
$ E_r T_r) E_r	$	
$ E_r T_r)	$	) einfügen
$ E_r T_r	$	
$ E_r	$	
$	$	

Beispiel 4.23: Fehlerbehandlung mit LL(1)-Grammatik (vgl. Bsp. 4.16).

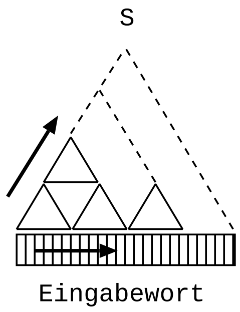


Abbildung 4.24: Bottom-Up-Analyse

$\texttt{E} \rightarrow \texttt{E + T} \mid \texttt{T}$
$\texttt{T} \rightarrow \texttt{T} * \texttt{F} \mid \texttt{F}$
$\texttt{F} \rightarrow \texttt{(E)} \mid \texttt{id}$

Keller	verbleibendes Eingabewort	anzuwendende Aktion
$\$$	$\texttt{id + id} * \texttt{id}\ \$$	shift
$\$\ \underline{\texttt{id}}$	$\texttt{+ id} * \texttt{id}\ \$$	reduce mittels $\texttt{F} \rightarrow \texttt{id}$
$\$\ \underline{\texttt{F}}$	$\texttt{+ id} * \texttt{id}\ \$$	reduce mittels $\texttt{T} \rightarrow \texttt{F}$
$\$\ \underline{\texttt{T}}$	$\texttt{+ id} * \texttt{id}\ \$$	reduce mittels $\texttt{E} \rightarrow \texttt{T}$
$\$\ \texttt{E}$	$\texttt{+ id} * \texttt{id}\ \$$	shift
$\$\ \texttt{E +}$	$\texttt{id} * \texttt{id}\ \$$	shift
$\$\ \texttt{E +}\ \underline{\texttt{id}}$	$* \texttt{id}\ \$$	reduce mittels $\texttt{F} \rightarrow \texttt{id}$
$\$\ \texttt{E +}\ \underline{\texttt{F}}$	$* \texttt{id}\ \$$	reduce mittels $\texttt{T} \rightarrow \texttt{F}$
$\$\ \texttt{E + T}$	$* \texttt{id}\ \$$	shift
$\$\ \texttt{E + T} *$	$\texttt{id}\ \$$	shift
$\$\ \texttt{E + T} * \underline{\texttt{id}}$	$\$$	reduce mittels $\texttt{F} \rightarrow \texttt{id}$
$\$\ \texttt{E +}\ \underline{\texttt{T} * \texttt{F}}$	$\$$	reduce mittels $\texttt{T} \rightarrow \texttt{T} * \texttt{F}$
$\$\ \underline{\texttt{E + T}}$	$\$$	reduce mittels $\texttt{E} \rightarrow \texttt{E + T}$
$\$\ \texttt{E}$	$\$$	accept (E ist Startsymbol)

Beispiel 4.25: Eine shift-reduce-Analyse des Wortes `id + id * id $` mit einer linksrekursiven Grammatik (vgl. Bsp. 4.13)

aber keine Reduktionsansätze sind (weil sie keinen Schritt in der Rechtsableitung darstellen). Im Bsp. 4.25 ist in der Satzform `E + T * id` nicht `E + T` der Ansatz, sondern `id`.

Eine Folge von Anfangssymbolen einer Satzform, die rechts nicht über den Ansatz hinausgeht, heißt **zulässiges Präfix** (*viable prefix*). D.h. sie steht zur Gänze im Keller (`E` oder `E + id` oder `E + T *` in Bsp. 4.25). Man kann jedes zulässige Präfix um Terminalsymbole ergänzen, sodaß eine (vollständige) Satzform entsteht. Das bedeutet, daß bei der Analyse kein Fehler entdeckt wird, solange man den Anfang der Eingabefolge zu einem zulässigen Präfix reduzieren kann.

Alle Sprachen, die *deterministisch* analysierbar sind, lassen sich durch **LR-Grammatiken** (Analyse von **L**inks nach rechts mittels **R**echtsableitungen (=Linksreduktionen)) beschreiben. Diese Grammatik-Klasse hat zudem die nützliche Eigenschaft, daß die LR-Grammatiken mit Vorausschau 1 genauso mächtig sind wie solche mit mehr Vorausschau. Das heißt, jede Sprache, die mit einer LR(k)-Grammatik (k beliebig) beschrieben werden kann, kann auch mit einer LR(1)-Grammatik beschrieben werden. LR(1)-Grammatiken sind bei der Analyse weitaus weniger aufwendig.

Bei einer LR(k)-Grammatik kann für jedes Eingabewort bei jedem Analyseschritt der Reduktionsansatz und damit die zu setzende Aktion aufgrund des Kellerinhalts und der nächsten k Eingabesymbole bestimmt werden. Weitere Informationen werden nicht benötigt. Hier sieht man (informell) den Grund für die größere Mächtigkeit der LR-Grammatiken: während bei LL(k)-Grammatiken die richtige Produktion allein mit ihren ersten k Terminalsymbolen bestimmt werden muß, stehen bei einer LR(k)-Grammatik die *ganze* rechte Seite der Produktion (im Keller) und noch weitere k Symbole zur Verfügung.

Weitere Vorteile der LR-Grammatiken sind, daß (mit ähnlichem Aufwand wie bei LL-Grammatiken) mächtigere Sprachen analysiert werden können, und daß die Erzeugung von tabellengesteuerten Parsern relativ einfach automatisiert werden kann.

LR-Analyse

Ein Kontroll-Algorithmus, der für alle LR-Parser gleich ist, verwendet Tabellen zur Durchführung der Analyse (Abb. 4.26). Die Tabellen bestimmen die zu analysierende Sprache und das genaue verwendete Analyseverfahren (LR, SLR, LALR, ...). Wir werden uns hier auf die Beschreibung des SLR-Verfahrens (*simple*-LR) beschränken.

Bei der Bottom-Up-Analyse wird generell im Keller das bisher analysierte Teilwort durch Zustände beschrieben.

Die Analysetabellen werden durch Zustände (Zeilen) und Grammatiksymbole inkl. `$` (Spalten) indiziert. Sie beinhalten Aktionen (action-Tabelle) bzw. Folgezustände (goto-Tabelle). Die Aktionen sind:

- `shift` (Eingabesymbol in den Keller schieben)
- `reduce p` (reduzieren; `p` ist die Nummer der Produktion)
- `accept` (richtige Beendigung der Analyse)
- `error` (Fehlermeldung und -behandlung)

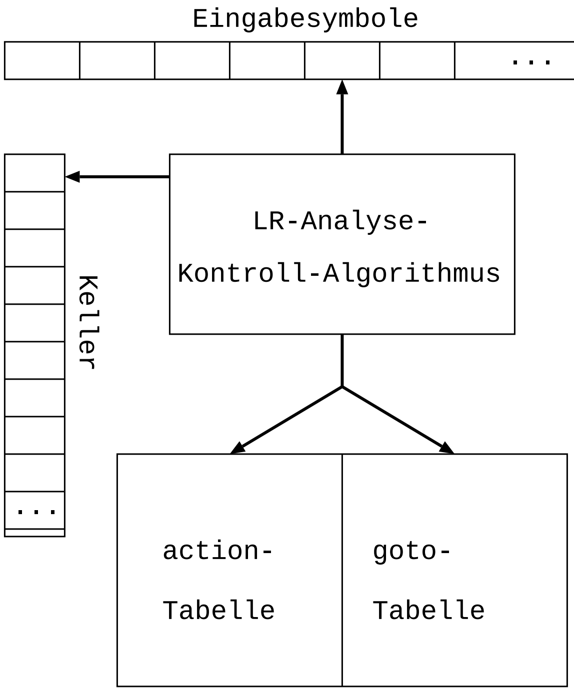


Abbildung 4.26: LR-Analyse-Modell

Generierung von Tabellen. Die Generierung der SLR-Tabellen wird anhand eines Beispiels vorgeführt. Zuerst wird die Grammatik durch Numerierung der Produktionen und Erweiterung um ein neues Startsymbol `S` vorbereitet (siehe Bsp. 4.27).

```
(0) S → E   (1) E → E op T   (3) T → ( E )
            (2) E → T        (4) T → id
```

Beispiel 4.27: Erweiterte Grammatik für einfache Ausdrücke

Generierung der goto-Tabelle. Jede Zeile der goto-Tabelle entspricht einem Zustand, repräsentiert durch eine Item-Menge. Ein **Item** ist eine Produktion, deren rechte Seite um eine Marke (`"."`) erweitert ist: z.B.

`T → .id` oder `E → E op.T`

Die Marke trennt Keller und Eingabefolge, d.h. die bereits analysierten und noch zu analysierenden Symbole.

Enthält die Item-Menge `I` eines Zustands ein Item der Form `N` $\rightarrow \alpha$`.A` β (Marke befindet sich unmittelbar links von einem Nonterminal), so sind auch alle Items der Form `A → .γ` in `I` enthalten (z.B. sind mit `E → E op.T` auch `T → .(T)` und `T → .id` enthalten). Das entspricht der Tatsache, daß zuerst eine der rechten Seiten von `A` abgeleitet werden muß, bevor `A` reduziert werden kann. Die Funktion `closure(I)` erweitert die Menge `I` zur vollständigen Item-Menge.

Ein Zustandsübergang findet statt, wenn die Marke über das Grammatiksymbol nach rechts geschoben wird: das entspricht der gerade erfolgten Reduktion eines Nonterminals bzw. dem Verbrauchen eines Terminalsymbols. Die Funktion `goto(I,G)` liefert die `closure`-Menge jener Menge von Items, die von `I` aus über das Grammatiksymbol `G` erreichbar sind. Beschreibt `I` den zulässigen Präfix α, so beschreibt `goto(I,G)` den zulässigen Präfix α `G` (siehe Alg. 4.28 und Bsp. 4.29).

```
I0 = closure({S → . Startsymbol }); M = {I0}
do {
  for( jede Item-Menge I in M und jedes Grammatiksymbol G ) {
    if( goto(I,G) != {} && goto(I,G) ∉ M )
      füge die Item-Menge goto(I,G) als In mit neuem Index n zu M hinzu
    if( goto(I,G)==Ik && I==Ij ) goto[j][G] = k;
  }
} while( neue Mengen wurden zu M hinzugefügt );
```

Algorithmus 4.28: Generierung der goto-Tabelle.

Generierung der action-Tabelle. Für die action-Tabelle werden die `First`- und `Follow`-Mengen der Grammatiksymbole und die goto-Tabelle benötigt. Das Verfahren ist in Alg. 4.30 zusammengefaßt. Bsp. 4.31 zeigt die Anwendung.

LR-Analyse-Algorithmus. Bei jedem Schritt wird in Abhängigkeit vom aktuellen Eingabesymbol a_i und vom obersten Kellerelement s_m eine Aktion $\texttt{action[}s_m\texttt{,}a_i\texttt{]}$ aus der action-Tabelle durchgeführt (siehe Alg. 4.32 und Bsp. 4.33).

Mehrdeutigkeiten

Bei bestimmten Sprachkonstrukten (Parameterlisten, etc.) ist die Struktur des Ableitungsbaumes unwichtig, sofern die Reihenfolge der Blätter gewahrt bleibt. Zu ihrer Beschreibung könnte man mehrdeutige Grammatiken verwenden und dem Analyse-Verfahren überlassen, welcher Baum generiert wird (siehe Abb. 4.5) .

Oft lassen sich auch andere Konstrukte (z.B. Ausdrücke) einfach mit mehrdeutigen Grammatiken beschreiben. Man braucht dann aber Zusatzregeln, mit denen die Analyse eindeutig wird. Ergebnis der Syntax-Analyse soll in jedem Fall jener (einzige) Ableitungsbaum sein, der der gewünschten Struktur entspricht. Die mehrdeutige Grammatik

$$\texttt{E} \rightarrow \texttt{E + E} \mid \texttt{E * E} \mid \texttt{(E)} \mid \texttt{id}$$

$\mathtt{I}_0 = \{\ \mathtt{S} \to \mathtt{.E}$	Der Anfangszustand 0 enthält $\mathtt{S} \to \mathtt{.E}$,
$\quad\ \mathtt{E} \to \mathtt{.E\ op\ T}$	davon die `closure`-Menge $\mathtt{I}_0$ bilden.
$\quad\ \mathtt{E} \to \mathtt{.T}$	
$\quad\ \mathtt{T} \to \mathtt{.(\ E\ )}$	
$\quad\ \mathtt{T} \to \mathtt{.id}\ \}$	
$\mathtt{I}_1 = \{\ \mathtt{S} \to \mathtt{E.}$	$\mathtt{I}_1$ = `goto(`$\mathtt{I}_0$`,E)`. Rechts von Marken keine
$\quad\ \mathtt{E} \to \mathtt{E.op\ T}\ \}$	Nonterminale $\to$ keine `closure`-Operation.
$\mathtt{I}_2 = \{\ \mathtt{E} \to \mathtt{T.}\ \}$	$\mathtt{I}_2$ = `goto(`$\mathtt{I}_0$`,T)`. Zustand 2 enthält nur fertige Items (die Marke steht ganz rechts) – hier muß reduziert werden.
$\mathtt{I}_3 = \{\ \mathtt{T} \to \mathtt{(.E\ )}$	$\mathtt{I}_3$ = `goto(`$\mathtt{I}_0$`,()`. Darin sind die Items von `E`
$\quad\ \mathtt{E} \to \mathtt{.E\ op\ T}$	enthalten, weil `E` im 1.Item unmittelbar nach
$\quad\ \mathtt{E} \to \mathtt{.T}$	einer Marke steht. Im Zustand 3 werden alle
$\quad\ \mathtt{T} \to \mathtt{.(\ E\ )}$	zulässigen Präfixe erkannt, die mit `(` enden,
$\quad\ \mathtt{T} \to \mathtt{.id}\ \}$	z.B. (, ((, `E op` (, ...
$\mathtt{I}_4 = \{\ \mathtt{T} \to \mathtt{id.}\ \}$	$\mathtt{I}_4$= `goto(`$\mathtt{I}_0$`,id)`. Jetzt sind alle Nachfolger des Zustands 0 bestimmt.
$\mathtt{I}_5 = \{\ \mathtt{E} \to \mathtt{E\ op.T}$	$\mathtt{I}_5$ = `goto(`$\mathtt{I}_1$`,op)`.
$\quad\ \mathtt{T} \to \mathtt{.(\ E\ )}$	
$\quad\ \mathtt{T} \to \mathtt{.id}\ \}$	

Die Generierung wird mit den anderen Elementen von `M(`$\mathtt{I}_2$`,`$\mathtt{I}_3$`,...)` fortgesetzt. Interessant sind `goto(`$\mathtt{I}_3$`,T)=`$\mathtt{I}_2$ und `goto(`$\mathtt{I}_3$`,()=`$\mathtt{I}_3$. Hier werden keine neuen Item-Mengen aufgebaut, sondern die bereits existierenden verwendet.
Gleichzeitig mit der Generierung der Item-Mengen wird die `goto`-Tabelle Zeile für Zeile angelegt. Die Zustände werden dabei wie die Indizes der Item-Mengen numeriert ($\mathtt{I}_0$ – Zustand 0, $\mathtt{I}_1$ – Zustand 1,...).

Beispiel 4.29: Generierung der goto-Tabelle zu Bsp. 4.27

`First`- und `Follow`-Mengen der Grammatiksymbole bestimmen
∀ Zustände `s` und alle Terminalsymbole `a` (inkl. `$`):
  trage in `action[s][a]` (`a` ist dabei das nächste Eingabesymbol) ein:
    `shift`: wenn `goto(`I_s`,a)=`I_j
    `reduce p`: wenn I_s das Item `N` $\rightarrow$ α`.` enthält und
      `a` $\in$ `Follow(N)` und
      die Produktion `N` $\rightarrow$ α die Nummer `p` hat
    `accept`: anstelle von `reduce 0`
    `error`: sonst

Algorithmus 4.30: Generierung der action-Tabelle.

```
First(S)={id,(}, First(E)={id,(}, First(T)={id,(}
Follow(S)={$}, Follow(E)={$,op,)}, Follow(T)={$,op,)}
```

`action[0][id]` = `shift`, da `goto(`I_0`,id) = `I_4
`action[2][op]` = `reduce 2`, da Item `E` $\rightarrow$ `T.` $\in$ I_2 und
  `op` $\in$ `Follow(E)` und
  `E` $\rightarrow$ `T` die Produktion Nr.`(2)` ist.
`action[1][$]` = `accept`, da `$` $\in$ `Follow(S)` und
  Item `S` $\rightarrow$ `E.` $\in$ I_1
`action[0][op]` = `error` (zur Kontrolle: `op` $\notin$ `First(S)`)

vollständige Tabellen:

	action					goto					
state	id	op	(	)	$	id	op	(	)	E	T
0	s	.	s	.	.	4	.	3	.	1	2
1	.	s	.	.	acc	.	5	.	.	.	.
2	.	r2	.	r2	r2	.	.	.	.	.	.
3	s	.	s	.	.	4	.	3	.	6	2
4	.	r4	.	r4	r4	.	.	.	.	.	.
5	s	.	s	.	.	4	.	3	.	.	7
6	.	s	.	s	.	.	5	.	8	.	.
7	.	r1	.	r1	r1	.	.	.	.	.	.
8	.	r3	.	r3	r3	.	.	.	.	.	.

Beispiel 4.31: Generierung der action-Tabelle (zu Bsp. 4.29)

```
for(;;)
  if( action[s_m][a_i] == shift ) {
    verbrauche das Eingabesymbol a_i
    schreibe den Inhalt von goto[s_m][a_i] in den Keller
  } else if( action[s_m][a_i] == reduce p ) {
    lösche soviel Elemente vom Keller, wie die rechte Seite von p Symbole hat
    sei s das neue oberste Kellersymbol und A die linke Seite von p,
    so schreibe den Inhalt von goto[s][A] auf den Keller
  } else if( action[s_m][a_i] == accept ) break;
  else Error
```

Algorithmus 4.32: LR-Analyse-Algorithmus.

Keller	verbleibende Eingabe	anzuwendende Aktion
0	id op id op id $	s (* shift, push goto[0,id] = 4 *)
0 4	op id op id $	r4 (* T $\rightarrow$ id, 1x pop, push goto[0,T] *)
0 2	op id op id $	r2 (* E $\rightarrow$ T *)
0 1	op id op id $	s
0 1 5	id op id $	s
0 1 5 4	op id $	r4 (* T $\rightarrow$ id *)
0 1 5 7	op id $	r1 (* E $\rightarrow$ E op T, 3x pop, push goto[0,E] *)
0 1	op id $	s
0 1 5	id $	s
0 1 5 4	$	r4 (* T $\rightarrow$ id *)
0 1 5 7	$	r1 (* E $\rightarrow$ E op T *)
0 1	$	acc

Beispiel 4.33: Analyse von `id op id op id $` (vgl. Bsp. 4.25).

kann z.B. mit den Zusatzregeln `left +, left *` Eingabewörter eindeutig analysieren. Die Zusatzregeln bedeuten, daß die Operatoren `+` und `*` linksassoziativ sind, und (aufgrund der Reihenfolge) daß `*` eine höhere Priorität als `+` besitzt. Auf diese Weise kann der Ausdruck `a+b*c` richtig als `a+(b*c)` analysiert werden.

Bei LR-Methoden entstehen durch mehrdeutige Grammatiken Mehrfach-Einträge in den action-Tabellen. Bei reduce/reduce-Konflikten ist nicht klar, mit welcher Produktion zu reduzieren ist; bei shift/reduce-Konflikten kann entweder sofort reduziert oder aber ein Eingabesymbol verbraucht werden.

Streichen der unerwünschten Einträge löst die Konflikte und macht die Analyse eindeutig (siehe Bsp. 4.34). Der Parser-Generator YACC erlaubt das automatische Streichen von Einträgen durch die Angabe von Zusatzregeln und durch Default-Annahmen. Bei shift/reduce-Konflikten streicht er den `reduce`-Eintrag (und wählt damit die 'längere' Variante); bei reduce/reduce-Konflikten bleibt der Eintrag für die früher definierte Produktion stehen.

So lassen sich auch *Spezialfälle* in Programmiersprachen einfach durch Streichen von mehrdeutigen Tabelleneinträgen behandeln, wenn der Sonderfall vor der allgemeinen Regel steht. Eine Grammatik zur Beschreibung mathematischer Formeln enthält z.B. die Produktionen

$$\texttt{F} \rightarrow \texttt{F < F} \mid \texttt{F = F} \mid \ldots \mid \texttt{F + F} \mid \ldots \mid \texttt{id} \mid \texttt{num}.$$

Um auch Variablen-Eingrenzungen darstellen zu können, wird als (längerer) Spezialfall als erste Alternative `F < id < F` eingefügt. Die entstehenden Konflikte in den Tabellen werden automatisch richtig gelöst.

4.4 Hierarchie der Analyse-Verfahren

Das in Kapitel 4.3 vorgestellte Verfahren heißt **SLR(1)**: Simple-LR mit Vorausschau 1. Es kann zu Doppeleinträgen in der action-Tabelle führen, weil der SLR-Algorithmus zuwenig Information über die Geschichte der Ableitung mitführt. Die Verwendung der `Follow`-Mengen ist zu allgemein, da `Follow(N)` alle Terminale enthält, die dem Nonterminal `N` folgen können. Diese Mengen sind unabhängig vom momentanen Zustand. Oft können aber einem Nonterminal N *während einer bestimmten Ableitung* (d.h. in einem bestimmten Zustand) nur einige der `Follow`-Symbole *tatsächlich* folgen.

Beim Generieren von **LR(1)**-Tabellen werden daher Items eingesetzt, die zusätzlich die ihnen tatsächlich folgenden `Follow`-Symbole (look-ahead-Mengen) beinhalten. Die Zustandsmenge des Automaten wird dadurch viel größer (viele Zustände sind in solche mit gleichen Items, aber verschiedenen look-ahead-Mengen aufgespalten); der Speicheraufwand, aber auch die Mächtigkeit steigt.

Der hohe Speicherbedarf von LR(1)-Parsern wird vermieden, wenn man Zustände mit gleichen Items zusammenfaßt und dabei die Vereinigung ihrer look-ahead- Mengen bildet. Das führt zu Look-Ahead-LR-Analysatoren, deren Zustandszahlen der von SLR-Parsern entsprechen. Durch das Zusammenfügen der Zustände können

Aus der mehrdeutigen Grammatik für Ausdrücke

```
(1) E → E + E     (3) E → ( E )
(2) E → E * E     (4) E → id
```

werden die folgenden SLR-Tabellen erstellt:

		action						goto
handle	state	id	+	*	(	)	$	E
ε	0	s3	.	.	s2	.	.	1
E	1	.	s4	s5	.	.	acc	.
(	2	s3	.	.	s2	.	.	6
id	3	.	r4	r4	.	r4	r4	.
E +	4	s3	.	.	s2	.	.	7
E *	5	s3	.	.	s2	.	.	8
(E	6	.	s4	s5	.	s9	.	.
E + E	7	.	s4/r1	s5/r1	.	r1	r1	.
E * E	8	.	s4/r2	s5/r2	.	r2	r2	.
(E)	9	.	r3	r3	.	r3	r3	.

In den Zuständen 7 und 8 kommt es zu shift/reduce-Konflikten. Beide Operatoren sollen linksassoziativ sein – in `action[7,+]` und `action[8,*]` muß also reduziert werden (`s4` bzw. `s5` werden gestrichen). Wird im Zustand 7 das Symbol `*` gelesen, d.h. wurde `E + E *` erkannt (siehe Spalte handle), so ist `shift` die richtige Aktion, da `*` die höhere Priorität hat und zuerst ausgeführt werden muß. Aus demselben Grund muß `action[8,+] = r2` gelten (das entspricht … `E * E +` …).
Für eine eindeutige (und strukturell zutreffende) Analyse haben die Zeilen 7 und 8 also das Aussehen:

		action						goto
handle	state	id	+	*	(	)	$	E
E + E	7	.	r1	s5	.	r1	r1	.
E * E	8	.	r2	r2	.	r2	r2	.

Die neue Tabelle ist auch mit LALR(1)- oder LR(1)-Verfahren nicht direkt (d.h. ohne Zusatzregeln) zu gewinnen.

Beispiel 4.34: Eindeutige Analyse durch das Streichen der unerwünschten Einträge aus der Tabelle.

reduce/reduce-, nicht jedoch shift/reduce-Konflikte entstehen. Bei richtigen Eingabeworten verhält sich der Parser wie der ursprüngliche LR-Parser, arbeitet aber wesentlich effizienter. Außerdem gibt es effiziente Verfahren, mit denen man **LALR(1)**-Tabellen direkt (also nicht über den Umweg der LR(1)-Zustände) aus einer Grammatik generieren kann.

Die Verfahren sind verschieden mächtig in Bezug auf die Grammatiken, die sie verarbeiten können.

$$\mathrm{LR}(1) \supset \mathrm{LALR}(1) \supset \mathrm{SLR}(1)$$

Die weniger mächtigen Verfahren erzeugen u.U. Doppeleinträge beim Aufbau der Tabellen (wie z.B. SLR für Bsp. 4.19), die mächtigere Verfahren noch eindeutig behandeln können. Bei mehrdeutigen Grammatiken entstehen bei jedem LR-Verfahren Mehrfach-Einträge in den action-Tabellen.

$$\text{kontextfrei (auch mehrdeutig)} \supset \mathrm{LR}$$

LR(1)-Parser können alle LL(1)-Grammatiken behandeln. Das gilt nicht für LALR und SLR, bei denen manche LL-Grammatiken mit ε-Produktionen zu Doppeleinträgen führen.

$$\mathrm{LR}(1) \supset \mathrm{LL}(1)$$

4.5 Wie schreibt man Grammatiken?

Das Erstellen von Grammatiken ähnelt sehr dem Schreiben von Programmen. Die zugrunde liegende Aufgabe läßt sich sehr oft auf mehrere Arten lösen. Man sollte versuchen, durch Aufspalten des Problems in kleine, überschaubare Module die Wiederverwendung von bewährten Teillösungen zu fördern.

Das Aufstellen einer Grammatik geht in drei Schritten vor sich:

1. Identifizieren von *syntaktischen Kategorien*, d.h. von wichtigen Bestandteilen der zu beschreibenden Sprache. Diesen Kategorien werden Nonterminale zugeordnet.

2. Suchen von Beziehungen zwischen syntaktischen Kategorien. Diese Beziehungen werden dann mit Grammatikregeln beschrieben. Eine Grammatikregel zeigt, wie sich die syntaktische Kategorie auf der linken Seite aus den syntaktischen Kategorien der rechten Seite zusammensetzt. Verschiedene Regeln mit gleicher linker Seite entsprechen verschiedenen Spezialisierungen der syntaktischen Kategorie der linken Seite.

3. Aus der *abstrakten Syntax* (das Ergebnis der ersten beiden Schritte) wird nun die *konkrete Syntax* festgelegt. Das betrifft vor allem die Reihenfolge der Symbole auf der rechten Seite einer Produktionsregel und eventuell zusätzliche Terminalsymbole, um die Struktur zu betonen und die Lesbarkeit zu verbessern.

Als Beispiel wollen wir eine Grammatik aufstellen, die Zahlen in verschiedenen Zahlensystemen beschreiben soll.

1. Schritt: Als Kandidaten für syntaktische Kategorien fallen die Basis und die Ziffernfolge ins Auge. Eine Ziffernfolge besteht naturgemäß aus Ziffern bzw. Einzelbuchstaben. Wir führen als Bezeichnungen die Nonterminale `B`, `F` und `Z` ein. Den ganzen Ausdruck bezeichnen wir mit `A`. Der Aufbau der Basis soll im Gegensatz zu den Ziffern der Ziffernfolge in der Grammatik nicht erklärt werden sondern als Einheit mit einem terminalen Symbol `num` betrachtet werden. Ihr Wertebereich liegt zwischen 2 und 36.

2. Schritt: Die Beziehungen zwischen den syntaktischen Kategorien sind leicht zu erstellen:

1. Ein Ausdruck `A` besteht aus einer Ziffernfolge `F` und der Basis `B`.
2. Eine Ziffernfolge `F` ist eine Aneinanderreihung von Ziffern `Z`.
3. Eine Ziffer ist eines der Zeichen '0' bis '9' oder einer der Buchstaben 'A' bis 'Z' (für Zahlensysteme bis zur Basis 36).
4. Eine Basis `B` ist eine Zahl `num`.

3. Schritt: Wir setzen die in den vorigen Schritten aufgestellte abstrakte Syntax aus Symbolen und Beziehungen nacheinander in konkrete Grammatikregeln um. Dabei wird die konkrete Syntax der Sprache festgelegt.

In unserem Beispiel muß die Ziffernfolge irgendwie von der Basis getrennt werden, damit wir diese beiden Gruppen von Zeichen auseinander halten können. Als Trennzeichen wählen wir das Zeichen '/'. Danach steht die Basis des Zahlensystems, z.B. A6E2/16, 10011/2, 755/8. Aus der ersten Beziehung ergibt sich die Regel

$$\texttt{A} \;\rightarrow\; \texttt{F '/' B}$$

Die Ziffernfolge aus der zweiten Beziehung wird z.B. durch

$$\begin{array}{lcl} \texttt{F} & \rightarrow & \texttt{F Z} \\ \texttt{F} & \rightarrow & \texttt{Z} \end{array}$$

dargestellt.

Eine einfache Umsetzung der dritten Beziehung nimmt keine Rücksicht auf Klassen von Zeichen sondern listet ganz einfach alle Alternativen auf.

$$\texttt{Z} \;\rightarrow\; \texttt{'0'} \mid \texttt{'1'} \mid \ldots \mid \texttt{'9'} \mid \texttt{'A'} \mid \texttt{'B'} \mid \ldots \mid \texttt{'Z'}$$

Die letzte Beziehung ergibt sich zu

$$\texttt{B} \;\rightarrow\; \texttt{num}$$

Die vollständige Grammatik hat damit die Form:

```
A  →  F '/' B
F  →  F Z
F  →  Z
Z  →  '0' | '1' |...| '9' | 'A' | 'B' |...| 'Z'
B  →  num
```

4.6 Qualitätskriterien für Grammatiken

Es gibt keine eindeutigen Kriterien für die Qualität von Grammatiken, wohl aber nützliche Stilregeln. Einige solcher Stilregeln sind:

- *Die Grammatik muß aufgabenmäßig abgefaßt sein.* Die syntaktischen Kategorien, die für die Verarbeitung von Bedeutung sind, müssen in der Grammatik festgelegt sein. Sind sie das nicht, dann läßt sich die Grammatik nur schwer attributieren. Die Beschreibung arithmetischer Ausdrücke hat z.B. als Forderung die unterschiedliche Priorität von Operatoren (Punktrechnung vor Strichrechnung). Sind diese Unterscheidungen schon in der Grammatik repräsentiert, dann vereinfacht sich die Attributierung ganz erheblich.

- *Die Grammatik muß analysegeeignet sein.* Dieser Punkt trifft besonders auf Grammatiken zu, die später mit Compilergeneratoren bearbeitet werden sollen. Für manche Syntaxanalyseverfahren sind z.B. Grammatiken mit linksrekursiven Symbolen nicht geeignet.

- *Grammatiken sollen eindeutig sein.* Die Eindeutigkeit erleichtert sowohl die Syntaxanalyse als auch die Zuordnung von Bedeutung durch die Attributierung.

Kapitel 5

Syntaxgesteuerte Übersetzung

In diesem Kapitel werden Methoden behandelt, die es erlauben, auf der Basis einer kontextfreien Grammatik nicht nur die Syntax-Analyse, sondern beliebige Übersetzungen zu definieren und zu implementieren. Die Methoden beruhen auf der **Attributierten Grammatik**. In einer Attributierten Grammatik werden den Grammatiksymbolen Attribute zugeordnet, und Übersetzungen werden als Berechnungen der Attribute beschrieben. Da die Berechnungen an die Grammatik gebunden sind, handelt es sich um eine **Syntaxgesteuerte Übersetzung**.

In ihrer allgemeinen Form kann man eine Attributierte Grammatik als „deklarative" Spezifikation eines Compilers ansehen. Der Ablauf der Übersetzung wird nicht explizit, sondern nur implizit durch die Abhängigkeiten der Attribute definiert. Compiler-Generatoren können aufgrund der Attributabhängigkeiten den Ablauf der Übersetzung ermitteln und einen Compiler aus der Spezifikation generieren.

Unter bestimmten Voraussetzungen kann man eine Attributierte Grammatik auch als **Übersetzungsschema** formulieren. Dabei wird der Ablauf der Übersetzung explizit festgelegt. Ein Übersetzungsschema ist eine „prozedurale" Spezifikation eines Compilers und als Eingabe für effiziente Compiler-Generatoren besonders geeignet. Wir werden zwei typische Vertreter solcher Generatoren behandeln.

5.1 Attributierte Grammatik

Eine **Attributierte Grammatik** (AG) ist eine kontextfreie Grammatik mit folgenden Erweiterungen:

- Jedem Grammatiksymbol können **Attribute** zugeordnet werden. Die Attribute sind in zwei Klassen eingeteilt, in **synthetisierte** (*synthesized*) und **ererbte** (*inherited*) Attribute.

- Jeder Produktion A $\rightarrow$ ... X ... können **Regeln** zugeordnet werden. Jede Regel berechnet ein Attribut und hat die allgemeine Form

$$\mathrm{b} = \mathrm{f}(\mathrm{c}_1,\ldots,\mathrm{c}_k)$$

b ist ein **synthetisiertes** Attribut des Symbols A der **linken** Seite oder ein **ererbtes** Attribut eines Symbols X der **rechten** Seite.

$c_1,\ldots,c_k$ sind **ererbte** Attribute des Symbols der **linken** Seite oder **synthetisierte** Attribute der Symbole der **rechten** Seite.

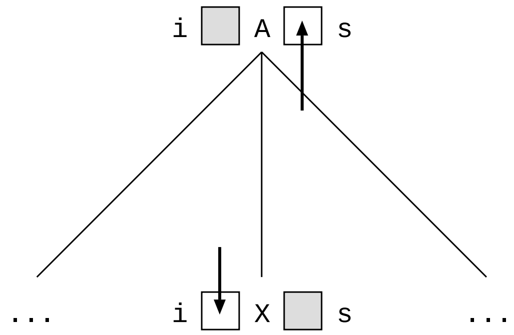


Abbildung 5.1: Attributberechnungen einer Produktion

Wir betrachten die Produktion `p: A` $\rightarrow$ `... X ...` im Ableitungsbaum (Abb. 5.1), wobei angenommen ist, daß `A` und `X` jeweils ein ererbtes Attribut `i` und ein synthetisiertes Attribut `s` haben. Die Pfeile zeigen, welche Attribute durch Regeln der Produktion p berechnet werden. Die anderen Attribute (schraffiert) werden durch Regeln der angrenzenden Produktionen berechnet. Synthetisierte Attribute geben also entlang des Ableitungsbaums Informationen von unten nach oben weiter, ererbte Attribute von oben nach unten.

Wir verwenden im folgenden die Record-Schreibweise `Y.a`, um ein Attribut `a` des Symbols `Y` zu bezeichnen. Dabei wählen wir im allgemeinen den Attributnamen `s` für synthetisierte, `i` für ererbte und `x` für Attribute von Terminalen.

5.2 Wie attributiert man Grammatiken?

Für die Attributierung geht man von der kontextfreien Grammatik aus und geht wie folgt vor:

1. Identifizieren von *semantischen Kategorien*, d.h. von Eigenschaften bzw. Werten, die für die beabsichtigte Übersetzung bzw. Auswertung eine Rolle spielen. Zur Benennung dieser semantischen Kategorien werden Attribute eingeführt. Diese sind entweder synthetisierte oder ererbte Attribute, je nachdem, ob die betrachtete Information im Syntaxbaum von unten nach oben oder in umgekehrter Richtung fließen soll.

   Dieser erste Schritt beeinflußt die entstehende Attributierung am nachhaltigsten. Man muß sich an dieser Stelle genau überlegen, welche Informationen man benötigt und in welcher Weise diese repräsentiert werden sollen.

2. *Zuordnung* von semantischen zu syntaktischen Kategorien, d.h. die Zuordnung von den Attributen zu den Nonterminalen. Es kann vorkommen, daß Nonterminale als „Zwischenlager“ für bestimmte Attribute eingesetzt werden müssen, die dort nicht benötigt werden. Nur so ist es möglich, Informationen weiter nach oben oder unten durchzureichen.

3. Bestimmen der *Attributauswertungsregeln:* Für jedes synthetisierte Attribut eines Nonterminals der linken Seite einer Regel und für jedes ererbte Attribut eines Nonterminals der rechten Seite einer Regel muß es eine Auswertungsregel geben.

Bei der Formulierung der Auswertungsregeln kann es notwendig sein, weitere Attribute einzuführen oder bestehende Attribute weiteren Nonterminalen zuzuordnen. Dies führt dann zu Iteration der Schritte 1 bis 3.

Synthetisierte und ererbte Attribute

Betrachten wir noch einmal unser Beispiel aus Kapitel 4.5. Wir wollen diese Grammatik attributieren, um den Wert einer Zahl zu berechnen.

1. Schritt: Als Ergebnis der Attributauswertung erwarten wir den Dezimalwert der Zahl in einem Attribut `val`. Bei der Betrachtung der Grammatik sehen wir, daß dieses Attribut Informationen in Richtung des Startsymbols zurückliefert, es wird also synthetisiert. Für die Basis brauchen wir auch ein Attribut `base`. Die Information wird jedoch bei der Berechnung des Wertes einer Ziffer gebraucht und muß daher als ererbtes Attribut zur entsprechenden Regel übergeben werden.

2. Schritt: Nun erfolgt die Zuordnung der semantischen zu den syntaktischen Kategorien. Alle Nonterminale erhalten das synthetisierte Attribut `val`. Die Ziffernfolge `F` erbt zusätzlich noch das Attribut `base`.

3. Schritt: Als letzten Schritt müssen wir alle konkreten Attributauswertungsregeln angeben. Dabei ist zu beachten, daß nur Attribute verwendet werden, die im jeweiligen Kontext auch definiert sind.

Wir beginnen mit den Definitionen der Regeln von unten, d.h. bei den Blättern. Die Reihenfolge ist rein willkürlich gewählt. Sie folgt nur dem Fluß der synthetisierten Attribute, die in unserem Beispiel häufiger auftreten.

Die Attribute werden nicht explizit deklariert, sondern implizit durch die Regeln eingeführt. Die Attributnamen sind frei wählbar und lokal für ein Grammatiksymbol gültig. Wenn in einer Produktion ein Grammatiksymbol mehrmals vorkommt (z.B. `F`), so wird zur Unterscheidung indiziert, damit sich die Regeln eindeutig auf die unterschiedlichen Vorkommen des Symbols in der Produktion beziehen können.

Den trivialen Abhängigkeiten der Regeln

```
F  →  Z
Z  →  '0' | '1' |...| '9' | 'A' | 'B' |...| 'Z'
B  →  num
```

entsprechen häufig „Transferregeln“ wie

```
F.val  =  Z.val
Z.val  =  0
Z.val  =  1
          ⋮
Z.val  =  35
B.val  =  value( num.x )
```

die Attributwerte unverändert weiterreichen bzw. bei Terminalsymbolen die Werte erst zur Verfügung stellen[1]. Alle synthetisierten Attribute der linken Seiten haben damit einen Wert erhalten. Auf den rechten Seiten gibt es keine ererbten Attribute, denen ein Wert zugewiesen werden müßte.

Die eigentliche Berechnung des Zahlenwertes geschieht in der Regel

$$\mathtt{F}_1 \rightarrow \mathtt{F}_2\ \mathtt{Z}$$

Der Wert für jede neue Ziffernfolge ergibt sich aus der alten Ziffernfolge multipliziert mit der Basis des Zahlensystems. Zu diesem Wert wird die neue Ziffer addiert. Die Basis steht im ererbten Attribut $\mathtt{F}_1\mathtt{.base}$. Damit und mit den restlichen synthetisierten Attributen der rechten Seite können wir die Attributierungsregel für das einzige synthetisierte Attribut der linken Seite formulieren. Fehlt nur noch die Regel für das einzige ererbte Attribut auf der rechten Seite. $\mathtt{F}_2\mathtt{.base}$ bekommt den Wert von $\mathtt{F}_1\mathtt{.base}$.

Wir erhalten

$$\begin{array}{lcl} \mathtt{F}_1\mathtt{.val} & = & \mathtt{F}_2\mathtt{.val * F}_1\mathtt{.base + Z.val} \\ \mathtt{F}_2\mathtt{.base} & = & \mathtt{F}_1\mathtt{.base} \end{array}$$

In der ersten Produktion müssen wir jetzt noch den Wert der Basis zur Verfügung stellen und die errechnete Zahl an das Startsymbol übergeben. Es ergeben sich die Regeln

```
A.val   =  F.val
F.base  =  B.val
```

Die vollständige attributierte Grammatik hat damit das folgende Aussehen:

[1] Z.B. übersetzt die Funktion `value()` die Textrepräsentation (die im Attribut `x` abgelegt ist) der Zahl des Symbols in einen numerischen Wert.

Produktion			Regeln		
A	→	F '/' B	A.val	=	F.val
			F.base	=	B.val
F_1	→	F_2 Z	F_1.val	=	F_2.val * F_1.base + Z.val
			F_2.base	=	F_1.base
F	→	Z	F.val	=	Z.val
Z	→	'0'	Z.val	=	0
Z	→	'1'	Z.val	=	1
	⋮			⋮	
Z	→	'Z'	Z.val	=	35
B	→	num	B.val	=	value(num.x)

Die durch eine attributierte Grammatik spezifizierte Übersetzung kann man exemplarisch durch einen **attributierten Ableitungsbaum** darstellen. Dabei werden die Attributwerte in die Ableitungsknoten eingetragen. Jede Übersetzung liefert genau eine Wertebelegung der Knoten. Abb. 5.2 zeigt den attributierten Ableitungsbaum für die Umwandlung von `"3F7/16"` in die Dezimalzahl 1015. Zur besseren Übersicht sind nur die synthetisierten Attribute dargestellt.

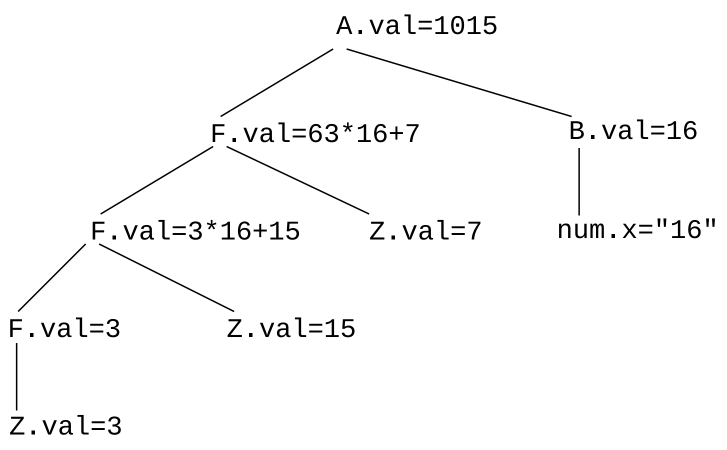


Abbildung 5.2: Attributierter Ableitungsbaum für die Umwandlung von `"3F7/16"` in die Dezimalzahl 1015. Nur synthetisierte Attribute sind dargestellt.

In Abb. 5.3 ist der **Abhängigkeitsgraph** des Ableitungsbaums dargestellt. Die Knoten des Abhängigkeitsgraphen sind als Punkte neben den Ableitungsknoten eingetragen. Sie repräsentieren beliebige Attributwerte. Ererbte Attribute stehen dabei links und synthetisierte Attribute rechts. Der Abhängigkeitsgraph des gesamten Ableitungsbaums wird aus den Abhängigkeitsgraphen der angewandten Produktionen zusammengesetzt.

Abb. 5.3 veranschaulicht den Informationsfluß bei der Ableitung der dreistelligen Hexadezimalzahl `"3F7/16"`. In diesem Beispiel sieht man, wie ererbte Attribute dazu verwendet werden, Informationen „hinunter“ in die Zweige des Baumes zu liefern.

Die Basis der umzuwandelnden Zahl muß ja für die Berechnung des Dezimalwertes bereitgestellt werden.

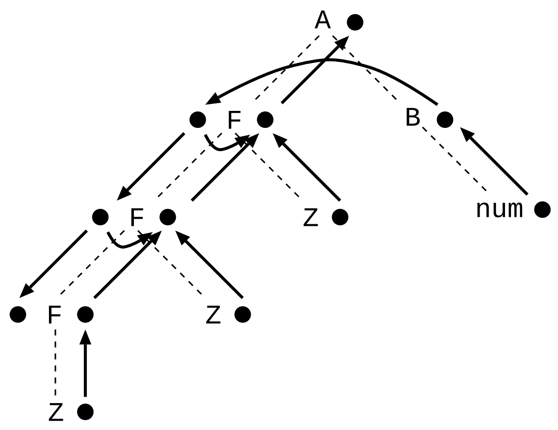


Abbildung 5.3: Abhängigkeitsgraph bei der Berechnung einer dreistelligen Zahl.

5.3 Qualitätskriterien für die Attributierung

Wir ergänzen die Stilregeln für Grammatiken um solche für Attributierungen:

- *Die Attributierung soll einfach und natürlich formuliert sein.* Dies empfiehlt sich wegen des Umfangs der Attributierung, der oft den der zugrundeliegenden Grammatik um ein Vielfaches übertrifft.
- *Die Attributierung muß vollständig sein*, d.h. in jedem Kontext muß für jedes zu berechnende Attribut eine Attributierungsregel zur Verfügung stehen.
- *Die Attributierung muß zyklenfrei sein.* Kein Attribut darf (direkt oder indirekt) von sich selbst abhängen.

5.4 S-Attributierte Grammatik

Eine AG, die nur synthetisierte Attribute enthält, heißt **S-attributiert**.

Die Nonterminale `E` und `T` in Bsp. 5.4 besitzen jeweils ein synthetisiertes Attribut `s`, in dem der Postfix-Code (P-Code) als String „berechnet“ wird.

Die Terminale `op` und `id` besitzen ein Attribut x, dessen Wert von der Lexikalischen Analyse geliefert wird (Tokenattribut). Wir nehmen hier an, daß die Operanden und Operatoren des zu übersetzenden Infix-Ausdrucks aus jeweils nur einem Zeichen bestehen und daß dieses Zeichen als Tokenattribut übergeben wird.

Die den Produktionen zugeordneten Regeln können als „Übersetzungsgleichungen“ gelesen werden:

	Produktion	Regeln
(1)	E $\rightarrow$ E_1 op T	E.s = E_1.s + T.s + op.x
(2)	E $\rightarrow$ T	E.s = T.s
(3)	T $\rightarrow$ (E)	T.s = E.s
(4)	T $\rightarrow$ id	T.s = id.x

Beispiel 5.4: Eine AG zur Übersetzung von Infix- in Postfix-Ausdrücke

1. Der P-Code für `E` ist die Verkettung der P-Codes für E_1 und `T` mit `op.x`.
2. Der P-Code für `E` ist der P-Code für `T`.
3. Der P-Code für `T` ist der P-Code für `E` (Klammern werden eliminiert).
4. Der P-Code für `T` ist `id.x`.

Abb. 5.5 zeigt auf der linken Seite den attributierten Ableitungsbaum für die Übersetzung von `"a+b"` in `"ab+"` . Die eingetragenen Attributwerte sind als Strings aufzufassen.

Rechts in Abb. 5.5 ist der **Abhängigkeitsgraph** des Ableitungsbaums dargestellt. In diesem Beispiel erfolgt die Berechnung aller Attribute im Ableitungsbaum von unten nach oben.

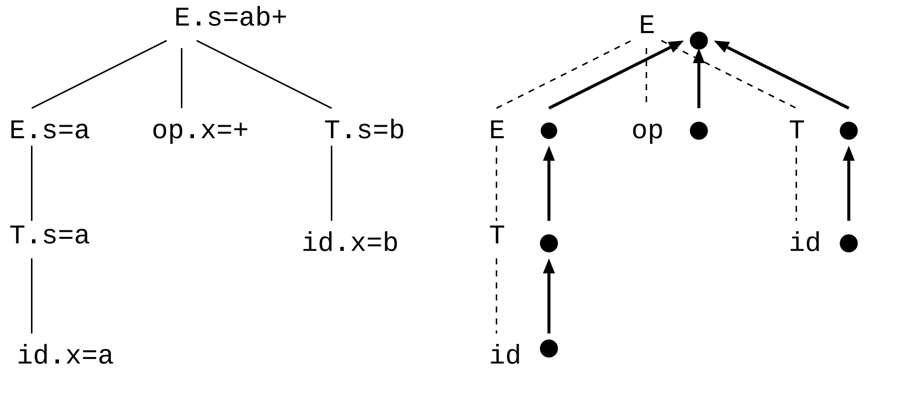


Abbildung 5.5: Attributierter Ableitungsbaum und Abhängigkeitsgraph

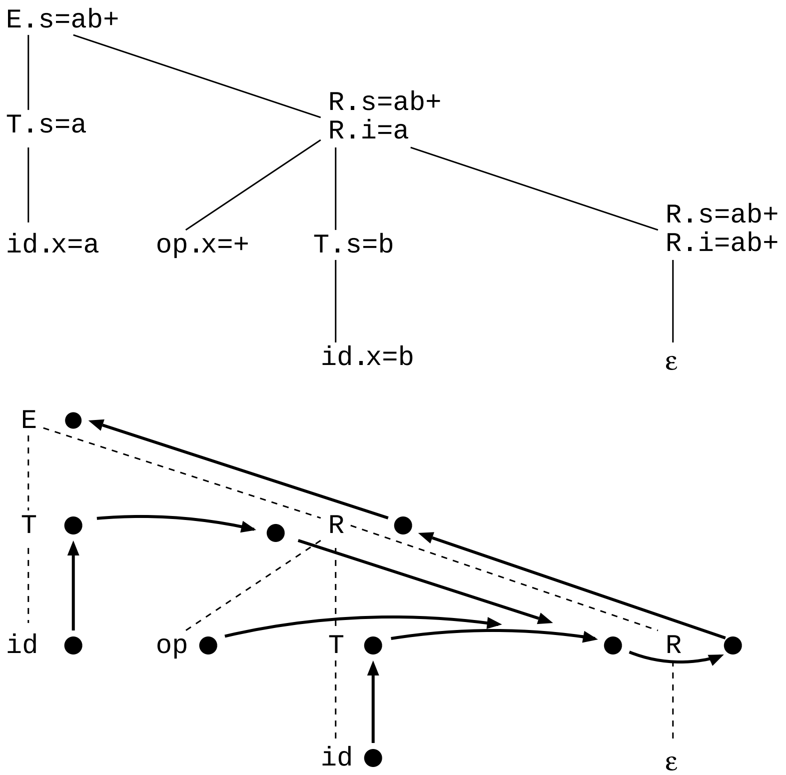
Durch einfache Modifikation kann man die AG von Bsp. 5.4 auch für andere Zwecke verwenden, z.B. würde durch `op.x + `E_1`.s + T.s` ein **Prefix-Code** erzeugt. Wenn die Operanden nicht symbolisch, sondern als Werte gegeben sind, kann man anstelle der Verkettung zum Beispiel eine Funktion `apply(op.x,`E_1`.s,T.s)` aufrufen, die den Operator auf die Operanden anwendet, d.h. man erhält die Spezifikation eines **Interpreters** für Infix-Ausdrücke.

Ererbte Attribute

Grundsätzlich dienen ererbte Attribute dazu, Werte im Ableitungsbaum von oben nach unten zu geben. Sie werden z.B. bei LL-Grammatiken wegen der Ersetzung von Linksrekursion durch Rechtsrekursion benötigt (Bsp. 5.6 und Kapitel 5.5).

Produktion	Regel
E $\rightarrow$ T R	R.i = T.s
	E.s = R.s
R $\rightarrow$ op T R_1	R_1.i = R.i + T.s + op.x
	R.s = R_1.s
R $\rightarrow$ ε	R.s = R.i
T $\rightarrow$ (E)	T.s = E.s
T $\rightarrow$ id	T.s = id.x

Attributierter Ableitungsbaum und Abhängigkeitsgraph (ererbte Attribute stehen links, synthetisierte rechts vom Ableitungsknoten)



Beispiel 5.6: Attributierte LL-Grammatik mit ererbten Attributen zur Übersetzung von Infix in Postfix.

5.5 L-Attributierte Grammatik

Eine AG heißt **L-attributiert** (LAG, genauer: LAG(1)), wenn alle Attribute in einem links-rechts Tiefendurchlauf (*left-right depth-first*) des Ableitungsbaums berechnet werden können, entsprechend der rekursiven Prozedur Alg. 5.7.

```
public void dfvisit(Node n) {
  for( alle direkten Nachfolger m von links nach rechts ) {
    berechne die ererbten Attribute von m
    dfvisit(m);
  }
  berechne die synthetisierten Attribute von n
}
```

Algorithmus 5.7: Berechnung der Attribute in einem links-rechts Tiefendurchlauf des Ableitungsbaums.

Aus der Prozedur `dfvisit` in Alg. 5.7 kann man ersehen, daß eine L-attributierte Grammatik folgende Bedingung erfüllen muß:

> In jeder Produktion `A` $\rightarrow$ ... `X` ... dürfen die ererbten Attribute von `X` nur von Attributen jener Symbole abhängen, die **links** von `X` stehen (d.h. von ererbten von `A` und synthetisierten von anderen Symbolen links von `X`).

Da die einschränkende Bedingung der L-attributierten Grammatik sich nur auf ererbte Attribute bezieht, ist jede S-attributierte Grammatik auch L-attributiert.

Die S-attributierte Grammatik von Bsp. 5.4 ist also L-attributiert und ebenfalls die Grammatik von Bsp. 5.6, da sie die obige Bedingung erfüllt. Man erkennt an den Pfeilen im Abhängigkeitsgraphen von Bsp. 5.6, daß die Attributberechnung in einem Tiefendurchlauf des Ableitungsbaums erfolgen kann.

Bsp. 5.8 zeigt eine nicht L-attributierte Grammatik. Es handelt sich um die Übersetzung Boolescher Ausdrücke, wobei der linke Teilausdruck E_1 Information über die Art des Operators benötigt.

Produktion	Regel
$E \rightarrow E_1$ op T	E_1.i = f(op.x)
...	

Beispiel 5.8: Ausschnitt aus einer nicht-L-attributierten Grammatik

5.6 Übersetzungsschema

Eine L-attributierte Grammatik kann man als **Übersetzungsschema** schreiben. Dabei werden die Regeln jeder Produktion geordnet und so in die rechte Seite der Produktion eingesetzt, daß sie beim Tiefendurchlauf des Ableitungsbaums von links nach rechts ausgeführt werden können. Die eingesetzen Regeln bezeichnet man als (semantische) **Aktionen**.

$$
\begin{array}{l}
\mathtt{E} \rightarrow \mathtt{T}\ \{\mathtt{R.i = T.s}\}\ \mathtt{R}\ \{\mathtt{E.s = R.s}\} \\
\mathtt{R} \rightarrow \mathtt{op\ T}\ \{\mathtt{R_1.i = R.i + T.s + op.x}\}\ \mathtt{R_1}\ \{\mathtt{R.s = R_1.s}\} \\
\mathtt{R} \rightarrow \varepsilon\ \{\mathtt{R.s = R.i}\} \\
\mathtt{T} \rightarrow \mathtt{(\ E\ )}\ \{\mathtt{T.s = E.s}\} \\
\mathtt{T} \rightarrow \mathtt{id}\ \{\mathtt{T.s = id.x}\}
\end{array}
$$

Beispiel 5.9: LL-Übersetzungschema der AG von Bsp. 5.6

In einem Übersetzungsschema muß jede Aktion, die ein ererbtes Attribut eines Symbols berechnet, links dieses Symbols stehen. Die Aktion darf nur Attributwerte von Symbolen verwenden, die links dieser Aktion stehen. Aktionen zur Berechnung des synthetisierten Attributs der linken Seite können generell am Ende der Produktion geschrieben werden.

Die in Bsp. 5.8 gezeigte Produktion kann man nicht als Übersetzungsschema schreiben, da die Aktion $\mathtt{E_1.i = f(op.x)}$ zur Berechnung des ererbten Attributs vor $\mathtt{E_1}$ plaziert werden muß. Die Aktion verwendet aber den Attributwert `op.x` des Symbols `op`, das nicht links dieser Aktion steht.

Bei einer allgemeinen AG (z.B. in Bsp. 5.6) wird nicht spezifiziert, in welcher Reihenfolge die Regeln der Attributberechnung auszuführen sind. Die Reihenfolge ergibt sich nur implizit aus den Abhängigkeiten der Attribute. Es handelt sich also um eine „deklarative“ Spezifikation. Beim Übersetzungsschema wird dagegen die Ausführungsfolge der Regeln in Form von Aktionen explizit angegeben. Es handelt sich hier um eine „prozedurale“ Spezifikation, und es ist entsprechend einfach, aus einem Übersetzungsschema einen Compiler zu generieren.

5.7 Top-Down-Compiler-Generator

Aus jedem **Übersetzungsschema**, das auf einer **LL-Grammatik** basiert, kann man einen Ein-Pass-Top-Down-Compiler erzeugen, denn ein Übersetzungsschema setzt voraus, daß die Grammatik L-attributiert ist. Die Attribute können daher in einem links-rechts Tiefendurchlauf berechnet werden, der genau dem Ablauf der Top-Down Syntax-Analyse entspricht.

Eingabe: Übersetzungsschema auf der Basis einer LL(1)-Grammatik

Ausgabe: Compiler, der nach dem Verfahren des rekursiven Abstiegs arbeitet.

Algorithmus: Der Generator erzeugt einen LL-Parser nach der Methode des rekursiven Abstiegs (siehe Kap. 4.2), erweitert um Parameter und Aktionen.

1. Für jedes **Nonterminal** A der Grammatik wird eine Funktion generiert, die für jedes ererbte Attribut von A einen Eingangs-Parameter hat und den Wert des synthetisierten Attributs von A als Funktionswert zurückgibt. Für jedes Attribut eines jeden Symbols, das in Produktionen von A auf der rechten Seite auftritt, wird eine lokale Variable deklariert.
2. Zur Behandlung der rechten Seite einer Produktion sind folgende drei Fälle zu unterscheiden:
   (a) Für jedes **Terminal** t mit einem synthetisierten Attribut wird eine Wertzuweisung an die für das Attribut t.s deklarierte lokale Variable generiert.
   (b) Für jedes **Nonterminal** N wird ein Aufruf s = N(i_1, ..., i_n) generiert. Dabei sind i_1, ..., i_n die lokalen Variablen für die ererbten Attribute von N, und s ist die lokale Variable für das synthetisierte Attribut von N.
   (c) Für jede **Aktion** wird der Code der Aktion eingesetzt, wobei jede Referenz auf ein Attribut durch die lokale Variable für dieses Attribut ersetzt wird.

Falls ein Nonterminal mehrere synthetisierte Attribute besitzt, können diese zu einem Record zusammengefaßt werden. Dies ist bei einem Übersetzungsschema immer möglich, weil alle Attribute in einem Durchlauf ausgewertet werden (siehe Bsp. 5.10).

Optimierung. Die erzeugten Funktionen E und R lassen sich noch vereinfachen. Zunächst kann man alle unnötigen lokalen Variablen eliminieren (siehe Bsp. 5.11).

Die Prozedur `R` kann wegen der Endrekursion als Schleife in `E` eingesetzt werden (vgl. Kap. 4.2). Man beachte aber, daß der erste Aufruf von `T` vor der Schleife stehen muß, da die Verkettung nur bei nachfolgenden Aufrufen von `T` (innerhalb der Schleife) durchzuführen ist (Bsp. 5.12).

Die optimierte Version entspricht auf Grammatikebene einer RRPG (bzw. EBNF). Man kann Übersetzungsschemata auch direkt mit RRPGs formulieren. Dann treten allerdings wegen der Schleifen Attributvariablen mit Mehrfach-Zuweisungen auf (entsprechend der Variablen `r` im obigen Programm).

5.8 Bottom-Up-Compiler-Generator

Aus einem **Übersetzungsschema** auf der Basis einer **LR-Grammatik** kann problemlos ein Bottom-Up-Compiler konstruiert werden, wenn die Grammatik **S-attributiert**

```
class TopDown {
  String E() {
    String ri, ts, rs;
    ts = T(); ri = ts; rs = R(ri); return rs;
  }

  String R( String ri ) {
    String opx, ts, r1i, r1s;
    if( sym.equals("op") {
      opx = att; nextsym();
      ts = T(); r1i = ri+ts+opx; r1s = R(r1i); return r1s;
    } else return ri;
  }

  String T() {
    String es, idx; if( sym.equals("(") {
      nextsym(); es = E(); nextsym(); return es;
    } else {
      idx = att; nextsym(); return idx;
    }
  }
}
```

Der Aufruf `nextsym()` liefert das nächste Token und dessen Attribut in `sym` und `att`. Man startet den generierten Compiler durch den Aufruf `nextsym(); System.out.println(TopDown.E());`.

Beispiel 5.10: Aus dem Übersetzungsschema von Bsp. 5.9 erzeugt der beschriebene Generator folgenden Top-Down-Compiler (als Menge von Prozeduren)

ist. In diesem Fall stehen Aktionen nur am Ende der Produktionen und können daher mit den syntaktischen Reduktionen gekoppelt werden.

Für die synthetisierten Attribute wird ein **Attributkeller** verwendet. Er nimmt das zu einem gekellerten Grammatiksymbol gehörige Attribut auf. Zustandskeller und Attributkeller können „parallel“ mit den gleichen Kellerzeigern verwaltet werden. Wir zeigen das Prinzip anhand des folgenden Übersetzungsschemas.

$$\texttt{A} \to \texttt{X Y Z } \{\texttt{A.s = f(X.s,Y.s,Z.s)}\}$$

Die Reduktion mit gekoppelter Aktion läuft in folgenden Schritten ab:

- Vor der Reduktion:

```
String E() {
  return R(T());
}

String R( String ri ) {
  String opx;
  if( sym.equals("op") ) {
    opx = att; nextsym();
    return( ri+T()+opx );
  } else return ri;
}
```

Beispiel 5.11: Vereinfachung von E und R aus Bsp. 5.10.

```
String E() {
  String opx, r;
  r = T();
  while( sym.equals("op") ) {
    opx = att; nextsym();
    r = r+T()+opx;
  }
  return r;
}
```

Beispiel 5.12: Optimierte Version zu Bsp. 5.11.

				`top` ↓
Zustandskeller:	...	`X`	`Y`	`Z`
Attributkeller:	...	`X.s`	`Y.s`	`Z.s`

- Ermitteln von `ntop = top-r+1` (`r` =Länge der rechten Seite):

		`ntop` ↓		`top` ↓
Zustandskeller:	...	`X`	`Y`	`Z`
Attributkeller:	...	`X.s`	`Y.s`	`Z.s`

- Berechnen des Attributs `A.s` auf der Position `ntop` (und Reduktion im Zustandskeller)

- Setzen von `top = ntop`

```
                           top
                            ↓
Zustandskeller:     ...    A
Attributkeller:     ...    A.s
```

Die Anwendung dieses Prinzips wird in Bsp. 5.13 gezeigt.

E → E_1 op T	{E.s = E_1.s + T.s + op.x}
E → T	{E.s = T.s}
T → (E)	{T.s = E.s}
T → id	{T.s = id.x}

Daraus ergibt sich der folgende Code für die Aktionen:

Produktion	Code
E → E_1 op T	`a[ntop] = a[top-2] + a[top] + a[top-1]`
E → T	
T → (E)	`a[ntop] = a[top-1]`
T → id	

Beispiel 5.13: Gegeben sei ein S-attributiertes Übersetzungsschema zur Übersetzung von Infix nach Postfix gemäß der AG von Bsp. 5.4.

Der Attributkeller ist mit `a` bezeichnet. `top` zeigt auf den letzten Kellereintrag und wird bei shift-Operationen entsprechend erhöht. Immer wenn ein Symbol von der Eingabe in den Keller gelangt, wird der Wert des Attributs – den die Lexikalische Analyse liefert – in den Attributkeller gespeichert. Dies gilt z.B. für die Symbole `aop` und `id`. Falls ein Symbol kein Attribut besitzt, wie z.B. die Symbole `(` und `)`, bleibt die zugehörige Position des Attributkellers leer.

Die Zeiger `top` und `ntop` beziehen sich auf die Situation beim Schritt (2) im vorher skizzierten Ablauf. Bei der Produktion `E` → `T` ist kein Code für die Zuweisung notwendig, da die Attribute `T.s` und `E.s` die gleiche Position im Attributkeller einnehmen (`top =ntop`).

Man sieht, daß ein Compilergenerator den Code für die Aktionen generieren kann. Allerdings haben wir bisher nur synthetisierte Attribute berücksichtigt.

Ererbte Attribute

Da die Aktionen zur Berechnung ererbter Attribute im Übersetzungsschema nicht am Ende der Produktion stehen, können sie nicht mit der Reduktion gekoppelt werden. Man wendet daher eine **Transformation** der Grammatik an (siehe Bsp. 5.14).

Bottom-Up-Compiler-Generator. Der Generator zur Erzeugung eines Ein-Pass-Bottom-Up-Compilers aus einem Übersetzungsschema arbeitet insgesamt wie folgt:

```
S → A B {C.i = f(A.s,B.s)} C {S.s = g(A.s,C.s)}
C → D E {C.s = h(C.i,E.s)}
```

Die Aktion, die das ererbte Attribut `C.i` berechnet, wird durch eine eindeutige **Markierung**, d.h. ein neues Grammatiksymbol `M` mit der Produktion `M` $\rightarrow \varepsilon$ ersetzt. Das Symbol `M` belegt im Attributkeller vor `C` einen Platz für den Wert von `C.i`. Der Wert von `C.i` kann somit als Wert eines synthetisierten Attributs von `M` berechnet werden, d.h. die Aktion zur Berechnung von `C.i` wird der Produktion `M` $\rightarrow \varepsilon$ als Code zur Berechnung des synthetisierten Attributs `M.s` zugeordnet:

Produktion	Code
`S → A B M C`	`a[ntop] = g(a[top-3],a[top])`
`M →` ε	`a[ntop] = f(a[top-1],a[top])`

Vor der Reduktion von `M` zeigt `top` auf `B` und es gilt `ntop =top+1`, weil `r =0` ist. Nach der Reduktion ist der Wert von `C.i` auf der Position `M` (direkt vor `C`) gespeichert. Obwohl `C` selbst noch nicht im Keller steht, kann jede Produktion von `C` den Wert `C.i` bereits im Keller ansprechen:

Produktion	Code
`C → D E`	`a[ntop] = h(a[top-2],a[top])`

Beispiel 5.14: Ein Übersetzungsschema mit einem ererbten Attribut.

Eingabe: Übersetzungsschema auf der Basis einer LR(1)-Grammatik. Jede Aktion zur Berechnung eines ererbten Attributs `X.i` muß unmittelbar links von `X` stehen, die Aktion zur Berechnung des synthetisierten Attributs `A.s` muß am Ende stehen. Beide Forderungen stellen keine prinzipiellen Einschränkungen dar.

Ausgabe: Compiler, der nach dem Verfahren der LR-Analyse arbeitet.

Algorithmus: Der Generator erzeugt einen LR-Parser (siehe Kap. 4.3) mit Erweiterung des Zustandskellers um einen **Attributkeller**.

Für jede Produktion `A` $\rightarrow \ldots$ `X` $\ldots$ des Übersetzungsschemas sind die auftretenden Aktionen folgendermaßen zu behandeln:

- Aus der Aktion zur Berechnung des **synthetisierten** Attributs `A.s` wird der entsprechende Code generiert und dem Reduktionsschritt der Produktion zugeordnet.
- Jede Aktion zur Berechnung eines **ererbten** Attributs `X.i` wird durch eine eindeutige Markierung `M` ersetzt und die Grammatik um die Produktion `M` $\rightarrow \varepsilon$ ergänzt. Aus der Aktion selbst wird der entsprechende

Code generiert und dem Reduktionsschritt der Produktion $\texttt{M} \rightarrow \varepsilon$ zugeordnet.

Die folgenden zwei Arten von Kopieraktionen können ignoriert werden, da der Wert, der an `X.i` zugewiesen werden soll, bereits auf der richtigen Position für `X.i` (d.h. direkt vor `X`) im Attributkeller steht:

$$\texttt{A} \rightarrow \{\texttt{X.i = A.i}\}\texttt{X} \ldots$$
$$\texttt{A} \rightarrow \ldots\ \texttt{Y}\{\texttt{X.i = Y.s}\}\texttt{X} \ldots$$

- Für die geänderte Grammatik wird ein LR-Parser erzeugt.

Das Problem der Transformation durch Markierungen ist aber, daß die Grammatik unter Umständen die LR-Eigenschaft verliert, sodaß Konflikte in den Parser-Tabellen auftreten. Bsp. 5.15 zeigt einen solchen Fall.

5.9 Ein-Pass-Übersetzung

Notwendige Voraussetzung zur vollständigen Attributberechnung während der Syntax-Analyse ist eine L-attributierte Grammatik (LAG). Die LAG muß entweder auf einer LL-Grammatik basieren (LL-attributierte Grammatik, LLAG) oder auf einer LR-Grammatik, die durch die oben angegebene Transformation ihre LR-Eigenschaft nicht verliert (LR-attributierte Grammatik, LRAG).

Jede LLAG ist auch LRAG, da durch eingefügte ε-Produktionen ohne Alternativen die LL-Eigenschaft nicht zerstört wird.

Bsp. 5.15 zeigt drei mögliche Lösungen eines Problems entsprechend den AG-Klassen von Abb. 5.16.

5.10 Mehrpass-Übersetzung

Falls die Einpass-Übersetzung nicht möglich ist, erzeugt die Syntax-Analyse einen expliziten Ableitungsbaum (Syntaxbaum), der von einem **Attribut-Auswerter** beliebig durchlaufen werden kann. Im allgemeinsten Fall wird für jeden Syntaxbaum der Abhängigkeitsgraph konstruiert und die Attributauswertung aufgrund der Abhängigkeiten vorgenommen. Diese Methode ist sehr aufwendig, da sie die statisch bekannten Abhängigkeiten der Grammatik nicht ausnützt. Wir behandeln im folgenden zwei effizientere Methoden.

Pass-orientierte Auswerter durchlaufen den Baum ein oder mehrmals nach einer *globalen Strategie*, z.B. mit links-rechts-Tiefensuche (LAG-Auswerter). Bei jedem Durchgang werden Attribute ausgewertet, die aufgrund der bis dahin verfügbaren Attributwerte berechnet werden können. Durch Analyse der Attributabhängigkeiten wird vor der Auswertung bestimmt, welche Attribute in den einzelnen Durchgängen zu berechnen sind.

Abb. 5.17 zeigt zwei alternative Produktionen für ein Nonterminal A und die Attributabhängigkeiten. Die zweite Produktion erfordert zwei Durchläufe eines LAG-Auswerters: Im ersten Durchgang wird `R.s` und im zweiten `Q.s` berechnet.

LLAG:

$$\begin{aligned}
&\texttt{S} \rightarrow \texttt{id}\ \{\texttt{L.i = 1}\}\ \texttt{L}\ \{\texttt{S.s = L.s}\}\\
&\texttt{L} \rightarrow \texttt{id}\ \{\texttt{L}_1\texttt{.i = L.i+1}\}\ \texttt{L}_1\ \{\texttt{L.s = L}_1\texttt{.s}\}\\
&\texttt{L} \rightarrow \varepsilon\ \{\texttt{L.s = L.i}\}
\end{aligned}$$

LRAG:

$$\begin{aligned}
&\texttt{S} \rightarrow \texttt{L}\ \{\texttt{S.s = L.s}\}\\
&\texttt{L} \rightarrow \texttt{L}_1\ \texttt{id}\ \{\texttt{L.s = L}_1\texttt{.s+1}\}\\
&\texttt{L} \rightarrow \texttt{id}\ \{\texttt{L.s = 1}\}
\end{aligned}$$

LAG, aber nicht LRAG:

$$\begin{aligned}
&\texttt{S} \rightarrow \{\texttt{L.i = 0}\}\ \texttt{L}\ \{\texttt{S.s = L.s}\}\\
&\texttt{L} \rightarrow \{\texttt{L}_1\texttt{.i = L.i+1}\}\ \texttt{L}_1\ \texttt{id}\ \{\texttt{L.s = L}_1\texttt{.s}\}\\
&\texttt{L} \rightarrow \texttt{id}\ \{\texttt{L.s = L.i+1}\}
\end{aligned}$$

Beispiel 5.15: Die Anzahl der Bezeichner in einer Liste soll berechnet werden.

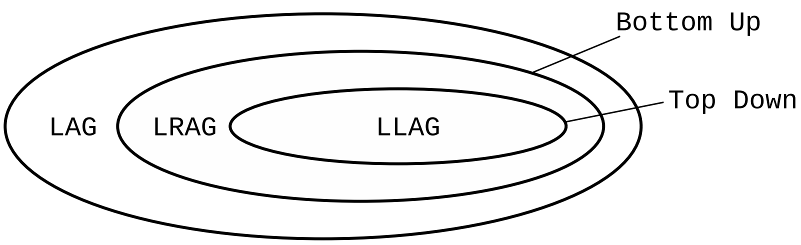


Abbildung 5.16: AG-Klassen für Ein-Pass-Übersetzung

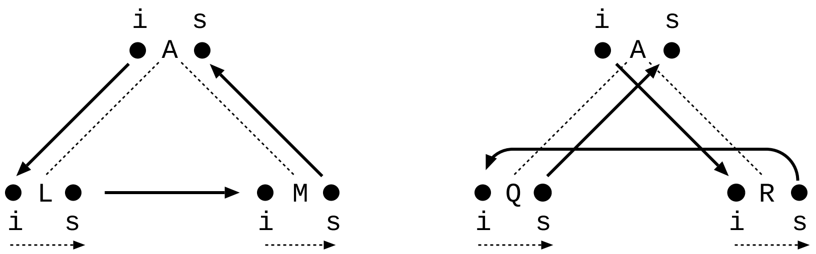


Abbildung 5.17: Eine L-attributierte und eine nicht-L-attributierte Produktion für A

In Abb. 5.18 ist eine Situation dargestellt, die bei der Semantischen Analyse des Overloading in Ada auftritt: Das ererbte Attribut `X.i` ist vom synthetisierten Attribut `X.s` abhängig. Der LAG-Auswerter berechnet im ersten Durchgang `X.s` und im zweiten Durchgang `X.c`.

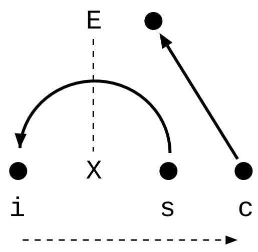


Abbildung 5.18: Nicht-L-attributierte Produktion E→ X

Visit-orientierte Auswerter durchlaufen den Baum nicht nach einer globalen Strategie, sondern entsprechend den Attributabhängigkeiten der Teilbäume.

Ein Visit-orientierter Auswerter kann z.B. konstruiert werden, indem man jedem Nonterminal eine eigene Auswertungsfunktion zuordnet.

```
public SType A( Node n, IType ai ) {
  switch( Produktion am Knoten n ) {
    case 'A → L M':                              // L vor M aufrufen
      li = ...ai...
      ls = L(child(n,1),li);
      mi = ...ls...
      ms = M(child(n,2),mi)
      return ...ms...
    case 'A → Q R':                              // R vor q aufrufen
      ri = ...ai...
      rs = R(child(n,2),ri)
      qi = ...rs...
      qs = Q(child(n,1),qi)
      return ...qs...
  }
}
```

Algorithmus 5.19: Auswertungsfunktion für das Nonterminal `A`.

Die Abhängigkeiten in Abb. 5.17 zeigen, daß bei der Produktion `A`→ `LM` die Auswertungsfunktion `L` vor `M` und bei der Produktion `A`→ `QR` die Funktion `R` vor `Q` aufgerufen werden muß. Die unterschiedlichen Aufruffolgen können in der Funktion

`A` berücksichtigt werden. Die Funktion `A` ist in Alg. 5.19 dargestellt. Dabei ist angenommen, daß die Knoten des Baums durch eine eindeutige Nummer `n` identifiziert sind und die Funktion `child(n,k)` die Nummer des k-ten direkten Nachfolgers von `n` liefert. Außerdem sei zu jedem Knoten gespeichert, welche Produktion angewandt wurde.

Die Auswertungsfunktion `A` bringt gegenüber dem LAG-Auswerter den Vorteil, daß jeder Teilbaum `A` in *einem* Durchlauf ausgewertet werden kann.

Im Beispiel von Abb. 5.18 benötigt man für `X` zwei Funktionen `X1` und `X2`, die in `E` nacheinander aufzurufen sind: `X1` berechnet `X.s` und `X2` berechnet `X.c`. Der Teilbaum `X` wird also zweimal besucht, d.h. der Visit-orientierte Auswerter verhält sich in diesem Beispiel wie der Pass-orientierte LAG-Auswerter.

Kapitel 6

Semantische Analyse

Dieses Kapitel behandelt die Semantische Analyse für Programmiersprachen mit einem strengen statischen Typsystem wie z.B. Pascal, Modula und Ada. Die wesentliche Aufgabe der Semantischen Analyse ist die **Typ-Überprüfung** (*type checking*). Dabei muß überprüft werden, ob die Verwendung von Objekten mit ihrer Deklaration verträglich ist.

Grundlage für die Typ-Überprüfung ist das **Typsystem** der jeweiligen Programmiersprache. Als Typsystem bezeichnet man die Menge der Regeln, die den Elementen einer Programmiersprache Typen zuordnen. Typen lassen sich formal durch Typ-Ausdrücke beschreiben und durch Graphen darstellen. Die zweckmäßige interne Darstellung von Typen ist für eine effiziente Typ-Überprüfung von großer Bedeutung. In Pascal und Modula beeinflußt die interne Darstellung sogar das Sprachkonzept bezüglich der Typ-Äquivalenz.

Der Zusammenhang zwischen Deklaration und Verwendung läßt sich nicht durch eine kontextfreie Grammatik beschreiben. Die Semantische Analyse ist daher ein klassisches Anwendungsfeld der Attributierten Grammatik. Zentrale Datenstruktur der Semantischen Analyse ist die **Symboltabelle** als 'Datenbasis' für die Typ-Überprüfung.

In den meisten Programmiersprachen treten Operatoren auf, deren Typ nicht eindeutig deklariert ist. Die Semantische Analyse muß den Typ solcher Operatoren bzw. den Typ entsprechender Teilausdrücke aus dem jeweiligen Kontext ableiten. Dabei sind auch die Regeln bezüglich impliziter Typ-Konversionen zu beachten.

Die Semantische Analyse soll die **statische Korrektheit** des Quellprogramms sicherstellen. Neben der Typanalyse fallen daher unter Umständen noch einige Nebenaufgaben an, z.B. in Modula die Überprüfung, ob der Bezeichner von Modulen und Prozeduren an ihrem Ende korrekt wiederholt wird. Solche Prüfungen benötigen aber keine speziellen Verfahren und werden daher in diesem Kapitel nicht explizit behandelt.

6.1 Typen

Typsysteme

Ein Typsystem ist die Menge der Regeln einer Programmiersprache, die den Programmelementen Typen zuordnen und die Äquivalenz und Kompatibilität von Typen definieren. Man kann **statische** und **dynamische Typsysteme** unterscheiden, je nachdem, ob die Typregeln zur Übersetzungszeit (=statisch) oder zur Laufzeit (=dynamisch) zur Anwendung kommen.

Statische Typsysteme haben den Vorteil, daß sie die Korrektheit (Sicherheit) und die Effizienz des Programms erhöhen. Der Compiler kann die Typen überprüfen und im erzeugten Programm die genau passenden Operatoren und Speicherbereiche festlegen. Bei dynamischen Typsystemen muß die Typ-Überprüfung generiert bzw. in das Laufzeitsystem eingebaut werden. Das Laufzeitverhalten ist daher entsprechend weniger effizient. Beide Systeme können auch kombiniert werden.

Ein Typsystem kann man als 'Filter' ansehen, das die Menge von Programmen bzw. Prozessen auf typkorrekte Teilmengen einschränkt. Dies wird in Abb. 6.1 anschaulich dargestellt.

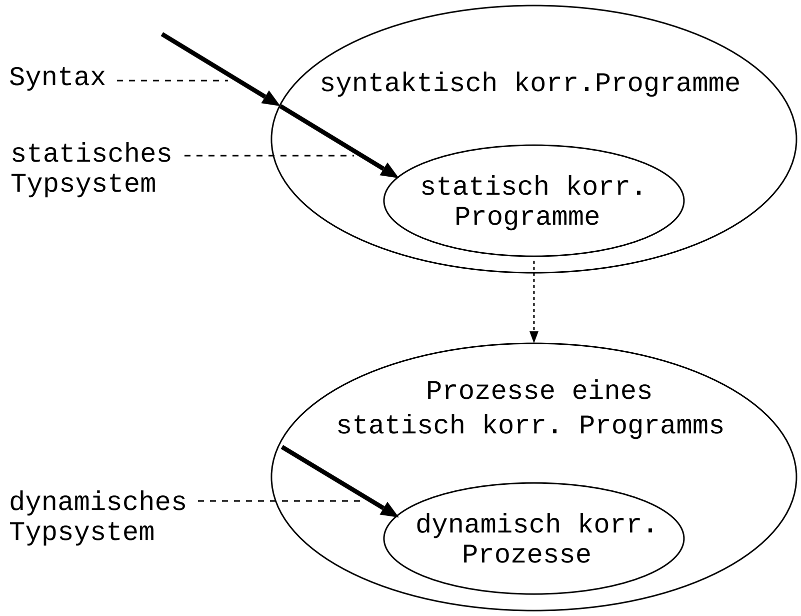


Abbildung 6.1: Statisches und dynamisches Typsystem

Typ-Ausdrücke

Ein Typ kann formal durch einen **Typ-Ausdruck** beschrieben werden. Typ-Ausdrücke bilden die Grundlage zur internen Darstellung von Typen. Daher werden wir Typ-Ausdrücke für eine 'typische' Programmiersprache definieren:

Ein **Basistyp** ist ein Typ-Ausdruck. Basistypen sind z.B. `bool`,`char`,`int` und `real`.

Ein **Typ-Konstruktor**, angewendet auf einen oder mehrere Typ-Ausdrücke, ist ein Typ-Ausdruck. Konstruktoren sind z.B.:

Array: `array(n,T)` mit `T` als Elementtyp und `n` = Anzahl der Elemente.

Produkt: $\mathtt{T}_1 \times \mathtt{T}_2$ (z.B. für Parameterlisten)

Record: $\mathtt{record((n}_1 : \mathtt{T}_1\mathtt{)} \times \ldots \times \mathtt{(n}_k : \mathtt{T}_k\mathtt{))}$ ($\mathtt{n}_i$: Feldnamen)

Zeiger: `pointer(T)`

Funktion: $\mathtt{T}_1 \rightarrow \mathtt{T}_2$

Ein Typ-Ausdruck kann als Baum dargestellt und implementiert werden. Endknoten sind mit Basistypen, alle anderen Knoten mit Typkonstruktoren markiert. Abb. 6.2 zeigt als Beispiel den Baum für einen Funktionstyp.

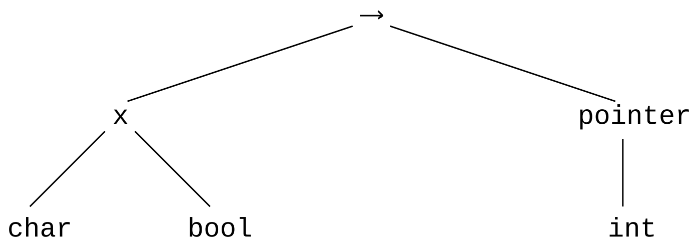


Abbildung 6.2: Typ-Ausdruck `char` $\times$ `bool` $\rightarrow$ `pointer(int)`

Typnamen und Typ-Äquivalenz

In vielen Programmiersprachen können Typnamen definiert werden. Dabei ist die Frage der Typ-Äquivalenz zu klären (siehe Bsp. 6.3).

```
TYPE  link =  POINTER TO cell;
 VAR  next :  link;
      last :  link;
      p :     POINTER TO cell;
      q,r :   POINTER TO cell;
```

Beispiel 6.3: Eine Typ- und Variablendeklaration (Modula-2).

Man unterscheidet im wesentlichen zwei Möglichkeiten, um die Äquivalenz von Typen zu definieren:

Struktur-Äquivalenz bedeutet, daß Typen mit gleicher Struktur gleich sind. Im Bsp. 6.3 haben dann alle Variablen den gleichen Typ.

Namen-Äquivalenz bedeutet, daß Typen mit gleichem Namen gleich sind. Jeder anonymen Typ-Konstruktion wird ein eindeutiger impliziter Name zugeordnet. Im Beispiel haben dann `next` und `last`, sowie `q` und `r` jeweils den gleichen Typ. Diese Art der Typ-Äquivalenz findet man in Pascal und Modula.

Die Namen-Äquivalenz kann anhand des Typbaums dargestellt und implementiert werden, indem man den Knoten (explizite oder implizite) Typnamen zuordnet. Zwei Typen sind dann gleich, wenn sie durch denselben Knoten repräsentiert werden.

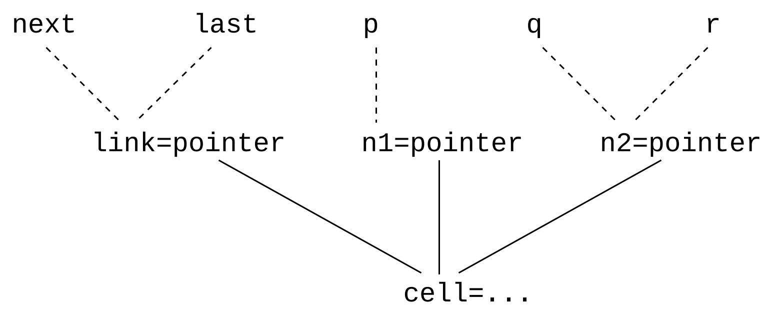


Abbildung 6.4: Namen-Äquivalenz

6.2 Symboltabelle

Die Namen aller im Quellprogramm deklarierten Objekte und deren Typen werden in einer **Symboltabelle** gesammelt. Jeder Eintrag enthält den Namen und den Typ des deklarierten Objekts. Namen werden durch das Attribut `id.x` repräsentiert, Typen werden als Zeiger auf den Typbaum dargestellt.

Die Symboltabelle wird bei der Verarbeitung der Deklarationen aufgebaut und bei der Verarbeitung der Anweisungen verwendet. In einer Attributierten Grammatik (ohne Seiteneffekte) tritt die Symboltabelle daher als Attribut auf, dessen Wert im Deklarationsteil berechnet und an den Anweisungsteil übergeben wird. Die AG in Bsp. 6.5 zeigt, wie die Symboltabelle aus den Deklarationen aufgebaut werden kann. Das Nonterminal `D` besitzt ein ererbtes und ein synthetisiertes Attribut. Der Wert des ererbten Attributs ist jeweils die Symboltabelle *vor* der Verarbeitung von `D`, der Wert des synthetisierten Attributs ist die Symboltabelle *nach* der Verarbeitung von `D`. Auf der obersten Ebene (erste Produktion) wird mit der leeren Tabelle begonnen und die vollständige Tabelle zurückgeliefert. Jede Deklaration einer Variablen (dritte Produktion) erzeugt einen neuen Eintrag in der Tabelle. Im Beispiel fassen wir die Symboltabelle als Liste auf, wobei die Operation || ein Element am Ende der Liste anhängt.

Produktion	Regeln
P → D begin S end	D.i = ()
	S.i = D.s
D → D$_1$; D$_2$	D$_1$.i = D.i
	D$_2$.i = D$_1$.s
	D.s = D$_2$.s
D → id : T	D.s = D.i ‖ (id.x,T.type)
T → int	T.type = int
T → real	T.type = real
T → arr [inum] of T$_1$	T.type = array(inum.x,T$_1$.type)

Beispiel 6.5: AG zur Erzeugung einer Symboltabelle aus Variablendeklarationen (Typen: `integer`, `real` und `array` mit Untergrenze 0).

Basistypen sind vorgegebene Endknoten von Typbäumen. Für strukturierte Typen muß aus der Deklaration der entsprechende Typbaum konstruiert werden. Die letzte Zeile der AG in Bsp. 6.5 definiert die Konstruktion des Typbaums für mehrdimensionale Arrays. Der Typbaum wird im synthetisierten Attribut `T.type` aufgebaut. Abb. 6.6 veranschaulicht die Konstruktion am attributierten Ableitungsbaum für ein zweidimensionales Array.

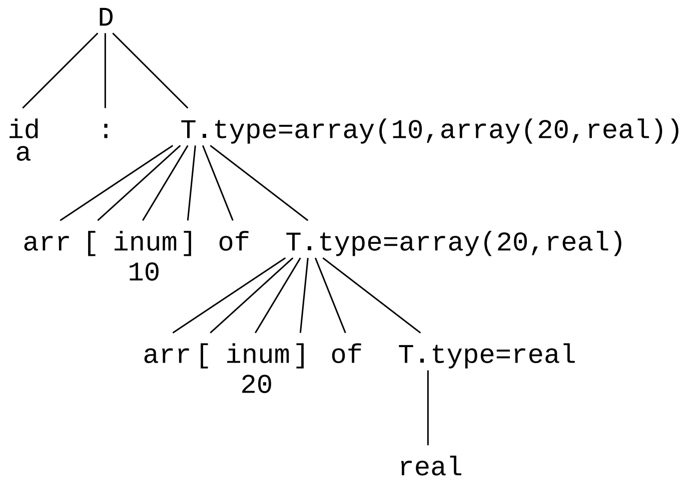


Abbildung 6.6: Attributierter Ableitungsbaum für die Deklaration `a:ARRAY[10] OF ARRAY[20] OF REAL`

Beim Aufbau der Symboltabelle müssen die Gültigkeitsbereiche von geschachtelten Prozedurdeklarationen berücksichtigt werden, d.h. innerhalb jeder Prozedurdeklaration ist die Symboltabelle um die lokalen Deklarationen zu erweitern. Dies läßt sich auf einfache Weise durch eine AG beschreiben (siehe Bsp. 6.7). Die Prozedurdeklaration `D` erweitert die Tabelle zunächst um den Eintrag für den Prozedurnamen. Die so erweiterte Tabelle wird einerseits nach außen zurückgegeben (da der Name außen sichtbar ist) und andererseits an den inneren Prozedurblock `P` weitergegeben. Dort wird die Tabelle durch die lokalen Deklarationen `D` ergänzt und an den Anweisungsteil `S` der Prozedur übergeben.

Produktion	Regeln
`D` $\rightarrow$ `proc id P`	`h = D.i` $\|$ `(id.x,proc)`
	`D.s = h`
	`P.i = h`
`P` $\rightarrow$ `D begin S end`	`D.i = P.i`
	`S.i = D.s`

Beispiel 6.7: AG zur Erzeugung der Symboltabelle bei geschachtelten Prozedurdeklarationen

6.3 Typ-Überprüfung

Bei imperativen Sprachen ist zwischen Anweisungen und Ausdrücken zu unterscheiden. Die Typ-Überprüfung bezieht sich nur auf Ausdrücke. Da Ausdrücke aber innerhalb von Anweisungen auftreten, muß die Symboltabelle als Attribut über die Anweisungen zu den Ausdrücken bis zu den elementaren Operanden weitergegeben werden (siehe Bsp. 6.8). Wir nehmen an, daß die Zugriffsfunktion `looktype(n,t)` zu einem gegebenen Namen `n` den in der Tabelle `t` eingetragenen Typ liefert. Wenn die Symboltabelle in der vorher beschriebenen Form als Liste aufgebaut ist, kann man die Funktion `looktype` durch sequentielle Rückwärtssuche implementieren, sodaß die lokalen Deklarationen zuerst gefunden werden.

Die eigentliche Typ-Überprüfung erfolgt in Bsp. 6.8 mithilfe des synthetisierten Attributs `type` aufsteigend im Ableitungsbaum. Dabei wird der Typ aller Teilausdrücke bestimmt, und mehrdeutige Operatoren werden eindeutig identifiziert. In Modula hat `"+"` zum Beispiel die zwei „überladenen“ Bedeutungen `int` $\times$ `int` $\rightarrow$ `int` und `real` $\times$ `real` $\rightarrow$ `real`.

Die Typ-Überprüfung verläuft ähnlich wie eine Berechnung des Ausdrucks. Man kann daher die Semantische Analyse als Interpreter ansehen, der Ausdrücke auf der 'Typebene' interpretiert. Anstelle der konkreten Werte, die zur Übersetzungszeit noch nicht vorliegen, werden nur die Typen als abstrakte Werte 'berechnet'. Diese Berechnung liefert als Ergebnis den Typ des Ausdrucks und in den Zwischenschritten die Typen der Teilausdrücke und Operatoren – oder eine Fehlermeldung.

Zur besseren Übersicht wird in den nachfolgenden Beispielen die Weitergabe der Symboltabelle als Attribut nicht mehr explizit angegeben.

Produktion	Regeln
`P → D begin S end`	`D.i = ()` `S.i = D.s`
`S → S₁ ; S₂`	`S₁.i = S.i` `S₂.i = S.i`
`S → id = E`	`E.i = S.i` `if( looktype(id.x,S.i) != E.type )` error

`E → E₁ + E₂`	`E₁.i = E.i`
	`E₂.i = E.i`
	`E.type =`
	`  if(E₁.type == int && E₂.type == int) int`
	`  else if(E₁.type == real && E₂.type == real) real`
	`  else error`

`E → id`	`E.type = looktype(id.x,E.i))`
`E → inum`	`E.type = int`
`E → rnum`	`E.type = real`

Beispiel 6.8: AG zur Typ-Überprüfung (Modula).

6.4 Typ-Konversion

Man unterscheidet explizite und implizite Typ-Konversion. Bei der **expliziten** Typ-Konversion muß eine Konversionsfunktion im Programm geschrieben werden (z.B. `FLOAT(i)` in Modula). Solche Funktionen können von der Semantischen Analyse wie jede andere Funktion behandelt werden.

Implizite Konversionen dagegen müssen von der Semantischen Analyse besonders berücksichtigt werden, z.B. in Pascal bei der Verknüpfung von integer- und real-Operanden, wobei integer nach real implizit konvertiert wird.

Die AG in Bsp. 6.9 überprüft die Typen unter Berücksichtigung der Regeln für implizite Konversionen in Pascal. Die Konversions-Operationen müssen bei der Code-Erzeugung an den entsprechenden Stellen generiert werden. Der Ablauf einer Typ-Überprüfung kann am attributierten Ableitungsbaum dargestellt werden. Die Abb. 6.10 basiert auf den Regeln von Bsp. 6.9. Die verwendeten Variablen seien wie folgt deklariert: `a,b:INT;y:REAL`.

Typ-Überprüfung und Operator-Identifizierung können bei den meisten Programmiersprachen in einem Durchlauf des Ausdrucksbaums mit einem synthetisierten Attribut erfolgen. Voraussetzung dafür ist, daß der Ergebnistyp jeder Operation eindeutig durch die Operandentypen bestimmt ist. Das gilt z.B. auch für C++, da die Operandentypen überladener Operatoren eindeutig sein müssen. Im Gegensatz dazu werden in Ada überladene Operatoren durch die Operandentypen *und den*

Produktion	Regeln

`E → E`$_1$` + E`$_2$	`E.type =`
	`  if( E`$_1$`.type == int && E`$_2$`.type == int) int`
	`  else if( E`$_1$`.type == real && E`$_2$`.type == real) real`
	`  else if( E`$_1$`.type == int && E`$_2$`.type == real) real`
	`  else if( E`$_1$`.type == real && E`$_2$`.type == int) real`
	`  else error`

Beispiel 6.9: AG zur Typ-Überprüfung (Pascal).

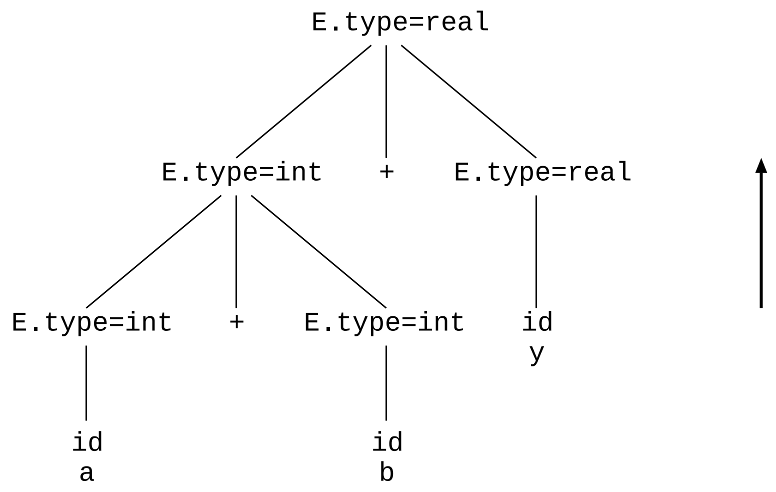


Abbildung 6.10: Typ-Überprüfung für `a+b+y`

Ergebnistyp identifiziert. Dadurch wird die Analyse wesentlich komplizierter.

6.5 Overloading

Overloading bedeutet, daß ein Symbol (Bezeichner, Operator) mit mehreren Bedeutungen 'überladen' wird. Die aktuelle Bedeutung muß jeweils aus dem Kontext ermittelt werden. In den meisten Programmiersprachen ist z.B. der Operator `"+"` überladen:

1. `int` × `int` → `int`
2. `real` × `real` → `real`

Der Typ von `"+"` kann hier eindeutig aus dem Typ der Operanden bestimmt werden. In Ada können jedoch zusätzlich im Programm weitere Typen für `"+"` definiert werden, und zwar auch solche mit den gleichen Operandentypen, z.B.

```
function "+"(i,j:integer) return real;
```

Damit ist der Operator `"+"` dreifach überladen:

1. `int` $\times$ `int` $\rightarrow$ `int`
2. `real` $\times$ `real` $\rightarrow$ `real`
3. `int` $\times$ `int` $\rightarrow$ `real`

Die beiden Operanden allein identifizieren den Operator nicht mehr eindeutig. Der Typ eines Ausdrucks `a+b` mit zwei `int`-Operanden ist z.B. `int` oder `real`, d.h. ein Element aus der Menge {`int,real`}. Der Typ des Operators `"+"` ist ein Element aus der Menge {`int` $\times$ `int` $\rightarrow$ `int,int` $\times$ `int` $\rightarrow$ `real`}.

Welcher Operator anzuwenden ist, muß durch den jeweiligen Kontext, d.h. durch den in den Kontext passenden Ergebnistyp des Ausdrucks, entschieden werden. Falls auch dadurch keine eindeutige Entscheidung möglich ist, liegt gemäß Sprachdefinition von Ada ein Typfehler vor.

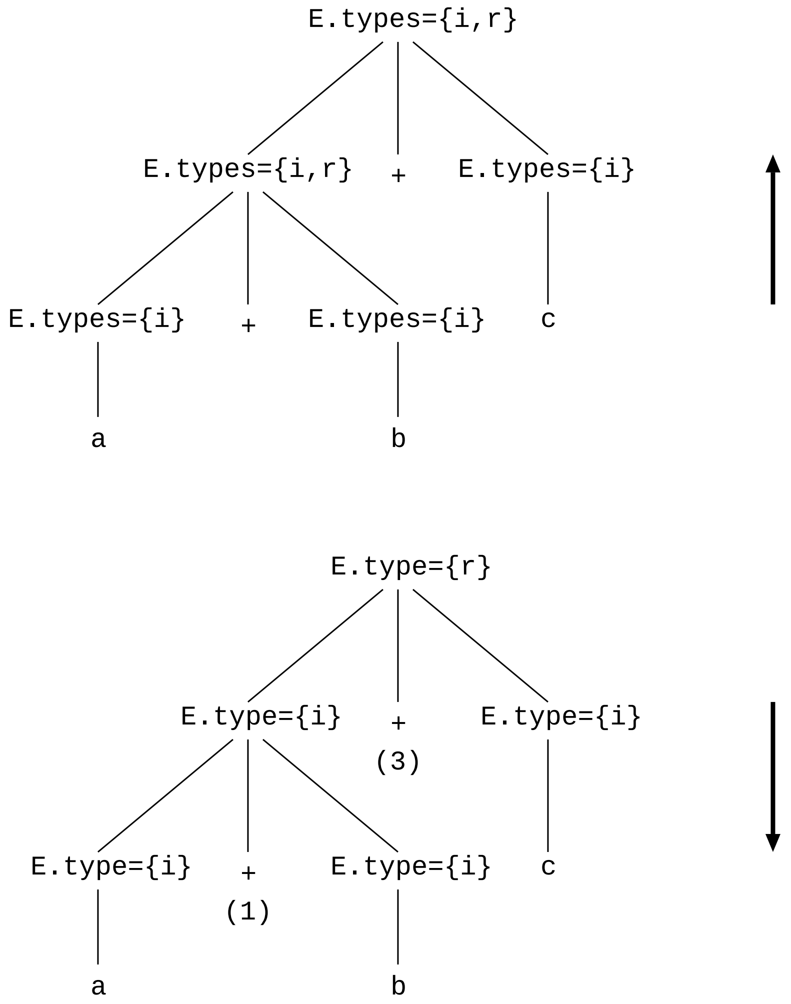
Zur Analyse eines Ausdrucks muß der zugehörige Baum in zwei Richtungen durchlaufen werden:

- Aufsteigende Berechnung der Typmengen an den Knoten.
- Absteigende Berechnung des eindeutigen Typs für jeden Knoten.

Bei der absteigenden Berechnung muß sich die Typmenge jedes Knotens auf genau ein Element reduzieren, damit die anzuwendenden Operatoren eindeutig bestimmt sind. Das Verfahren wird in Bsp. 6.11 veranschaulicht. Zunächst werden im Attribut `types` die Typmengen der Knoten *aufsteigend* anhand der definierten Typen der Bezeichner und des Operators `"+"` berechnet. Im Beispiel ergibt sich an der Wurzel ein eindeutiger Typ. Dieser wird in das Attribut `type` übernommen, und *absteigend* werden jene Typen gestrichen, für die es keine passende Operatordefinition gibt. Im Beispiel ist der Typ `i` aus der Menge {`i,r`} zu streichen, da `i` $\times$ `r` $\rightarrow$ `r` für den Operator `"+"` nicht definiert ist.

Die AG in Bsp. 6.12 beschreibt das Verfahren formal, wobei angenommen ist, daß die Funktion `looktypes` die Typmengen der deklarierten Bezeichner bzw. Operatoren aus der Symboltabelle liefert.

Aus der Sicht der Attributierten Grammatik erkennt man die Notwendigkeit für zwei Baumdurchläufe an der Produktion `E'` $\rightarrow$ `E` (Wurzel des Ableitungsbaumes). Das ererbte Attribut `type` des Knotens `E` hängt vom synthetisierten Attribut `types` desselben Knotens ab. Die AG von Bsp. 6.12 ist also nicht L-attributiert.



Beispiel 6.11: Gegeben seien die Variablen a und b vom Typ i (integer) und y vom Typ r (real). Der Operator `"+"` sei mit $\{(1)\ i \times i \rightarrow i,\ (2)\ r \times r \rightarrow r,\ (3)\ i \times i \rightarrow r\}$ überladen (Ada). Die Typ-Überprüfung des Ausdrucks a+b+y wird durch die gezeigten attributierten Ableitungsbäume veranschaulicht

Produktion	Regeln
$\mathtt{S} \rightarrow \mathtt{id = E}$	$\mathtt{E.type = looktypes(id.x)} \cap \mathtt{E.types}$
$\mathtt{E} \rightarrow \mathtt{E_1\ op\ E_2}$	$\mathtt{E.types} = \{\mathtt{t} \mid (\mathtt{t_1} \times \mathtt{t_2} \rightarrow \mathtt{t}) \in \mathtt{looktypes(op.x)}$ $\wedge\ \mathtt{t_1} \in \mathtt{E_1.types} \wedge \mathtt{t_2} \in \mathtt{E_2.types}\}$
	$\mathtt{E_1.type} = \{\mathtt{t_1} \mid (\mathtt{t_1} \times \mathtt{t_2} \rightarrow \mathtt{t}) \in \mathtt{looktypes(op.x)}$ $\wedge\ \mathtt{t_1} \in \mathtt{E_1.types} \wedge \mathtt{t_2} \in \mathtt{E_2.types}$ $\wedge\ \mathtt{t = E.type}\}$
	$\mathtt{E_2.type} = \{\mathtt{t_2} \mid (\mathtt{t_1} \times \mathtt{t_2} \rightarrow \mathtt{t}) \in \mathtt{looktypes(op.x)}$ $\wedge\ \mathtt{t_1} \in \mathtt{E_1.types} \wedge \mathtt{t_2} \in \mathtt{E_2.types}$ $\wedge\ \mathtt{t = E.type}\}$
$\mathtt{E} \rightarrow \mathtt{id}$	$\mathtt{E.types = looktypes(id.x)}$

Falls für das Attribut `E.type` mehrelementige Mengen auftreten wird ein Fehler ausgelöst.

Beispiel 6.12: AG zur Typ-Überprüfung (Ada)

Kapitel 7

Zwischendarstellungen

Ein Quellprogramm, das Syntax-Analyse und Semantische Analyse durchlaufen hat, kann prinzipiell (durch einen Interpreter) ausgeführt werden. Die Darstellung eines Programms in diesem Stadium bezeichnen wir als **Zwischendarstellung**.

Die allgemeinste Form der Zwischendarstellung ist der (attributierte) **Syntaxbaum** oder **Operatorbaum**, der das Programm als hierarchische Struktur von Operatoren und Operanden darstellt.

Bei einem **linearen Zwischencode** wird das Programm als lineare Folge von Operatoren und Operanden dargestellt. Einfache Codes dieser Art sind **Postfix**- und **Prefix**-Code. Der Postfix-Code hat als Zwischencode für Interpreter besondere Bedeutung, da er nur einen Stack für die Operanden erfordert. Als Zwischencode im Compiler wird häufig der **Quadrupel**-Code verwendet, ein linearer Zwischencode, dessen Elemente Quadrupel (4-Tupel) sind.

Wir werden in diesem Kapitel zunächst die elementaren Zusammenhänge zwischen Syntaxbaum und linearen Codes zeigen und dann den Quadrupel-Code eingehend behandeln. Insbesondere wird unter Verwendung Attributierter Grammatiken gezeigt, wie aus typischen Konstrukten der Quellsprache entsprechende Folgen von Quadrupeln erzeugt werden können.

Der Quadrupelcode eignet sich als Grundlage für einfache Verfahren der Codeerzeugung. Er kann aber auch verwendet werden, um Kontroll- und Datenflußgraphen zu konstruieren, die man zur optimalen Codeerzeugung benötigt.

7.1 Syntaxbaum

Ein Syntaxbaum (Operatorbaum) stellt das Quellprogramm als Struktur von Operatoren und Operanden dar. Jeder Knoten des Baums bestimmt eine Operation, deren Operanden die anhängenden Teilbäume sind. Abb. 7.1 skizziert zwei Beispiele.

Bei LR-Grammatiken erhält man den Syntaxbaum auf einfache Weise durch Vergröberung (Abstraktion) des Ableitungsbaums. Bei LL-Grammatiken wird wegen der fehlenden Linksrekursion die Programmstruktur weniger deutlich durch den Ableitungsbaum dargestellt, so daß die Erzeugung des Syntaxbaums aufwendiger ist. In jedem Fall aber ist es möglich, den Syntaxbaum während der Syntax-Analyse explizit zu erzeugen.

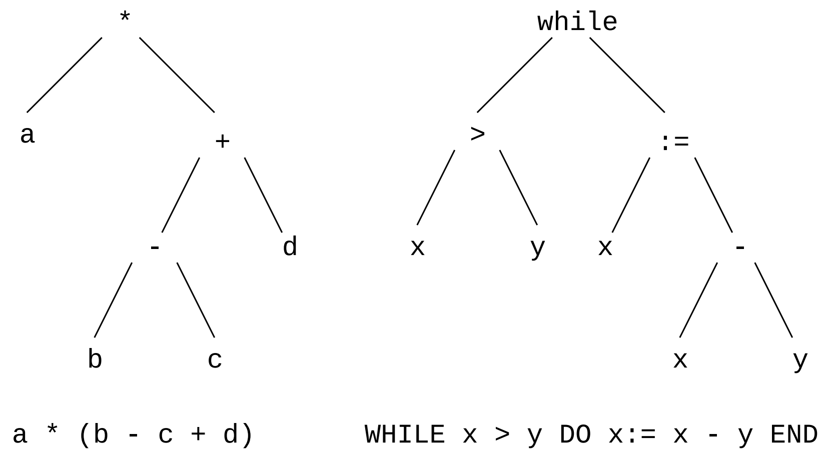


Abbildung 7.1: Syntaxbäume

Der Syntaxbaum ist als universelle Zwischendarstellung des Quellprogramms für weitere Transformationen in Richtung eines Zielprogramms gut geeignet. Er kann prinzipiell aber auch als Programm durch einen rekursiven Interpreter ausgeführt werden (mit entsprechenden Eingabedaten). Bei einem WHILE-Operator müßte der rekursive Interpreter z.B. den Schleifen-Unterbaum wiederholt aufrufen, bis die Wiederholungsbedingung nicht mehr erfüllt ist. Im Anhang B wird die Implementierung eines rekursiven Interpreters in Prolog gezeigt.

Zur Erzeugung von Maschinencode ist jedenfalls eine Linearisierung des Baums erforderlich. Im folgenden zeigen wir den Zusammenhang zwischen Syntaxbaum und den linearen Formen Postfix-, Prefix- und Quadrupel-Code.

7.2 Postfix und Prefix

Postfix- und Prefix-Code sind **klammerfreie Folgen** von Operatoren und Operanden. Dabei ist zu beachten, daß die Stelligkeit jedes Operators eindeutig ist (z.B. müssen unäres und binäres `"-"` unterschiedlich dargestellt werden).

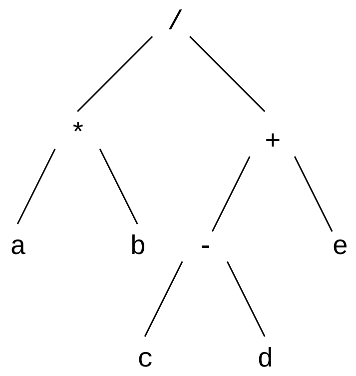
Die **Postfix-Folge** (*postorder*) kann wie folgt aus dem Baum erzeugt werden: Bei einem Tiefendurchlauf wird jeder Knoten *nach* dem Besuch seiner Nachfolger ausgegeben, also jeder Operator nach seinen Operanden (Abb. 7.2) .

Die **Prefix-Folge** (*preorder*) kann wie folgt aus dem Baum erzeugt werden: Bei einem Tiefendurchlauf wird jeder Knoten *vor* dem Besuch seiner Nachfolger ausgegeben, also jeder Operator vor seinen Operanden (Abb. 7.3) .

Die Prefix-Folge ist eine klammerfreie Form der Funktions-Schreibweise

```
/(*(a,b),+(-(c,d),e))
```

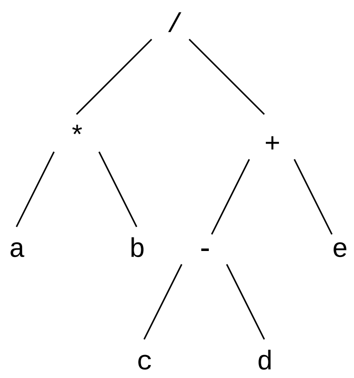
Die geklammerte Version kann man durch folgende Ergänzung leicht erzeugen: Bei jedem Innenknoten wird nach Ausgabe des Operators vor dem Besuch des ersten Nachfolgers eine öffnende Klammer, vor jedem weiteren Nachfolgerbesuch ein



Postfix-Folge:

a b * c d - e + /

Abbildung 7.2: Postfix-Folge zum Syntaxbaum



Prefix-Folge:

/ * a b + - c d e

Abbildung 7.3: Prefix-Folge zum Syntaxbaum

Beistrich und nach dem letzten Nachfolgerbesuch eine schließende Klammer ausgegeben.

7.3 Quadrupel

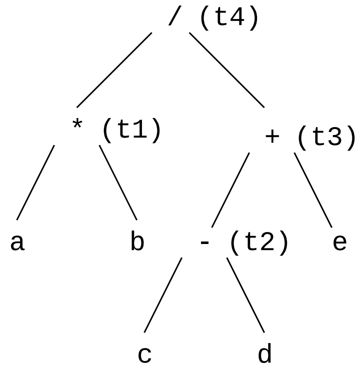
Ein Quadrupel besteht aus folgenden vier Elementen:

(Operator, 1.Operand, 2.Operand, Ergebnis).

Da Quadrupel drei Adressen enthalten, nennt man sie auch **Drei-Adreß-Code**.

Die Erzeugung aus dem Syntaxbaum erfolgt ähnlich wie bei der Postfix-Folge mit dem Unterschied, daß eindeutige symbolische Namen für die Operatorknoten generiert und ausgegeben werden (Abb. 7.4).

Zur besseren Lesbarkeit werden Quadrupel oft im Stil einer Programmiersprache als elementare Anweisungen geschrieben, also z.B.




```
Quadrupel-Folge:

( *, a , b , t1)
( -, c , d , t2)
( +, t2, e , t3)
( /, t1, t3, t4)
```

Abbildung 7.4: Quadrupel-Folge zum Syntaxbaum

`t1 = a + b` statt `(+,a,b,t1)`

Man beachte, daß Quadrupel im Unterschied zum Postfix- oder Prefix-Code nur maximal zweistellige Operatoren enthalten können. Der Vorteil von Quadrupeln liegt vor allem darin, daß sie einen symbolischen Namen für das Ergebnis enthalten und daher innerhalb einer Folge verschiebbar sind. Quadrupel-Code eignet sich daher für Optimierungen.

Wir verwenden folgenden Code (für die Quadrupel `(op,x,y,z)`):

`z = x`	`z` erhält den Wert von `x`
`z = op x`	`op` ist ein unärer Operator
`z = x op y`	`op` ist ein binärer Operator
`goto m`	unbedingter Sprung zur Marke `m`
`if x op y goto m`	Sprung zur Marke `m`, wenn `x op y TRUE` ist, `op` ist ein Vergleichsoperator
`z = @ x`	`z` erhält den Wert des Speicherplatzes, dessen Adresse in `x` steht
`@ z = x`	der Speicherplatz, dessen Adresse in `z` steht, erhält den Wert von `x`

Im folgenden werden wir mit Attributierten Grammatiken die Erzeugung von Quadrupeln für typische Konstrukte einer Programmiersprache beschreiben.

7.4 Zuweisungen

Die AG in Abb. 7.5 definiert die Übersetzung von Zuweisungen in Quadrupel. Der Operator `"+"` steht stellvertretend für binäre arithmetische Operatoren.

Die Funktion `lookadr(id.x)` liefert die Speicheradresse von `id` (aus der Symboltabelle). Die Funktion `newtemp` erzeugt eine neue Hilfsvariable `ti` und liefert deren

Produktion	Regeln
`S → V = E`	`S.c = E.c ‖ gen(V.a '=' E.a)`
$E \rightarrow E_1 + E_2$	`E.a = newtemp`
	$E.c = E_1.c \parallel E_2.c \parallel gen(E.a\ \text{'='}\ E_1.a\ \text{'+'}\ E_2.a)$
`E → V`	`E.a = V.a`
	`E.c = V.c`
`E → num`	`E.a = newconst(num.x)`
	`E.c = ''`
`V → id`	`V.a = lookadr(id.x)`
	`V.c = ''`

Abbildung 7.5: Erzeugung von Quadrupeln für die Zuweisungs-Anweisung

Adresse zurück. Die Funktion `newconst(num.x)` speichert den Wert der Konstanten `num` ab und liefert dessen Adresse zurück. Aus Gründen der leichteren Lesbarkeit ist im Quadrupelcode jedoch der Wert der Konstanten angegeben und nicht dessen Adresse. Die Adressen werden im Attribut `E.a` weitergegeben.

Die Folge der Quadrupel wird im Attribut `E.c` generiert. Der Operator "‖" verkettet Quadrupel bzw. Quadrupelfolgen miteinander. Die Funktion `gen` generiert ein einzelnes Quadrupel. Argument von `gen` ist eine Folge von Stringkonstanten und Stringvariablen. In dieser Folge ersetzt `gen` alle Variablen durch ihren Wert und liefert den gesamten String als Funktionswert. Bsp. 7.6 zeigt die für eine Zuweisung generierten Quadrupel.

```
t1  =  a + b
t2  =  t1 + c
 a  =  t2
```

Beispiel 7.6: Quadrupel für die Zuweisung `a=a+b+c`. Die Namen stehen hier für die (in der Symboltabelle eingetragenen) Speicheradressen.

7.5 Indizierte Variablen

Wir setzen voraus, daß die Untergrenze aller Indexbereiche `0` ist. Die Adresse einer indizierten Variablen $a[i_1, i_2, \ldots, i_k]$ wird durch folgende Formel berechnet.

$$((\ldots((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) w + adr(a)$$

mit

n_j	Länge (=Anzahl der Elemente) der j-ten Dimension
w	Anzahl der Speichereinheiten für ein Element
adr(a)	Anfangsadresse des Arrays a im Speicher

Die Längen n_j der Indexbereiche seien statisch deklariert – sie können in die Symboltabelle zu a eingetragen werden. Die Größe w ergibt sich aus dem Typ der Elemente und ist bei einem statischen Typsystem auch zur Übersetzungszeit bekannt. Bsp. 7.7 zeigt die für eine indizierte Variable generierten Quadrupel.

Formel: `(u*5+v)*4 + adr(a)`

Quadrupel:

```
t1 = 5  * u
t2 = t1 + v
t3 = 4  * t2
t4 = t3 + adr(a)
t5 = @ t4
 x = t5
```

Beispiel 7.7: Gegeben ist eine Modula-2-Deklaration `a:ARRAY[10,5] OF INTEGER` und eine Zuweisung an eine indizierte Variable `x =a[u,v]`. Ein integer-Element benötige 4 Bytes.

7.6 Boolesche Ausdrücke

Bei der Übersetzung Boolescher Ausdrücke ist zunächst die Frage zu klären, wie der Wert eines solchen Ausdrucks dargestellt werden soll. Es gibt zwei prinzipielle Möglichkeiten:

Numerische Methode: Die Werte werden durch Zahlen dargestellt, z.B. FALSE = 0 und TRUE ≠ 0. Diese Werte können durch Maschinenbefehle verarbeitet und bei der Zuweisung an eine Variable direkt verwendet werden.

Kontrollfluß-Methode: Die Werte werden durch Positionen im Code dargestellt. Jedem Booleschen Ausdruck werden zwei Marken zugeordnet, die als Sprungziele die beiden möglichen Werte des Ausdrucks repräsentieren.

Wir zeigen das Prinzip beider Methoden an einem einfachen Ausdruck in Abb. 7.8. Dabei bedeutet `*+n` die relative Adresse n im Quadrupel-Code. Der Operator '<' wird hier stellvertretend für alle Vergleichsoperatoren verwendet.

Bei der numerischen Methode wird ein Muster von Sprungbefehlen mit festen Relativadressen und einer temporären Variable verwendet, deren Wert (0 oder 1) als Wert des Ausdrucks weiterverarbeitet werden kann. Bei Verwendung einer AG wird

Numerische Methode	Kontrollfluß-Methode
`if a<b goto *+3`	`if a<b goto Ltrue`
`t1 = 0`	`goto Lfalse`
`goto *+2`	
`t1 = 1`	

Abbildung 7.8: Quadrupel-Darstellungen des Ausdrucks a<b

die Adresse der temporären Variable als synthetisiertes Attribut des Ausdrucks der Umgebung bekanntgegeben.

Die Kontrollfluß-Methode verwendet ein parametrisiertes Muster: `Ltrue` und `Lfalse` sind formale Wertparameter, denen symbolische Marken als aktuelle Sprungziele übergeben werden. Dem Ausdruck wird also „mitgeteilt“, wohin im Falle true bzw. false gesprungen werden soll. Aus der Sicht einer AG sind `Ltrue` und `Lfalse` zwei ererbte Attribute des Ausdrucks.

Die Übersetzung Boolescher Ausdrücke nach der Kontrollfluß-Methode wird durch die AG von Abb. 7.9 formal beschrieben. Die Funktion `newlabel` erzeugt – ähnlich wie die Funktion `newtemp` – eindeutige symbolische Namen. Das Grundmuster von Abb. 7.8 findet man als Regel zur Vergleichsoperation `rop` in Abb. 7.9 wieder. Man beachte, daß die Operatoren `and` und `or` im erzeugten Code nicht vorkommen. Für den Operator `not` wird überhaupt kein Code generiert, es werden nur die Sprungziele vertauscht.

Für die Operatoren `and` und `or` wird „jumping code“ generiert, d.h. die Ausdrücke werden nicht unbedingt zur Gänze ausgewertet. Bsp. 7.10 zeigt den Code für einen Booleschen Ausdruck mit dem Operator `and`. Falls der linke Operand `a<b` den Wert 'false' ergibt, wird das externe Sprungziel `L2` verwendet, d.h. der zweite Operand `c<d` wird nicht mehr ausgewertet.

Abb. 7.11 veranschaulicht, wie die Sprungziele als Attribute gemäß der AG von Abb. 7.9 übergeben werden, um den „jumping code“ zu erzeugen. Jedem Booleschen Ausdruck `B` sind die beiden ererbten Attribute `B.true` und `B.false` zugeordnet. Ihre Werte sind die beiden Sprungziele des Ausdrucks im Falle 'true' bzw. 'false'. Bei einem `and`-Operator wird das 'false'-Sprungziel direkt an das Attribut `B.false` des linken Teilausdrucks weitergegeben, während das Attribut `B.true` des linken Teilausdrucks eine neu generierte Marke als Sprungziel auf den zweiten Teilausdruck erhält. Ausdrücke mit `or` werden analog behandelt (siehe AG in Abb. 7.9).

7.7 Kontrollanweisungen

Die Übersetzung von Kontrollanweisungen hängt davon ab, ob der steuernde Ausdruck mit der numerischen oder der Kontrollfluß-Methode arbeitet. Wir werden daher beide Methoden berücksichtigen und jeweils als typische Vertreter von Kontrollanweisungen die IF- und die WHILE-Anweisung behandeln.

Produktion	Regeln
B $\rightarrow$ B_1 or B_2	B_1.true = B.true B_1.false = newlabel B_2.true = B.true B_2.false = B.false B.c = B_1.c $\|\|$ gen(B_1.false ':') $\|\|$ B_2.c
B $\rightarrow$ B_1 and B_2	B_1.true = newlabel B_1.false = B.false B_2.true = B.true B_2.false = B.false B.c = B_1.c $\|\|$ gen(B_1.true ':') $\|\|$ B_2.c
B $\rightarrow$ not B_1	B_1.true = B.false B_1.false = B.true B.c = B_1.c
B $\rightarrow$ E_1 rop E_2	B.c = E_1.c $\|\|$ E_2.c $\|\|$ gen('if' E_1.a rop.x E_2.a 'goto' B.true) $\|\|$ gen('goto' B.false)
B $\rightarrow$ true	B.c = gen('goto' B.true)
B $\rightarrow$ false	B.c = gen('goto' B.false)

Abbildung 7.9: Übersetzung Boolescher Ausdrücke nach der Kontrollfluß-Methode

```
     if a<b goto L3
     goto L2
L3:  if c<d goto L1
     goto L2
```

Beispiel 7.10: Quadrupel für den Booleschen Ausdruck `(a<b) and (c<d)`. Die externen Sprungziele für die Fälle 'true' und 'false' seien `L1` und `L2`.

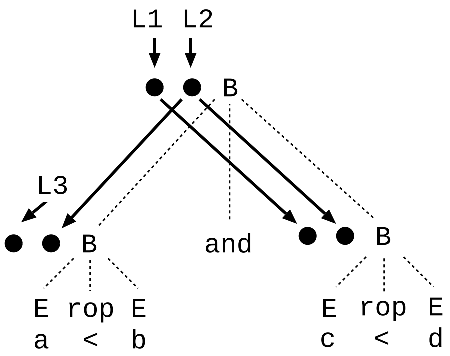


Abbildung 7.11: Syntaxbaum und Attributübergabe zu Bsp. 7.10

Für die **numerische Methode** sind die zu erzeugenden Sprungstrukturen und eine AG zur Erzeugung dieser Strukturen in Abb. 7.12 zusammengefaßt.

Für die **Kontrollfluß-Methode** sind die Sprungstrukturen und die AG in Abb. 7.13 zusammengefaßt.

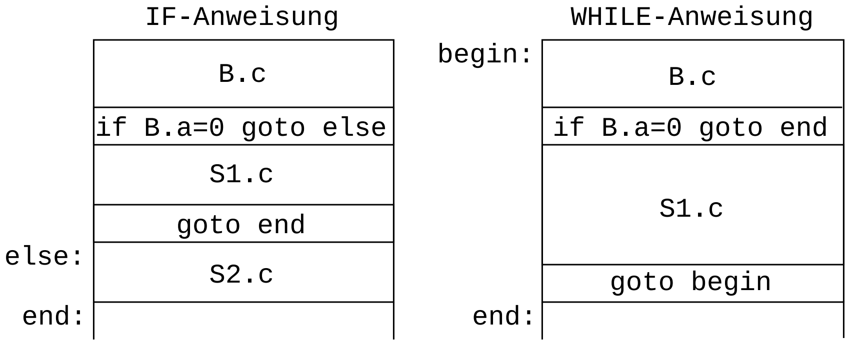
Bsp. 7.14 vergleicht die numerische und die Kontrollfluß-Methode bei der Übersetzung des Programmstücks zur Berechnung des `ggT`. Die Kontrollfluß-Methode ist für dieses Beispiel günstiger.

Der Code kann noch weiter vereinfacht werden (Nachoptimierung), indem man die beiden Vergleichsoperatoren umkehrt, d.h. $\neq$ durch $=$, und $>$ durch $\leq$ ersetzt. Bsp. 7.15 zeigt das optimierte Programmstück zur Kontrollflußmethode aus Bsp. 7.14. Sprünge zu einer Marke, an der wieder ein unbedingter Sprung steht, können durch diesen ersetzt werden, wie im Beispiel `goto L6` durch `goto L1`. Ein Sprung auf die dem Sprungbefehl unmittelbar folgende Stelle kann ebenfalls eliminiert werden.

7.8 Kontrollflußgraph und Datenflußgraph

Der **Kontrollflußgraph** beschreibt alle strukturell möglichen Abläufe eines Programms. Jeder Knoten stellt einen Grundblock (*basic block*) und jede Kante einen möglichen Kontrollübergang dar. Ein Grundblock ist eine Anweisungsfolge, die genau dann ausgeführt wird, wenn die erste Anweisung der Folge ausgeführt wird. Der Kontrollflußgraph kann aus dem Quadrupelcode eines Programms konstruiert werden. Abb. 7.16 zeigt den Graphen für das Programm `ggT` von Bsp. 7.14.

Jeder Grundblock kann vollständig durch einen **Datenflußgraphen** beschrieben werden. Die Knoten sind mit Operatoren markiert. Die Kanten repräsentieren Daten, die am Ausgangsknoten erzeugt und am Zielknoten verwendet werden. Knoten ohne Vorgänger sind als nullstellige Operatoren aufzufassen, d.h. jeder Knoten des Graphen erzeugt bzw. repräsentiert einen Wert. Datenflußgraphen lassen sich aus Quadrupelfolgen konstruieren, indem man die Teilgraphen der einzelnen Quadrupel aneinanderhängt. Abb. 7.17 zeigt den Datenflußgraphen für die Quadrupelfolge von



Produktion	Regeln

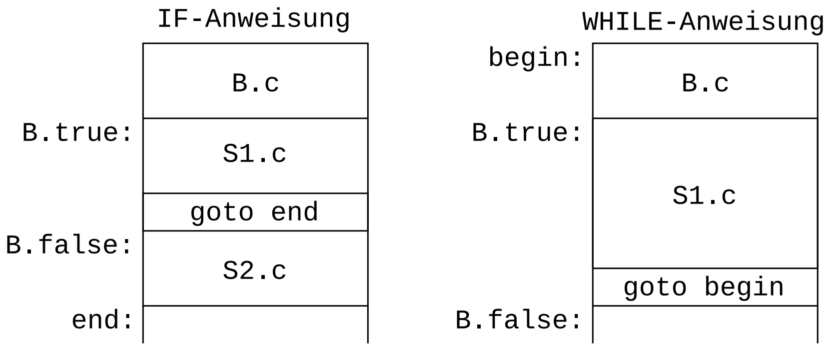
`S` $\rightarrow$ `if B then` S_1 `else` S_2	`else = newlabel`
	`end = newlabel`
	`S.c = B.c` $\|\|$
	`gen('if' B.a '=0 goto' else)` $\|\|$
	S_1`.c` $\|\|$ `gen('goto' end)` $\|\|$
	`gen(else ':')` $\|\|$ S_2`.c` $\|\|$
	`gen(end ':')`

`S` $\rightarrow$ `while B do` S_1	`begin = newlabel`
	`end = newlabel`
	`S.c = gen(begin ':')` $\|\|$ `B.c` $\|\|$
	`gen('if' B.a '=0 goto' end)` $\|\|$
	S_1`.c` $\|\|$ `gen('goto' begin)` $\|\|$
	`gen(end ':')`

Abbildung 7.12: Übersetzung von Kontrollanweisungen bei der numerischen Methode

Bsp. 7.7.

Insgesamt bietet der Quadrupelcode also die Möglichkeit, Programmdarstellungen zu erzeugen, bei denen einerseits der Kontrollfluß und andererseits der Datenfluß betrachtet wird. Diese Aufteilung ist zweckmäßig für Methoden der Optimierung und Codeerzeugung.



Produktion	Regeln
S $\rightarrow$ if B then S_1 else S_2	B.true = newlabel B.false = newlabel end = newlabel S.c = B.c $\|\|$ gen(B.true ':') $\|\|$ S_1.c $\|\|$ gen('goto' end) $\|\|$ gen(B.false ':') $\|\|$ S_2.c $\|\|$ gen(end ':')
S $\rightarrow$ while B do S_1	begin = newlabel B.true = newlabel B.false = newlabel S.c = gen(begin ':') $\|\|$ B.c $\|\|$ gen(B.true ':') $\|\|$ S_1.c $\|\|$ gen('goto' begin) $\|\|$ gen(B.false ':')

Abbildung 7.13: Übersetzung von Kontrollanweisungen bei der Kontrollfluß-Methode

```
WHILE x≠y DO
  IF x>y THEN
    x = x-y
  ELSE
    y = y-x
  END
END
```

Numerische Methode		Kontrollfluß-Methode	
L1:	`if x≠y goto *+3`	L1:	`if x≠y goto L2`
	`t1 = 0`		
	`goto *+2`		
	`t1 = 1`		
	`if t1=0 goto L2`		`goto L3`
	`if x>y goto *+3`	L2:	`if x>y goto L4`
	`t2 = 0`		
	`goto *+2`		
	`t2 = 1`		
	`if t2=0 goto L3`		`goto L5`
	`t3 = x-y`	L4:	`t1 = x-y`
	`x = t3`		`x = t1`
	`goto L4`		`goto L6`
L3:	`t4 = y-x`	L5:	`t2 = y-x`
	`y = t4`		`y = t2`
L4:	`goto L1`	L6:	`goto L1`
L2:		L3:	

Beispiel 7.14: Zum Vergleich werden die numerische und die Kontrollfluß-Methode zur Übersetzung eines Quellprogramms (`ggT`) angewendet.

```
L1:  if x=y goto L3
     if x≤y goto L5
     t1 = x-y
     x = t1
     goto L1
L5:  t2 = y-x
     y = t2
     goto L1
L3:
```

Beispiel 7.15: Durch Anwendung der Nachoptimierung auf den mit der Kontrollfluß-Methode entstandenen Code aus Bsp. 7.14 entsteht dieses Programm.

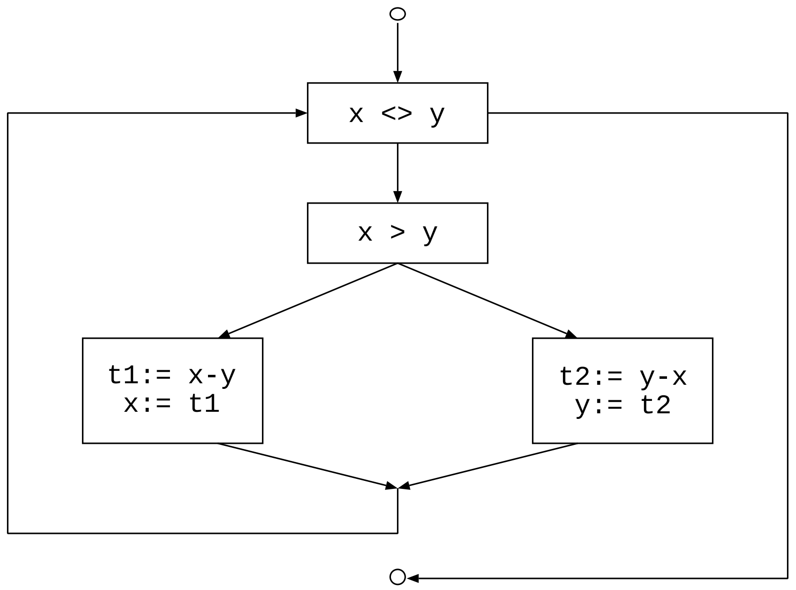


Abbildung 7.16: Kontrollflußgraph für das Programm `ggT` von Bsp. 7.14.

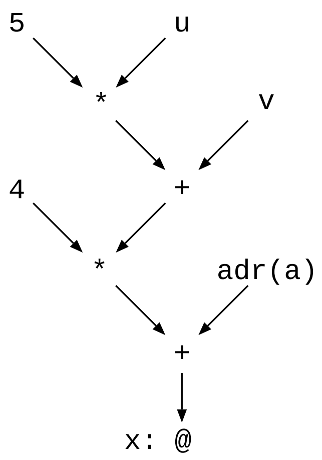


Abbildung 7.17: Datenflußgraph für die Quadrupelfolge von Bsp. 7.7.

Kapitel 8

Codeerzeugung

Die Codeerzeugung wandelt die maschinenunabhängige Zwischendarstellung in Assembler- oder Maschinencode um. Sie besteht aus drei Teilaufgaben:

Befehlsauswahl (*instruction selection*, *code selection*). Die Operationen der Zwischendarstellung werden zu Befehlen der Zielmaschine zusammengefaßt.

Befehlsanordnung (*instruction scheduling*). Die Befehle werden im Hinblick auf eine schnelle Ausführung auf Pipeline- oder superskalaren Prozessoren angeordnet.

Registerbelegung (*register allocation*). Die während früherer Phasen verwendeten Pseudoregister werden durch Maschinenregister ersetzt.

Die Codeerzeugung wird oft als maschinenabhängiger Teil des Compilers bezeichnet. Die im folgenden vorgestellten Methoden lassen sich jedoch auf eine große Klasse von Prozessorarchitekturen anwenden, zu der fast alle heute in PCs und größeren Computern verwendeten Prozessoren gehören. Die Methoden müssen nur mit den entsprechenden Prozessordaten parametrisiert werden. In diesem Kapitel zeigen wir die Probleme der Codeerzeugung anhand der Prozessorarchitekturen 68000 und MIPS.

Ein Befehl in 68000-Assembler hat folgende Syntax:

$$\textit{Name.Größe Operand1,Operand2}$$

Der Name des Befehls ist meist selbsterklärend, die Größe gibt an, auf welchen Daten der Befehl arbeitet. `b` steht für 1 byte, `w` für 2, und `l` für 4. Von den Operanden ist der rechte der Zieloperand, viele Befehle (z.B. `add`) verwenden ihn auch als Eingabeoperanden. Für die Operanden kann eine Vielfalt von Adressierungsarten verwendet werden, auf die wir hier nicht eingehen. Der 68000 hat zwei Klassen von Registern, Adreßregister (`An`) und Datenregister (`Dn`).

Befehle im MIPS-Assembler haben folgende Form:

$$\textit{Name Operand1,Operand2,Operand3}$$

Dabei ist im allgemeinen der linke Operand der Zieloperand, die anderen beiden die Eingabeoperanden. Die Befehe arbeiten auf Registern und mit Konstanten, Zugriffe auf den Speicher erfolgen über spezielle Lade- und Speicherbefehle der Form *Name Register, const(Register)*. Der zweite Operand gibt dabei die Adresse an.

8.1 Befehlsauswahl

Die Befehlsauswahl erfolgt auf Grundblockebene, wobei Grundblöcke als Datenflußgraphen dargestellt werden. Zur Vereinfachung betrachten wir zunächst aber nur Bäume[1], also Datenflußgraphen, bei denen jedes Datum nur einmal verwendet wird.

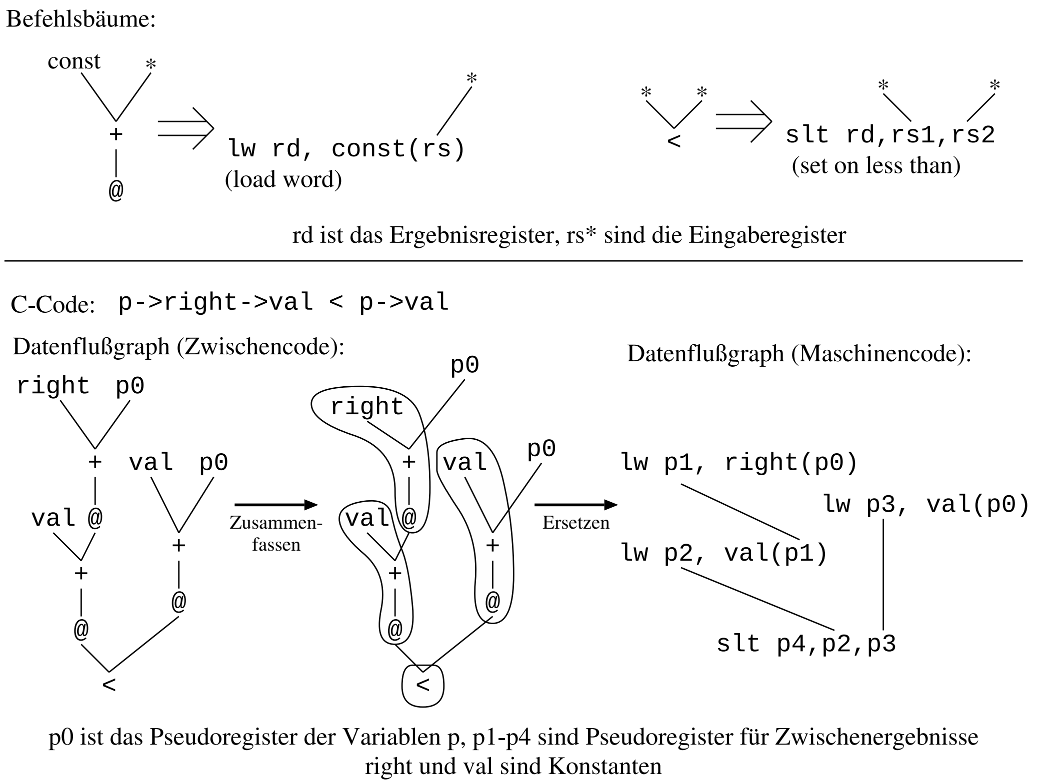


Abbildung 8.1: Befehlsauswahl für die MIPS-Architektur

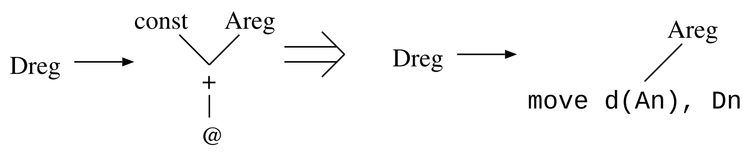
Die Knoten des Datenflußgraphen sind die Operatoren der Zwischensprache. Maschinenbefehle entsprechen Bäumen der Zwischensprache (Ausnahmen werden später besprochen). Die Aufgabe der Befehlsauswahl ist es, den Datenflußgraphen

[1]Datenflußgraphen werden üblicherweise so gezeichnet, daß die Daten von oben nach unten fließen. Bei Reduktion auf Bäume ergibt sich dabei die etwas ungewöhnliche Darstellung mit der Wurzel nach unten. Bei einer Expedition in den Resselpark können Sie sich überzeugen, daß diese Darstellung den Grundsätzen der Natur nicht widerspricht.

in Bäume aufzuteilen, die Maschinenbefehlen entsprechen. Die Bäume werden dann durch die entsprechenden Befehle ersetzt. Das Ergebnis ist ein Datenflußgraph, in dem die Knoten Maschinenbefehle sind. Abb. 8.1 zeigt beispielhaft die Befehlsauswahl für die MIPS-Architektur.

Im Baum eines Befehls steht dort ein $*$, wo der Baum eines weiteren Befehls eingesetzt werden kann. Allerdings können nicht immer alle Befehle so einfach kombiniert werden. So kann z.B. auf dem 68000 der Befehl „`move (An),Dn`“ das Ergebnis des Befehls „`and #const,Dn`“ nicht unmittelbar verwenden, da es in einem Datenregister liegt, aber in einem Adreßregister gebraucht wird. Beim 68000 müssen wir also zwei Arten von Sternen verwenden. Die eine (nennen wir sie Dreg) steht für Bäume, die ihr Ergebnis in einem Datenregister abliefern, die andere (Areg) steht entsprechend für Bäume mit dem Ergebnis in einem Adreßregister.

Die Beziehung zwischen dem Zwischencode und dem Maschinencode kann mit Regeln wie der folgenden beschrieben werden:



Die beiden Teile der Regeln schauen ähnlich wie Grammatikproduktionen aus. Man spricht daher von Baumgrammatiken, Areg und Dreg werden als Nonterminale bezeichnet. Die Beziehung zwischen Zwischencodebaum und Maschinencodebaum ist: Wenn ein Baum auf der einen Seite mit einer bestimmten Folge von Produktionen dieser Seite abgeleitet werden kann, dann erhält man durch die entsprechende Folge von Produktionen auf der anderen Seite einen äquivalenten Baum in der anderen Darstellung.

In der Praxis werden die Regeln etwas anders geschrieben:

$$\text{Dreg} \rightarrow @(+(\text{const},\text{Areg})) \quad \texttt{move const(An),Dn}$$

Bäume werden linearisiert als prefix-Ausdrücke dargestellt. Und die Produktion im Maschinenteil der Regel wird einfach durch den Befehl ersetzt. Die Produktion kann folgendermaßen rekonstruiert werden: Die linke Seite ist das gleiche Nonterminal wie im Zwischendarstellungsteil und die rechte Seite enthält ebenfalls dieselben Nonterminale wie die rechte Seite des Zwischendarstellungsteils; diese Nonterminale werden als Kinder direkt an den Maschinenbefehl gehängt.

Die Einführung verschiedener Nonterminale anstatt des $*$ hat einen weiteren Vorteil: Prozessoren wie der 68000 können die Befehle mit vielen Adressierungsarten kombinieren. Anstatt jede Kombination von Befehl und Adressierungsart hinzuschreiben, kann man eine Regel für eine Klasse von Adressierungsarten bilden, und sie in allen entsprechenden Befehlen verwenden.

Baumgrammatiken für die Befehlsauswahl müssen jeden Zwischencode-Baum beschreiben können. Dabei stört es nicht, wenn die Grammatik mehrdeutig ist; Die verschiedenen Ableitungen für den Baum entsprechen den verschiedenen Möglichkeiten, den Baum in Maschinensprache zu übersetzen.

Von diesen Möglichkeiten sollte eine optimale bezüglich Ausführungszeit und/oder Speicherverbrauch ausgewählt werden. Daher fügen wir zu jeder Regel noch eine Zahl, die die Kosten des entsprechenden Maschinenbefehls beschreibt, also die Ausführungszeit oder den Platzbedarf des Befehls. Der im folgenden beschriebene Algorithmus liefert dann ein optimales Ergebnis, wenn die Gesamtkosten sich aus der Summe der Kosten für die einzelnen Regeln ergeben. Das ist bei heutigen Prozessoren nicht immer der Fall, da Befehle überlappt ausgeführt werden können. Trotzdem ist die Annahme eines additiven Verhaltens eine gute Annäherung.

```
Für jeden Knoten n im Baum, von den Blättern ausgehend:
  t ist der Teilbaum, dessen Wurzel n ist
  Für jedes Nonterminal A:
    cost[n][A]=∞
    Für jede Regel R der Form „A→ Baummuster“ (ausgenommen
        Kettenregeln), sodaß Baummuster auf t paßt:
      cost1=R.cost
      tree1.op=n.op
      childno=1
      Für jedes Nonterminal B in Baummuster:
        u ist der Knoten an der Stelle in t,
          die B in Baummuster einnimmt
        cost1=cost1+cost[u][B]
        tree1.child[childno]=tree[u][B]
        childno=childno+1
      Wenn cost1<cost[n][A] dann
        cost[n][A]=cost1
        tree[n][A]=tree1

  Wende Kettenregeln an wie oben die anderen bis sich nichts mehr ändert
```

Algorithmus 8.2: Befehlsauswahl-Algorithmus

Der Befehlsauswahl-Algorithmus (siehe Alg. 8.2) geht von den Blättern aus zur Wurzel vor und ermittelt für jeden Knoten n und jedes Nonterminal A eine Regel R mit folgenden Eigenschaften: Aus A kann mit der Regel der Teilbaum abgeleitet werden, dessen Wurzel n ist; und diese Ableitung hat minimale Kosten. Diese Berechnung greift auf die optimale Befehlsauswahl und die zugehörigen Kosten für untergeordnete Baum-Nonterminal-Kombinationen zurück.

Konkret geschieht folgendes: Die minimalen Kosten für eine Baum-Nonterminal-Kombination bei Verwendung einer bestimmten Regel an der Wurzel berechnen sich als Summe der Kosten der Regel und der Kosten der von der Regel verwendeten Teilbaum-Nonterminal-Kombinationen. Die Kosten aller Teilbäume sind schon bekannt, da sich der Algorithmus von den Blättern zur Wurzel vorarbeitet. Zu jedem

Knoten werden für jedes Nonterminal die Gesamtkosten und der Maschinencodebaum mit den niedrigsten Gesamtkosten gespeichert.

Ein Problem stellen dabei die Kettenregeln dar, also Regeln der Form *Nonterminal1* $\rightarrow$ *Nonterminal2*. Bei der Berechnung von Kosten mit Hilfe einer Kettenregel würde auf die Kosten eines Nonterminals desselben Teilbaumes zurückgegriffen, die eventuell noch nicht oder nicht vollständig berechnet sind. Eine einfache Lösung ist es, Kettenregeln nach allen anderen zu berücksichtigen, und, wenn sich die Kosten des Baums ändern, noch einmal alle Kettenregeln zu berücksichtigen, solange bis sich nichts mehr ändert. Diese Wiederholung der Kettenregelberechnung dient für den Fall, daß eine Kettenregel von einer anderen abhängt.

Nr.	Regel	Befehl	Kosten
1	Dreg $\rightarrow$ Areg	`move.l An, Dn`	4
2	Areg $\rightarrow$ Dreg	`move.l Dn, An`	4
3	Areg $\rightarrow$ @(Areg)	`move.l (An),An`	12
4	Dreg $\rightarrow$ @(Areg)	`move.l (An),Dn`	12
5	Dreg $\rightarrow$ +(Areg,Dreg)	`add.l An,Dn`	8
6	Dreg $\rightarrow$ +(Dreg,Dreg)	`add.l Dn,Dn`	8
7	Areg $\rightarrow$ +(Dreg,Areg)	`add.l Dn,An`	8
8	Areg $\rightarrow$ +(Areg,Areg)	`add.l An,An`	8
9	Dreg $\rightarrow$ +(@(Areg),Dreg)	`add.l (An),Dn`	14
10	Areg $\rightarrow$ +(@(Areg),Areg)	`add.l (An),An`	14
11	Dreg $\rightarrow$ $@_b$(Areg)	`move.b (An),Dn`	8
12	Dreg $\rightarrow$ $@_b$(+(Areg,Dreg))	`move.b 0(An,Dn),Dn`	14
13	Dreg $\rightarrow$ $@_b$(+(Areg,Areg))	`move.b 0(An,An),Dn`	14
14	Areg $\rightarrow$ `an`	ε	0
15	Dreg $\rightarrow$ `dn`	ε	0

Abbildung 8.3: Teil einer Befehlsauswahl-Baumgrammatik für den 68000

Abb. 8.3 zeigt einige Regeln, die sich im Beispiel in Abb. 8.4 anwenden lassen. `an` ist ein Terminalsymbol, das für ein Adreßregister steht, entsprechend steht `dn` für ein Datenregister. Die Kosten einer Regel sind die Laufzeit des Befehls in Maschinenzyklen. Der Zwischencode-Baum in Abb. 8.4 entspricht dem C-Ausdruck `(*p)[x]`, wobei `p` auf einen Zeiger auf ein Zeichen-Array zeigt und in einem Adreßregister liegt; `x` liegt in einem Datenregister. Das Ergebnis soll in einem Datenregister landen. Bei jedem Knoten sind für jedes Nonterminal die Kosten der optimalen Befehlsauswahl für den entsprechenden Unterbaum eingetragen. „tree“ enthält einen optimalen Maschinencodebaum für den entsprechenden Teilbaum und das Nonterminal, der allerdings aus Platzgründen linearisiert und mit Regelnummern statt der Maschinenbefehle dargestellt ist.

Betrachten wir nun, wie die Berechnung für den Wurzelknoten vor sich geht: Das Nonterminal Areg kann den Baum nur über die Kettenregel 2 ableiten; wir

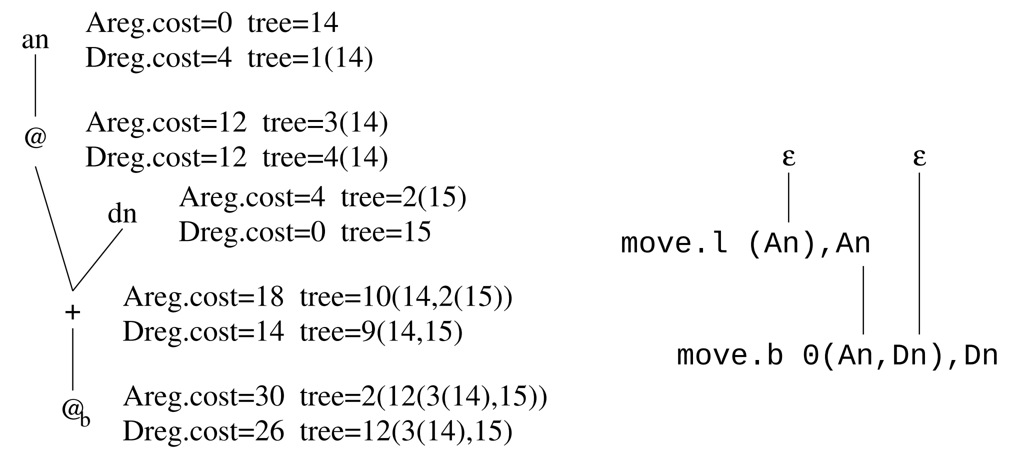


Abbildung 8.4: Optimale Befehlsauswahl

betrachten diese Variante daher später. Für Dreg kommen die Regeln 11, 12, und 13 in Frage. Die Berechnung der Kosten für die Regel 12 verläuft folgendermaßen: Die Regel verwendet den @-Knoten des Baums als Areg (Kosten: 12) und den dn-Knoten des Baums als Dreg (Kosten: 0), die Regel selbst kostet 14 Zyklen, die Gesamtkosten sind daher 26 Zyklen. Für Regel 11 ergeben sich ebenfalls Kosten von 26 Zyklen, für Regel 13 dagegen 30 Zyklen. Die Regeln 11 und 12 sind daher optimal. Wir wählen willkürlich Regel 12. Der zugehörige Maschinencodebaum ergibt sich wie folgt: Der Operator ist der zur Regel gehörige Befehl `move.b 0(An,Dn),Dn`. Der Baum hat für jedes Nonterminal der rechten Seite der Zwischencodeproduktion ein Kind, und zwar für Areg den optimalen Maschinencodebaum für Areg am @-Knoten: `move.l (An),An`(ε) und für Dreg den optimalen Maschinencodebaum für Dreg am Knoten dn: ε. Insgesamt ergibt sich also der in Abb. 8.4 rechts gezeigte Baum.

Nun können die Kettenregeln angewandt werden, z.B. die Regel 2, um die optimalen Kosten für das Nonterminal Areg des Wurzelknotens zu berechnen: Sie ergeben sich aus den 26 Zyklen von Dreg im Wurzelknoten und 4 Zyklen für die Regel selbst, also insgesamt 30 Zyklen. Der Maschinencodebaum hat als Operator den Befehl `move.l Dn, An` und nur ein Kind: den optimalen Maschinencodebaum für Dreg am Knoten $@_b$, den wir gerade vorher ermittelt haben.

Jetzt kann auch die Regel 1 für die Berechnung von Dreg angewandt werden, dadurch ergibt sich aber natürlich keine Verbesserung (Kosten: 34); die Regeln 1 und 2 fügen ja nur Kopierbefehle ein, deren wiederholte Anwendung das Programm wohl nicht schneller macht.

Die Eingabe der Befehlsauswahl sind meist nicht Bäume, sondern azyklische Graphen (DAGs). Leider gibt es für DAGs im allgemeinen kein ähnlich einfaches, optimales und effizientes Verfahren wie das oben beschriebene für Bäume. Ein Lösungsweg ist, den DAG in Bäume aufzuteilen, indem alle Kanten entfernt werden, die zwischen einem Knoten mit mehreren Eltern und diesen Eltern verlaufen. Der Code für die Teilbäume ist optimal, der Code für den gesamten DAG nicht unbedingt, aber oft.

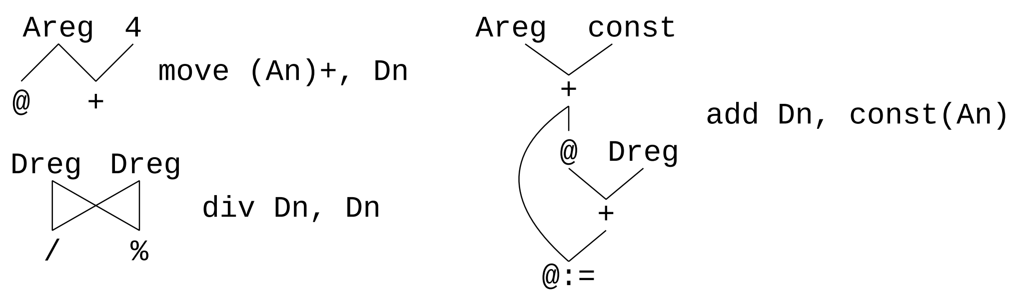


Abbildung 8.5: Befehle, deren Operatordarstellung kein Baum ist

Ein weiteres Problem stellen Befehle dar, deren Operatordarstellung kein Baum ist (siehe Abb. 8.5). Dazu zählen Befehle mit mehreren Resultaten wie z.B. Divisionsbefehle, die auch noch den Rest liefern, und Befehle mit der Autoincrement-Adressierungsart, aber auch viele Befehle, die auf Speicheroperanden arbeiten (read-modify-write-Befehle) . Solche Befehle können nicht mit Verfahren erzeugt werden, die auf Baummustererkennung beruhen. Die einfachste Lösung ist, sich damit zufriedenzugeben, daß der Compiler diese Befehle nicht verwendet. Eine andere Möglichkeit ist, nachträglich geeignete Befehlskombinationen zusammenzufassen, wobei diese Befehle entstehen.

8.2 Befehlsanordnung

Die Umformung des Datenflußgraphen eines Grundblocks in eine lineare Form, wie sie der Prozessor erwartet, ist Aufgabe der Befehlsanordnung. Bei Prozessoren, die durch Pipelining und/oder superskalare Architektur Befehle parallel abarbeiten, ist das Ziel der Befehlsanordnung eine gute Ausnutzung des Parallelismus durch Minimierung von ungenutzten Wartezyklen (*delay slots*, *pipeline bubbles*).

Auf dem MIPS R4000-Prozessor kann zum Beispiel einen Zyklus nach dem Start eines Ladebefehls ein anderer Befehl ausgeführt werden. Das Ergebnis des Ladebefehls steht aber erst nach drei Zyklen zur Verfügung. Wird ein Befehl gestartet, der das Ergebnis verwendet, dann muß er so lange warten und es entstehen ungenutzte Wartezyklen. Wenn die Befehle so angeordnet werden, daß nach einem Ladebefehl zwei Befehle folgen, die das Ergebnis nicht verwenden, wird der Prozessor optimal ausgenutzt. Auch bei superskalaren Prozessoren wie dem 586 tritt das Problem auf: Nur unabhängige Befehle können im gleichen Zyklus ausgeführt werden. Ein verwandtes Problem gibt es auch bei Verzweigungsbefehlen in einigen RISC-Architekturen. Bei der MIPS-Architektur wird ein Verzweigungsbefehl erst mit Verspätung ausgeführt, vorher wird noch der unmittelbar hinter dem Verzweigungsbefehl liegende Befehl ausgeführt (*delayed branch*). Dieser Branch-delay-slot sollte natürlich mit einem sinnvollen und unabhängigen Befehl gefüllt werden.

Zunächst stellt sich die Frage, welche Befehlsanordnungen überhaupt korrekt sind. Ein Befehl a muß in folgenden Fällen vor einem anderen Befehl b ausgeführt werden:

Datenflußabhängigkeit (*flow dependence, RAW dependence*): Wenn b von einem Register oder Speicherplatz liest, der von a beschrieben wurde.

Lese-Schreib-Abhängigkeit (*anti dependence, WAR dependence*): Wenn b auf eine Stelle schreibt, die zuvor von a gelesen wurde.

Schreib-Schreib-Abhängigkeit (*output dependence, WAW dependence*): Wenn b auf eine Stelle schreibt, die zuvor von a beschrieben wurde.

potentielle Abhängigkeit Wenn nicht sicher ist, daß zwei Zugriffe auf verschiedene Stellen erfolgen, z.B. bei Zugriffen über Zeiger, muß die Annahme getroffen werden, daß eine Abhängigkeit zwischen den zwei Zugriffen besteht.

Die Abhängigkeiten zwischen den Befehlen bilden einen Datenabhängigkeitsgraphen (*data dependence graph*), das ist ein um Lese-Schreib-, Schreib-Schreib-, und potentielle Abhängigkeitskanten erweiterter Datenflußgraph. Jede Befehlsanordnung, die die Abhängigkeiten berücksichtigt, ist korrekt.[2] Die Abb. 8.6 zeigt den Abhängigkeitsgraphen für eine Schleife in MIPS-Assembler; Integerregister werden in der Form `$`n geschrieben, Fließkommaregister in der Form `$f`n. `$32` is ein Label.

Wenn es um die beste Lösung geht, ist das Problem der Befehlsanordnung im allgemeinen NP-vollständig, d.h. alle bekannten Algorithmen weisen exponentielles Laufzeitverhalten auf. Daher werden in der Praxis heuristische Algorithmen verwendet, die in vertretbarer Zeit gute, wenn auch nicht immer optimale Befehlsanordnungen finden.

Der Standard-Algorithmus für Befehlsanordnung heißt *List Scheduling* und ist in Alg. 8.7 dargestellt: Einer der Befehle ohne Vorgänger im Graphen (ein sogenannter Anführer) wird ausgewählt und aus dem Graphen entfernt. Dieser Schritt wird wiederholt bis der Graph leer ist. Die Reihenfolge, in der die Befehle entfernt wurden, ist die Befehlsanordnung.

Dieser Algorithmus läßt offen, welcher Befehl ausgewählt wird. Dies ist die Aufgabe der heuristischen Auswahlfunktion „wähle“. Eine typische Auswahlfunktion zur Ausnutzung des Parallelismus lautet:

- Wähle einen Befehl, der als nächstes ausgeführt werden kann; also einen, der möglichst kurz oder am besten gar nicht warten muß, bis andere Befehle seine Eingabedaten berechnet haben. Wenn es mehrere Kandidaten gibt,

- Wähle einen Befehl mit maximaler Pfadlänge zum Ende des Datenabhängigkeitsgraphen. Zur Berechnung der Pfadlänge geben wir jeder Kante eine Länge: Die Länge einer Abhängigkeitskante ist die Zeit, die nach dem Start des Befehls am Anfang der Kante vergehen muß, bis der Befehl am Ende der Kante ausgeführt werden kann. Diese Heuristik sorgt dafür, daß Delay-slots früh freigelegt werden, solange sie noch gefüllt werden können.

```
for (i=0; i<n; i++) {
   y[iy] = y[iy]+a*x[ix];
   ix += incx;
   iy += incy;
}

   $32:
A  mul    $14, $4, 8
B  addu   $6, $5, $14
C  mul    $15, $7, 8
D  addu   $24, $15, $8
E  l.d    $f4, 0($24)
F  mul.d  $f6, $f4, $f12
G  l.d    $f8, 0($6)
H  add.d  $f10, $f6, $f8
I  s.d    $f10, 0($6)
J  addu   $7, $7, $9
K  addu   $4, $4, $10
L  addu   $2, $2, 1
M  bne    $11, $2, $32
```

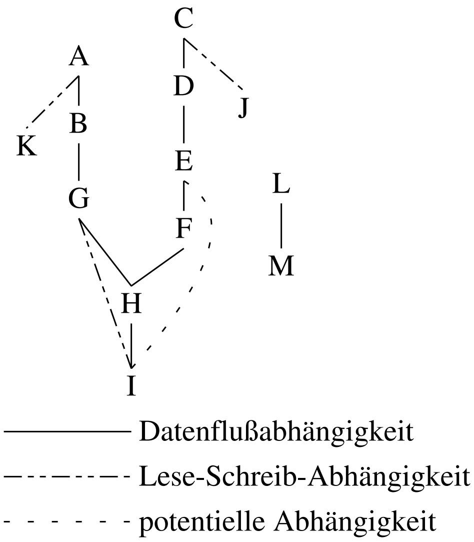
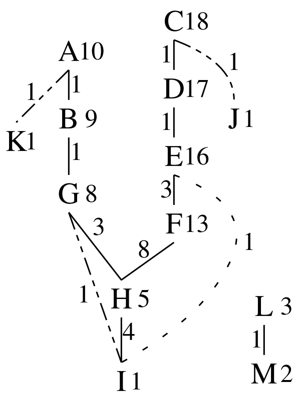


Abbildung 8.6: Eine Schleife in C und MIPS-Assembler und ihr Datenabhängigkeitsgraph

list_scheduling(Graph)
  Anordnung=leer
  while Graph ≠ leer
    Anführer={Knoten im Graph ohne Vorgänger}
    Nächster-Befehl=wähle(Anführer)
    Hänge Nächster-Befehl an Anordnung an
    Entferne Nächster-Befehl aus Graph

Algorithmus 8.7: List Scheduling




```
  $32:
C mul    $15, $7, 8
D addu   $24, $15, $8
E l.d    $f4, 0($24)
A mul    $14, $4, 8
B addu   $6, $5, $14
F mul.d  $f6, $f4, $f12
G l.d    $f8, 0($6)
L addu   $2, $2, 1
K addu   $4, $4, $10
J addu   $7, $7, $9
H add.d  $f10, $f6, $f8
M bne    $11, $2, $32
I s.d    $f10, 0($6)
```

Abbildung 8.8: Der Datenabhängigkeitsgraph mit Kanten- und Pfadlängen und der Code nach der Befehlsanordnung

Abb. 8.8 zeigt den Datenabhängigkeitsgraphen dekoriert mit Kantenlängen für den R4000. Weiters ist bei jedem Knoten die Pfadlänge bis zum Ende dargestellt. Für den Verzweigungsbefehl M ist dabei eine Pfadlänge von zwei Zyklen bis zum Ende angegeben, da es von der Ausführung des Verzweigungsbefehls bis zum Ende des Grundblocks eben zwei Zyklen dauert (*delayed branch*).

Am Anfang der Befehlsanordnung sind A, C, und L die Anführer und sie könnten alle sofort ausgeführt werden. Aufgrund der Pfadlänge wird C gewählt. Dadurch werden D und J zu Anführern, die ebenfalls sofort ausgeführt werden könnten. Als nächstes werden die Befehle D und E ausgewählt. Dann gibt es mit F erstmals einen Anführer, der nicht sofort ausgeführt werden kann. Aber noch gibt es genügend andere Kandidaten. Im weiteren werden A, B, F, G und L ausgewählt. Jetzt wird M zu einem Anführer, aber als Verzweigungsbefehl darf er erst als vorletzter oder letzter Befehl gewählt werden. Daher lautet die weitere Folge K, J (oder J, K) und H. H muß zwar noch drei Zyklen auf das Ergebnis von F warten, aber H ist der einzige wählbare Anführer und wird daher ausgewählt. Nun kann endlich M gewählt werden, und in den Delay-slot von M kommt der Befehl I, der weitere zwei Zyklen auf das Ergebnis von H warten muß. Der Grundblock benötigt bei dieser Befehlsanordnung 18 Zyklen, bei der ursprünglichen 25. Die Befehlsanordnung ist sogar optimal: Da der kritische Pfad des Datenabhängigkeitsgraphen 18 Zyklen lang ist, gibt es sicher keine kürzere.

[2]Graphentheoretiker nennen solche Anordnungen eines Graphen „topologisch sortiert“.

8.3 Registerbelegung

Moderne Compiler erzeugen zunächst Code, der beliebig viele Pseudoregister verwendet. Die Aufgabe der Registerbelegung ist es, die Pseudoregister möglichst alle in Maschinenregistern unterzubringen, bzw. wenn das nicht gelingt, weniger wichtige Pseudoregister in den Speicher auszulagern. Im Gegensatz zur Befehlsauswahl und der Befehlsanordnung, die auf der Grundblockebene („lokal") arbeiten, erfolgt die Registerbelegung heute für eine ganze Prozedur („global" im irreführenden Compilerbau-Jargon).

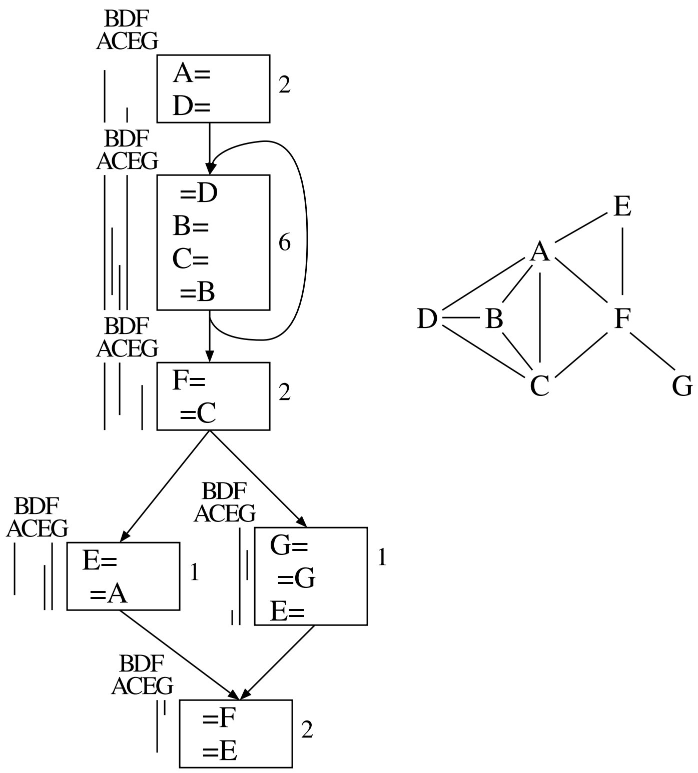


Abbildung 8.9: Ein Programm, die Aktivitätsbereiche seiner Pseudoregister, und sein Konfliktgraph

Ausgangspunkt der Registerbelegungsalgorithmen ist die Erkenntnis, daß zwei Pseudoregister im selben Maschinenregister untergebracht werden können, wenn sie nicht zur gleichen Zeit aktiv (*live*) sind. Daraus ergibt sich die grundlegende Datenstruktur für die Registerbelegung: Der Konfliktgraph (*interference graph*, *conflict graph*) enthält für jedes Pseudoregister einen Knoten. Zwei Knoten sind durch eine Kante verbunden, wenn die beiden Pseudoregister zur gleichen Zeit aktiv sind. Abb. 8.9 zeigt ein Programm, und zwar die für die Registerbelegung wichtigen Teile, nämlich die Zuweisungen ($x =$) an und die Verwendungen von ($= x$) Pseudoregistern und den Kontrollfluß. Weiters werden die Aktivitätsbereiche der Pseudoregister (links von den Grundblöcken), die erwartete Ausführungshäufigkeit der Grundblöcke (rechts) und der Konfliktgraph dargestellt.

Das Problem der Registerbelegung läßt sich umformulieren in das Problem, den Knoten Maschinenregister zuzuordnen, sodaß zwei Knoten, die durch eine Kante verbunden sind, verschiedene Register bekommen. Dieses Problem[3] ist im Operations Research als Graphenfärben (*graph colouring*) bekannt und wie das Problem der Befehlsanordnung NP-vollständig, daher können wir die optimale Lösung nicht in vertretbarer Zeit finden. Wir greifen wieder auf heuristische Algorithmen zurück, um eine gute Lösung zu finden.

Das Grundprinzip der im Compilerbau verwendeten Algorithmen ist einfach: Nimm einen Knoten und weise ihm ein Register zu, das von den Nachbarn des Knotens noch nicht verwendet wird; wiederhole diesen Schritt für alle Knoten. Folgende Fragen bleiben offen:

Was passiert, wenn für einen Knoten kein Register mehr übrig ist?

Eine weit verbreitete Methode ist es, das entsprechende Pseudoregister in den Speicher auszulagern (*spill*). Viele Maschinenbefehle arbeiten jedoch nur mit Registern und nicht mit Operanden im Speicher. Die Pseudoregister müssen daher kurzfristig in einem Maschinenregister gehalten werden, in der Nähe jener Maschinenbefehle, die das Pseudoregister verwenden. Das Auslagern ersetzt also ein langlebiges Pseudoregister durch viele kurzlebige.

In der Praxis werden Knoten, für die kein Register übrig ist, zunächst einmal ignoriert. Nach dem Ende der Registerzuweisung wird der Auslagerungscode (*spill code*) eingefügt, der Konfliktgraph neu aufgebaut und erneut versucht, eine Registerzuweisung zu finden.

In welcher Reihenfolge werden die Knoten abgearbeitet?

Die Reihenfolge der Abarbeitung bestimmt, welche Pseudoregister ein Maschinenregister bekommen und welche nicht. Sie ist daher sehr wichtig für die Qualität der Registerbelegung.

Pseudoreg	Kosten
A	4
B	18
C	10
D	14
E	6
F	6
G	3

Abbildung 8.10: Die Auslagerungskosten der Pseudoregister aus Abb. 8.9, wenn ein Speicherbefehl einen Zyklus und ein Ladebefehl zwei Zyklen kostet

[3] Genauer gesagt, das Problem, die minimale Anzahl von Registern zu finden, für die es eine solche Zuordnung gibt.

Ein wichtiges Kriterium ist, wieviel an Laufzeit[4] es kosten würde, das Pseudoregister auszulagern (siehe Abb. 8.10). Ein weiterer wichtiger Faktor ist der Grad des Knotens, also die Anzahl der Kanten, die von ihm ausgehen: Ein hochgradiger Knoten hat viele Nachbarn, denen er ein Register wegnehmen kann und sollte daher eher ausgelagert werden als ein niedergradiger Knoten. Diese beiden Kriterien können z.B. folgendermaßen zu einer Prioritätsfunktion $p(n)$ kombiniert werden:

$$p(n) = \frac{Kosten(n)}{Grad(n)}$$

Die Knoten könnten nun einfach in der Reihenfolge absteigender Priorität abgearbeitet werden. Es gibt aber noch eine bessere Variante: Wenn ein Knoten weniger Nachbarn (also einen kleineren Grad) hat als k, die Anzahl der zur Verfügung stehenden Register, dann bleibt auf jeden Fall ein Register für den Knoten übrig, auch wenn zuerst alle seine Nachbarn ein Register bekommen. Wir entfernen daher zunächst einmal alle Knoten n mit $Grad(n) < k$ aus unserer Betrachtung und aus dem Konfliktgraphen. Dadurch reduziert sich eventuell der Grad weiterer Knoten unter die k-Grenze. Diese erhalten ebenfalls garantiert ein Register, allerdings unter der Bedingung, daß die früher entfernten Knoten erst nach diesen Knoten ihr Register erhalten. Auf diese Weise kann der Graph solange reduziert werden, bis der Graph leer ist oder nur mehr Knoten mit $Grad(n) \geq k$ übrigbleiben. Ist der Konfliktgraph leer, dann kann eine Registerbelegung gefunden werden, indem die Knoten in der umgekehrten Reihenfolge des Entfernens abgearbeitet werden.

Bleiben noch Knoten übrig, so wird zunächst die Prioritätsfunktion berechnet und der Knoten mit der niedrigsten Priorität entfernt, damit er dann bei der Abarbeitung so spät wie möglich drankommt. Ergeben sich dadurch wieder Knoten mit $Grad(n) < k$, werden die zuerst entfernt. Anhand der Prioritätsfunktion wird nun der nächste Knoten bestimmt usw., bis der Graph leer ist. Auch hier wird die Registerbelegung in der umgekehrten Reihenfolge der Entfernung aus dem Graphen vorgenommen. In diesem Fall ist der Erfolg aber nicht garantiert und es müssen unter Umständen Knoten ausgelagert werden.

Welches der verfügbaren Register wird einem Knoten zugewiesen?

Bei Knoten mit $Grad(n) \geq k$ hängt es von den Registern, die seine Nachbarn zugewiesen bekommen haben, ab, ob ein Register für den Knoten übrigbleibt oder nicht. Man könnte annehmen, daß dadurch die Qualität der Registerbelegung stark beeinflußt wird. Versuche mit ausgeklügelten Registerzuweisungsverfahren haben aber enttäuschende Resultate erbracht. In der Praxis wird daher das erstbeste Register verwendet.

Beispiel

Wir wollen eine Registerbelegung für das Programm in Abb. 8.9 mit drei Registern finden. Der erste Schritt ist, die Knoten zu ordnen. E und G können sofort aus dem

[4] Normalerweise muß man sich hier mit Abschätzungen behelfen, da zur Übersetzungszeit nicht bekannt ist, wie oft die entsprechenden Programmteile zur Laufzeit ausgeführt werden.

Graphen entfernt werden, da sie weniger als drei Nachbarn haben. Dadurch hat F auch weniger als drei Nachbarn und kann ebenfalls entfernt werden. Damit bleiben A, B, C und D mit jeweils drei Nachbarn übrig. Wir entfernen A, da seine Priorität mit 4/3 am niedrigsten ist. Dann haben B, C, und D nur mehr zwei Nachbarn und können in beliebiger Reihenfolge entfernt werden. Die Reihenfolge der Registerzuweisung ist nun umgekehrt: B, C, D, A, F, E, G. Wir weisen nun B das Register 1 zu, was für A, C und D nur mehr die Register 2 und 3 läßt. Nach der Zuweisung von 2 an C und 3 an D ist für A kein Register mehr übrig. Daher setzen wir mit F fort (verfügbar: 1, 3) und geben ihm Register 1. E und G erhalten beide das Register 2.

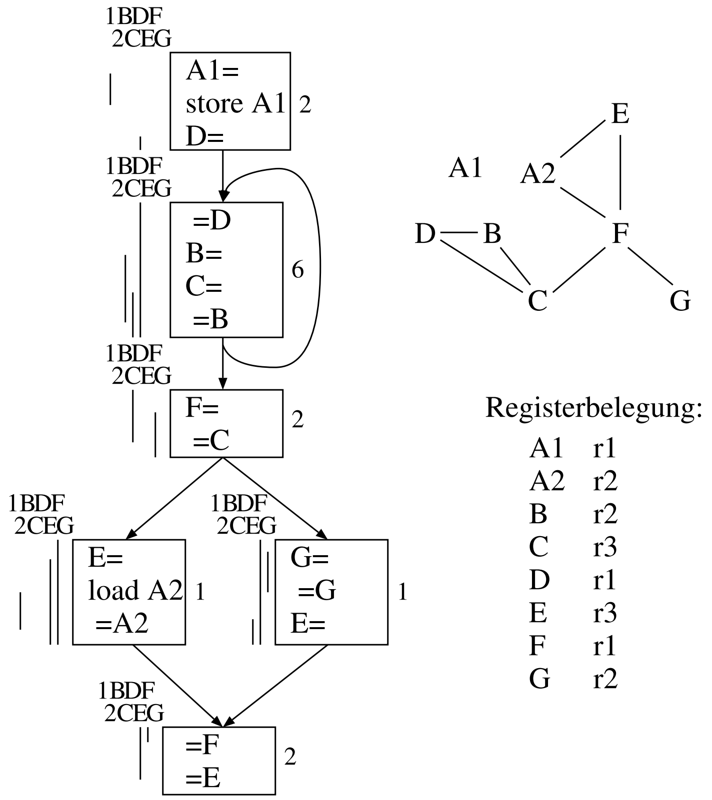


Abbildung 8.11: Auslagerung des Pseudoregisters A, der neue Konfliktgraph, und eine Registerbelegung

Da für das Pseudoregister A kein Maschinenregister gefunden wurde, wird A ausgelagert. Dies erfolgt durch Einfügen eines Speicherbefehls nach der Zuweisung an A und eines Ladebefehls vor der Verwendung von A. Die beiden Aktivitätsbereiche von A werden auf zwei Pseudoregister A1 bzw. A2 aufgeteilt, da sie nicht dasselbe Register erhalten müssen. Dadurch ergibt sich ein neuer Konfliktgraph (siehe Abb. 8.11). Diesmal können bei der Bestimmung der Reihenfolge alle Knoten entfernt werden, ohne einen Knoten mit drei Nachbarn entfernen zu müssen. Daher gibt es natürlich auch eine komplette Registerbelegung.

Entfernung von Kopierbefehlen

Der Konfliktgraph enthält auch alle Informationen, um Kopierbefehle von einem Register in ein anderes wegzuoptimieren. Ein Kopierbefehl von einem Pseudoregister in ein anderes kann wegoptimiert werden, wenn es keinen Konflikt zwischen den beiden Pseudoregistern gibt. Dann werden die beiden Knoten im Konfliktgraphen einfach zu einem verschmolzen (*coalesced*). Dabei werden natürlich auch die Kanten der beiden Knoten vereinigt. Diese Optimierungsphase wird unmittelbar nach dem Aufbau des Konfliktgraphen ausgeführt, noch bevor der erste Versuch der Registerzuteilung unternommen wird.

Es stellt sich natürlich die Frage, ob es genug Kopierbefehle gibt, daß sich das überhaupt auszahlt. Nun, viele Optimierungen lassen sich einfacher formulieren und programmieren, wenn bedenkenlos Kopierbefehle verwendet werden können. Diejenigen davon, die unnötig sind, werden später wieder wegoptimiert.

Damit sich diese Optimierung in mehr Fällen anwenden läßt, kann noch das Konfliktkonzept verfeinert werden: Zwei Pseudoregister können nicht dasselbe Register verwenden, wenn sie zur gleichen Zeit aktiv sind und (während dieser Zeit) verschiedene Werte enthalten. Das heißt, hinter einem Kopierbefehl sind zwei Pseudoregister zunächst einmal nicht in Konflikt, auch wenn sie beide aktiv sind.

Zwei-Adreß-Maschinen

Während der Befehlsauswahl und der Registerbelegung werden alle Befehle so behandelt, als wären die Operanden der Befehle unabhängig voneinander. Doch in sogenannten Zwei-Adreß-Maschinen (z.B. x86, 68000) muß bei den meisten Befehlen das Ergebnis im gleichen Register stehen wie eine Eingabe. Die bisher verwendeten Drei-Adreß-Befehle können folgendermaßen in Zwei-Adreßbefehle umgewandelt werden: Aus

```
r1 = r2+r3
```

wird

```
r1  =  r2
r1  =  r1+r3
```

Diese Transformation wird zweckmäßigerweise vor dem Entfernen der Kopierbefehle durchgeführt.

Kapitel 9

Laufzeitsystem

Das Laufzeitsystem eines Compilers umfaßt alle Komponenten, die zum Ablauf eines erzeugten Programms notwendig sind (und vom Betriebssystem nicht bereitgestellt werden). Das Laufzeitsystem muß einerseits auf den Compiler und andererseits auf das Betriebssystem abgestimmt werden. Die Konstruktion von Laufzeitsystemen ist ein wesentlicher Teil des Compilerbaus. Dieser Bereich wird derzeit von Compilergeneratoren nicht unterstützt.

Eine der wichtigsten Komponenten des Laufzeitsystems ist die Verwaltung des Speichers für Variablen, Prozeduren und Parameter. Wir behandeln in diesem Kapitel die zwei grundlegenden Methoden der **Speicherverwaltung**: Die **Stack**- und die **Heap**-Verwaltung. Die Speicherverwaltung hängt von den Eigenschaften der Quellsprache ab. Compiler und Speicherverwaltung müssen daher in einem „integrierten Entwurf" aufeinander abgestimmt werden.

Ein weiterer wichtiger Teil des Laufzeitsystems sind die Ein/Ausgabe-Prozeduren. Diese Prozeduren basieren auf dem verwendeten Betriebssystem. In den meisten Programmiersprachen werden Ein/Ausgabe-Prozeduren wie gewöhnliche Prozeduren behandelt. Daher ist eine besondere Abstimmung auf den Compiler nicht notwendig.

Das Laufzeitsystem ist auch dafür zuständig, daß das Objektprogramm in seinem Ablauf überwacht wird und daß im Fehlerfall Meldungen gegeben werden, die für den Benutzer verständlich sind. In der Praxis trifft man häufig auf Systeme, die in dieser Hinsicht unzulänglich sind.

Ein optimales Fehlerverhalten zur Laufzeit erreicht man durch verstärkten Informationsaustausch zwischen Compiler und Laufzeitsystem. Der Compiler muß dem Laufzeitsystem Informationen verfügbar machen, die dieses für den normalen Ablauf nicht benötigen würde. Dazu zählen z.B. die Bezeichner von Variablen und die Indexgrenzen von Feldern. Es ist daher zweckmäßig, die Symboltabelle an das Laufzeitsystem zu übergeben. Dann kann im Fehlerfall ein „post-mortem dump" ausgegeben werden, der die aktuellen Werte der Variablen (mit ihren Bezeichnern) enthält.

9.1 Speicherorganisation

Das von einem Compiler erzeugte Maschinenprogramm (Objektprogramm) besteht zur Laufzeit aus folgenden Komponenten:

- Code (Befehle)
- Statische Daten
- Dynamische Daten

Der Code wird vom Compiler generiert. Speicherbedarf und Adressierung der statischen Daten können ebenfalls vom Compiler bestimmt werden. Die Adressen der dynamischen Daten werden erst zur Laufzeit des Objektprogramms bekannt. Daher muß der Speicher für diese Daten vom Laufzeitsystem verwaltet werden. Abb. 9.1 zeigt eine typische Speicheraufteilung zur Laufzeit.

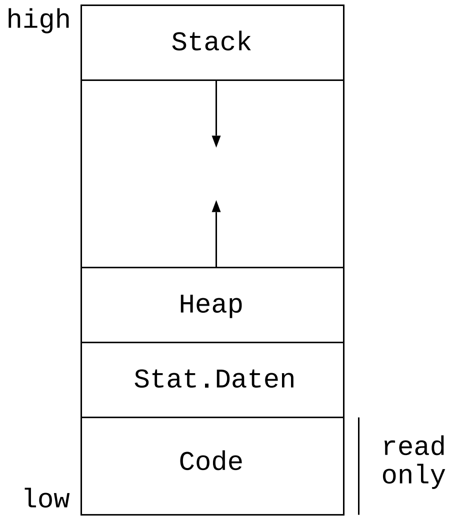


Abbildung 9.1: Speicheraufteilung zur Laufzeit

Man unterscheidet zwei Methoden zur Verwaltung der dynamischen Daten:

Die **Stack-Verwaltung** wird verwendet, um die Datenbereiche von Prozeduren bei geschachtelten und rekursiven Aufrufen zu verwalten. Die Speicherzuteilung ist streng an jeden Prozeduraufruf gekoppelt: Der Datenbereich wird beim Aufruf zugeteilt und bei der Rückkehr wieder freigegeben.

Die **Heap-Verwaltung** ist notwendig, wenn Speicher zu beliebigen Zeitpunkten angefordert und freigegeben werden soll.

Meistens benötigt man sowohl Stack- als auch Heap-Verwaltung. Es ist üblich, für beide einen gemeinsamen Speicherbereich zu reservieren und diesen wie in Abb. 9.1 nach Bedarf dynamisch in Stack und Heap aufzuteilen.

9.2 Stack-Verwaltung

Rekursive Prozeduren benötigen bei jedem Aufruf einen eigenen Speicherbereich, der als **Activation Record** (AR) oder **Frame** bezeichnet wird (Abb. 9.2) .

Parameter
stat.Vorgänger Rücksprungadr. dyn.Vorgänger
lokale Daten

Abbildung 9.2: Activation Record einer Prozedur

Die **Parameter** werden von der aufrufenden Prozedur im AR der aufgerufenen Prozedur abgelegt. Bei einer Funktion wird im Parameterbereich ein Platz für den Funktionswert reserviert – falls die Rückgabe nicht über ein Register erfolgt. Der **statische Vorgänger** (*static link*, *access link*) ist ein Zeiger auf den AR der statisch umschließenden Prozedur. Er wird nur bei Programmiersprachen mit geschachtelten Prozedurdeklarationen benötigt (siehe Kap. 9.3). Die **Rücksprungadresse** gibt an, wo die aufrufende Prozedur nach Beendigung der aufgerufenen Prozedur fortgesetzt werden soll. Der **dynamische Vorgänger** (*dynamic link*, *control link*) ist ein Zeiger auf den AR der aufrufenden Prozedur. Er wird verwendet, um nach Beendigung der aufgerufenen Prozedur den Stack zurückzusetzen. Die **lokalen Daten** sind die Werte der lokal deklarierten Variablen der Prozedur. Dazu kommen unter Umständen noch Zwischenergebnisse für Ausdrücke.

Die Stack-Verwaltung beruht darauf, daß beim Aufruf einer Prozedur der zugehörige AR angelegt und beim Verlassen der Prozedur wieder freigegeben wird. Zur Verwaltung des Stack verwenden wir zwei Register (Abb. 9.3):

- `S` (Stack-Zeiger) zeigt auf die Spitze des Stack.
- `A` (AR-Zeiger) zeigt auf den aktuellen, zuletzt erzeugten AR.

Alle Komponenten im AR der aktuellen Prozedur können mit `r(A)` indiziert adressiert werden, wobei die Abstände `r` zur Übersetzungszeit aus dem Typ und dem Speicherbedarf der Komponenten berechnet werden. Dadurch ist auch der gesamte Speicherbedarf `d` der lokalen Daten jeder Prozedur bekannt.

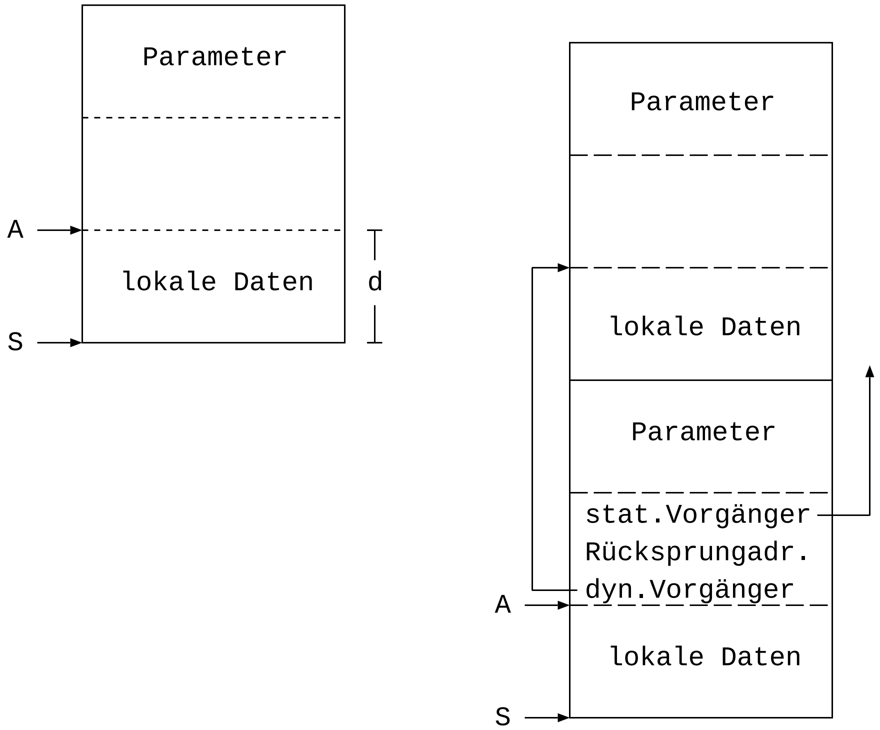


Abbildung 9.3: Prozeduraufruf

Aufrufende Prozedur:	*Aufgerufene Prozedur:*
aktuelle Parameter berechnen und kellern	
stat. Vorgänger berechnen und kellern	
Unterprogrammaufruf	Rücksprungadresse kellern
	dyn. Vorgänger (`A`) kellern
	`A=S`
	`S=S+d`
	. . . Prozedurrumpf . . .
	`S=S-d`
	`A=`dyn. Vorgänger
	dyn. Vorgänger entkellern
	Rücksprungadresse entkellern
	Rücksprung
stat. Vorgänger entkellern	
aktuelle Parameter entkellern	

Abbildung 9.4: Sequenz beim Prozeduraufruf.

Aufruf-Sequenz (calling sequence)

Die Operationen „kellern“ und „entkellern“ in Abb. 9.4 beinhalten entsprechende Veränderungen des Stack-Zeigers `S`. Im allgemeinen muß auch der Inhalt von Registern beim Prozedur-Aufruf gesichert werden. Sie werden ebenfalls im AR abgespeichert, und zwar entweder von der aufrufenden oder von der aufgerufenen Prozedur.

Falls eine Prozedur mit variabler Parameteranzahl aufgerufen werden soll, muß auch diese Anzahl bekannt sein, damit die aufgerufene Prozedur die Parameterliste abarbeiten kann. Da `A` auf den Anfang der lokalen Daten und nicht auf den Anfang des AR zeigt, bleiben die Distanzadressen aller anderen Komponenten unabhängig von der Anzahl der Parameter konstant.

Dynamische Arrays werden im Bereich der lokalen Daten jeweils durch einen Zeiger repräsentiert und im Anschluß an den aktuellen AR auf den Stack gelegt.

Abb. 9.5 zeigt den Stack für eine Prozedur mit zwei dynamischen Arrays `X` und `Y`. Man sieht, daß die Anfangsadresse des Array `Y` von der Größe des Array `X` abhängt und daher vom Compiler nicht als Relativadresse bestimmt werden kann.

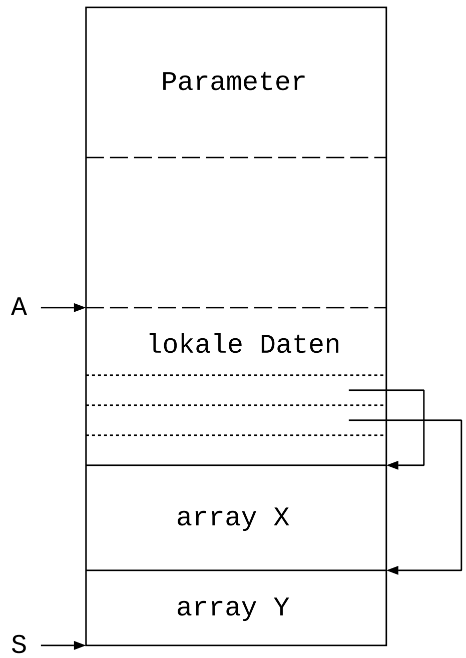


Abbildung 9.5: Dynamische Arrays im Stack

Bei einer Stack-Verwaltung ohne dynamische Arrays kann man den AR-Zeiger einsparen und nur mit dem Stack-Zeiger S adressieren (z.B. MIPS Compiler). Bei „Blatt-Prozeduren“ (*leaf procedures*), die selbst keine Prozeduren aufrufen, kann man das Anlegen eines AR im Stack überhaupt einsparen.

9.3 Nicht-lokale Variablen

Die lokalen Variablen der aktuell aufgerufenen Prozedur werden mit `A` als Basis adressiert. Bei Programmiersprachen mit geschachtelten Prozedurdeklarationen müssen auch die jeweils gültigen nicht-lokalen Variablen adressiert werden.

Adressierung mit statischen Ketten

Zu diesem Zweck wird in jedem AR ein Zeiger auf den **statischen Vorgänger**, d.h. auf den AR der umschließend deklarierten Prozedur, gespeichert. Ausgehend von `A` gibt dann die Kette der statischen Vorgänger (**statische Kette**) die auf jeder Schachtelungstiefe adressierbaren Variablen an. Der Anfangszeiger der statischen Kette (hier `A`) wird auch Umgebungszeiger (*environment pointer*) genannt.

Abb. 9.6 zeigt ein Programm mit geschachtelten Prozedurdeklarationen: Die Schachtelung ist durch Einrückungen dargestellt: Wenn ein Prozedurbezeichner `X` die Tiefe `n` hat, so haben die in `X` deklarierten Objekte und Anweisungen die Tiefe `n+1`. Prozeduraufrufe sind zur Verdeutlichung mit einem Pfeil gekennzeichnet.

Jede Deklaration und jede Referenz besitzt eine eindeutige Schachtelungstiefe. Wenn man `P` die Tiefe `n=0` zuordnet, so hat die Deklaration von `Q` die Tiefe `n=1` und der Aufruf (Referenz) →`Q` die Tiefe `n=2`. Die Differenz der Schachtelungstiefen zwischen Referenz und Deklaration bezeichnen wir mit `dn`. Diesen Wert kann der Compiler für jede Referenz bestimmen.

Angenommen, in jedem AR des Stack sei der statische Vorgänger eingetragen und eine Prozedur `X` soll aufgerufen werden: Dann findet man den statischen Vorgänger von `X`, indem man die statische Kette von `A` ausgehend `dn`-mal dereferenziert.

Als Beispiel betrachten wir Abb. 9.6. Der Stack wird mit `P` initialisiert (genauer: mit dem AR für `P`). `A` zeigt zunächst auf `P`. Dann ist `T` aufzurufen und der statische Vorgänger zu bestimmen. Aufruf und Deklaration von `T` haben die gleiche Tiefe, also ist `dn=0`. Daher zeigt `A` selbst (als Anfang der statischen Kette) auf den statischen Vorgänger, hier auf `P`. `A` wird in `T` als statischer Vorgänger eingetragen. Danach wird `A` auf den AR von `T` gesetzt.

Als nächstes muß `Q` aufgerufen und der statische Vorgänger bestimmt werden, wobei `dn=1` ist. Die statische Kette wird ausgehend von `A` (jetzt auf `T` zeigend) einmal dereferenziert und liefert wiederum einen Zeiger auf `P`. Der Zeiger wird in `Q` eingetragen und `A` auf `Q` gesetzt.

Schließlich wird `R` mit `dn=0` aufgerufen. `A` (jetzt auf `Q` zeigend) zeigt wieder direkt auf den statischen Vorgänger.

Abb. 9.6 zeigt den Inhalt des Stacks, wenn im Programm die Stelle (*) erreicht ist. Ausgehend von `A` (auf `R` zeigend) werden über die statische Kette alle momentan adressierbaren Variablen erreicht. Den statischen Gültigkeitsregeln entsprechend sind dies `b` in `Q` und `a` in `P`. Der AR von `T` wird von der statischen Kette „umgangen“.

Um nicht-lokale Variablen zu adressieren, wird wiederum die Differenz `dn` der Schachtelungstiefen zwischen Referenz und Deklaration verwendet. Der Compiler generiert für jeden Zugriff eine `dn`-malige Dereferenzierung der statischen Kette (siehe Bsp. 9.7).

```
PROGRAM P
  a:REAL
  PROCEDURE Q
    b:REAL
    PROCEDURE R
      c:REAL
      (*)
    END R
    →R
  END Q
  PROCEDURE T
    b:REAL
    →Q
  END T
  →T
END P
```

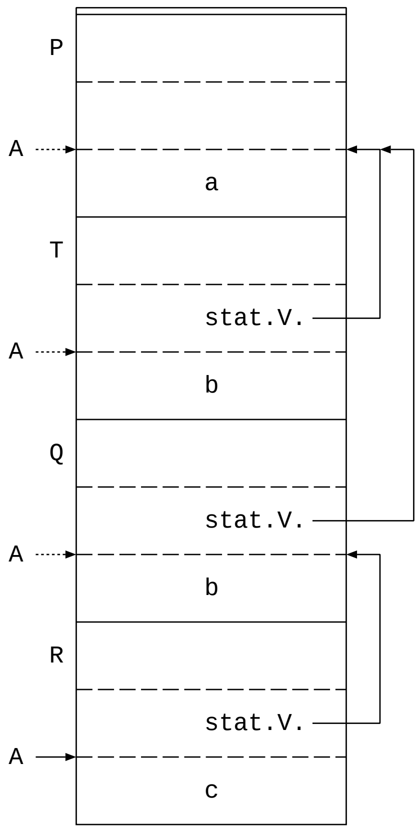


Abbildung 9.6: Modula-2-Programm und Stack mit statischen Ketten

Nicht-lokale Variable `a`:	`move sv(A),R`
	`move sv(R),R`
	$\mathtt{r}_a$`(R)`
Nicht-lokale Variable `b`:	`move sv(A),R`
	$\mathtt{r}_b$`(R)`
Lokale Variable `c`:	$\mathtt{r}_c$`(A)`

Beispiel 9.7: Adressierung lokaler und nicht-lokaler Variablen. Angenommen, an der Position (*) im Programm von Abb. 9.6 werden die Variablen `a,b` und `c` verwendet. Die Relativadressen seien $\mathtt{r}_a$, $\mathtt{r}_b$ und $\mathtt{r}_c$. Die Relativadresse der Speicherstelle des statischen Vorgängers sei `sv`.

Adressierung mittels Display

Der Zugriff auf eine nicht-lokale Variable mittels statischer Kette wird umso aufwendiger, je größer `dn` ist. Dies kann durch ein **Display** vermieden werden. Ein Display ist ein Block von Indexregistern `D1`...`Dm`, in dem eine Kopie der statischen Kette verwaltet wird (Abb. 9.8). Die statische Kette enthält zu jedem Zeitpunkt für jede Schachtelungstiefe genau einen Zeiger. Daher kann jeder Schachtelungstiefe ein Indexregister zugeordnet werden, das diesen Zeiger aufnimmt. Die Adressierung von Variablen erfolgt dann ausschließlich aufgrund der Schachtelungstiefe `n` ihrer *Deklaration* über das Indexregister `Di` mit `i=n` (vgl. Bsp. 9.7):

Nicht-lokale Variable `a`:	$\mathtt{r}_a$`(D1)`
Nicht-lokale Variable `b`:	$\mathtt{r}_b$`(D2)`
Lokale Variable `c`:	$\mathtt{r}_c$`(D3)`

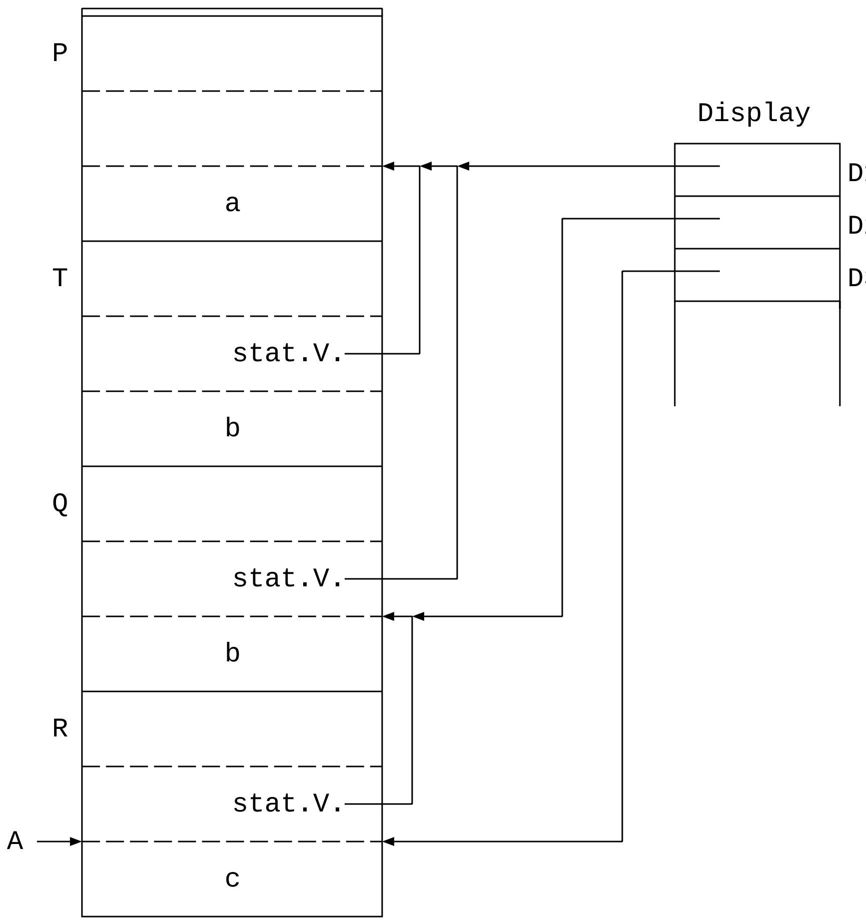


Abbildung 9.8: Stack mit Display für das Programm von Abb. 9.6

Die einfachste Display-Organisation besteht darin, bei jedem Prozeduraufruf alle Zeiger der statischen Kette in das Display zu kopieren (Abb. 9.8). Durch Unterscheiden der Aufrufsituationen läßt sich der Aufwand aber reduzieren. Man sieht z.B., daß beim Aufruf von `Q` nur `D2`, aber nicht `D1` geändert werden muß. Beim Aufruf von `R` ist nur `D3` zu laden.

Während bei der statischen Kette die Rückkehr aus einer Prozedur keinen Aufwand verursacht, muß ein Display unter Umständen auch bei der Rückkehr reorganisiert werden, und zwar immer dann, wenn eine Prozedur mit `dn>0` aufgerufen wurde. Dies gilt z.B. für jede rekursive Prozedur.

Zusammenfassend ist festzustellen, daß ein Display die Adressierung von Variablen vereinfacht, aber einen höheren Aufwand bei Aufruf und Rückkehr der Prozeduren bedingt.

9.4 Parameterübergabe

Der Bereich am Beginn eines AR ist für die Parameter reserviert. Dieser Bereich liegt im Stack unmittelbar anschließend an den AR der aufrufenden Prozedur und kann daher von der aufrufenden Prozedur gefüllt werden. Die Parameterübergabe besteht generell darin, die Werte der aktuellen Parameter zu berechnen und in den Parameterbereich zu übergeben. In diesem Sinne ist jede Parameterübergabe eine Wert-Übergabe. Allerdings hängt die Art des berechneten und übergebenen Wertes folgendermaßen von der Art des Parameters ab:

Wert-Parameter Der Wert ist das Ergebnis der Auswertung des aktuellen Parameters. Ein formaler Wert-Parameter kann wie eine lokale Variable adressiert werden.

Variablen-Parameter Der Wert ist die absolute Adresse des aktuellen Parameters. Ein formaler Variablen-Parameter muß indirekt adressiert werden.

Prozedur-Parameter Der Wert ist das Tupel <`P`, statischer Vorgänger>. `P` repräsentiert die Startadresse. Sie wird i.a. bei der Übersetzung festgelegt. Der statische Vorgänger wird bei der Übergabe der Prozedur bestimmt, so als ob die Prozedur an der Übergabestelle aufgerufen würde. Beim Aufruf der formalen Prozedur muß der statische Vorgänger nicht mehr ermittelt werden, da er übergeben wurde.

In Abb. 9.9 ist ein Programm mit einem Prozedur-Parameter dargestellt sowie der Zustand des Stacks, wenn der Ablauf die Stelle (*) erreicht hat.

Für alle Parameterarten gilt folgendes: Wenn ein formaler Parameter selbst wieder als aktueller Parameter verwendet wird, so kann der Wert des formalen Parameters direkt verwendet, d.h. kopiert werden.

9.5 Dynamische Objekte

In vielen prozeduralen Sprachen gibt es dynamische Objekte (Variablen), die zu einem beliebigen Zeitpunkt angelegt (z.B. mit `new`) und wieder freigegeben werden können (explizit oder implizit). Der Speicherbereich für diese Objekte kann nicht in

```
PROGRAM P
  a:REAL
  PROCEDURE T(X)
    b:REAL
    →X
  END T
  PROCEDURE Q
    b:REAL
    PROCEDURE R
      c:REAL
      (*)
    END R
    →T(R)
  END Q
  →Q
END P
```

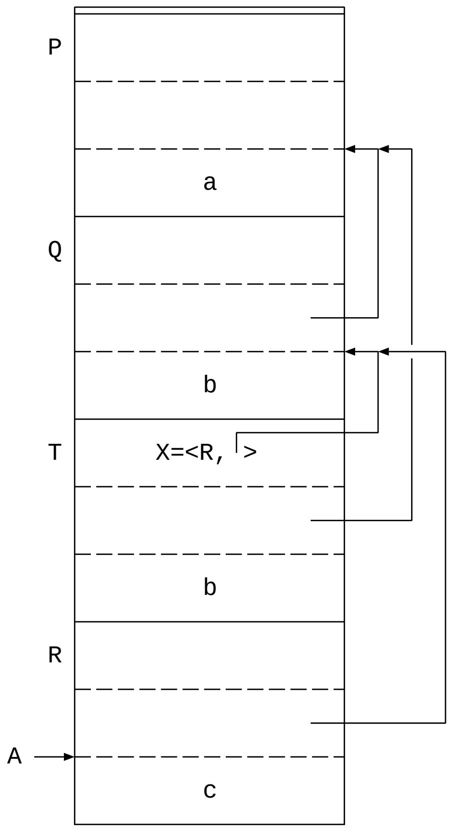


Abbildung 9.9: Modula-2-Programm und Stack mit Prozedur-Parameter

einem Stack verwaltet werden, sondern man benötigt dazu die allgemeinere Organisation eines Heap (siehe Kapitel 9.6).

In objektorientierten Sprachen treten dynamische Objekte als Instanzen von Klassen auf. Im Unterschied zu prozeduralen Sprachen enthalten diese Objekte nicht nur Werte und Referenzen von Variablen, sondern auch Referenzen auf die zugeordneten Methoden.

Abb. 9.10 zeigt die grundlegende Struktur von Objekten einer objektorientierten Sprache mit einfacher Vererbung. Die Unterklasse `B` erbt von der Oberklasse `A` die Variable `a` und die Methode `f`. Die Methode `g` wird in `B` überschrieben. An die deklarierte Variable `v` vom Typ `A` können Objekte der Klasse `A` oder `B` zugewiesen werden. Beim Aufruf `v.g()` bestimmt die Klassenzugehörigkeit des zugewiesenen Objekts, ob die Methode `g` von `A` oder `B` aufgerufen wird. Der Compiler kann die Methode statisch nicht bestimmen, da im allgemeinen Zuweisungen vom Ablauf des Programms abhängen. Um die Methode zur Laufzeit auswählen zu können, muß das Objekt selbst Informationen über seine Klassenzugehörigkeit bzw. die zugehörigen Methoden beinhalten. Daher enthält jedes Objekt einen Zeiger auf den Klassen-Deskriptor, in dem die Adressen der Methoden der Klasse stehen. Methoden gleichen Namens haben in den Deskriptoren die gleiche relative Position (z.B. `A_g` und `B_g`),

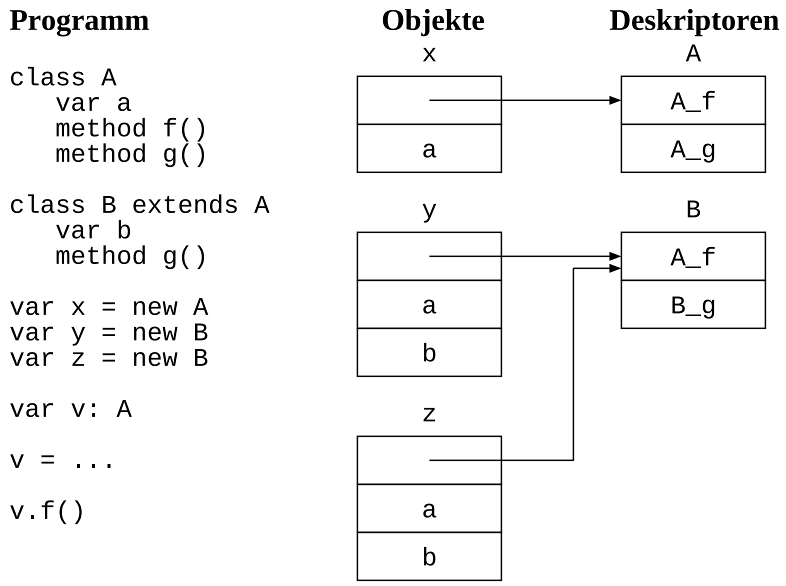


Abbildung 9.10: Grundlegende Struktur von Objekten einer objektorientierten Sprache mit einfacher Vererbung.

die statisch vom Compiler bestimmt werden kann.

In diesem grundlegenden Implementierungsmodell erfordert der Methodenaufruf gegenüber einem gewöhnlichen Prozeduraufruf zwei zusätzliche Speicherzugriffe (Adresse des Deskriptors, Adresse der Methode).

9.6 Heap-Verwaltung

In einem Heap werden Speicherobjekte (Zellen) in beliebiger Folge belegt und freigegeben. Heaps sind für Programmiersprachen ebenso relevant wie für Betriebs- bzw. Datenbanksysteme (z.B. Fileverwaltung).

Das Laufzeitsystem muß für jede Zelle folgende Operationen bereitstellen (Abb. 9.11):

Allokierung Vor Verwendung einer Zelle muß Speicher für sie allokiert und ggf. initialisiert werden.

Manipulation Während ihrer Lebensdauer müssen Referenzen auf die Zelle manipuliert werden können.

Deallokierung Nach der letzten Verwendung einer Zelle kann sie deallokiert werden – der Speicher ist für eine neue Zelle wiederverwendbar.

Die Wahl einer geeigneten Heap-Verwaltung wird durch Anforderungen der Anwendung sowie Vorgaben der Programmiersprache und des Systems (z.B. virtueller Speicher) bestimmt:

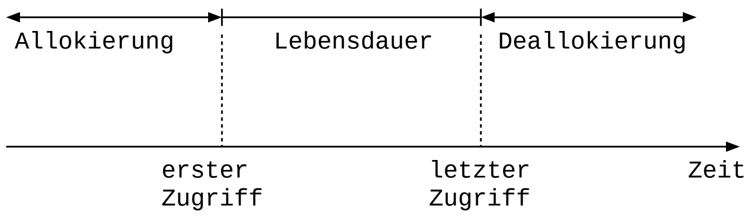


Abbildung 9.11: Lebensdauer einer Zelle

Anwendung

Meistens wird für Programme eine möglichst geringe Gesamtlaufzeit gefordert. Der durch die Gesamtkosten für Allokierung, Manipulation und Deallokierung bestimmte **Durchsatz** der Heap-Verwaltung soll also möglichst groß sein. Bei Programmen, die jedoch eine geringe maximale **Antwortzeit** erfordern (interaktive Systeme, Echtzeitsysteme), werden oft Verfahren mit schlechterem Durchsatz in Kauf genommen.

Programmiersprache

Allokierung/Deallokierung

In den meisten prozeduralen Programmiersprachen können Zellen nur **explizit** durch Anweisungen im Programm angefordert werden (z.B. `NEW` in Modula, `malloc` in C). Der Programmierer ist für die explizite Freigabe verantwortlich (z.B. `DISPOSE`, `free`). Dadurch können *ungültige Zeiger* (*dangling references*) und *Speicherverluste* (*garbage*) auftreten. Zugriffe über ungültige Zeiger auf freigegebene Zellen können das Laufzeitsystem zerstören.

Auch für prozedurale Sprachen ist eine **implizite** Freigabe denkbar, die oft sogar effizienter als eine explizite ist. Lokal in einer Prozedur angelegte Zellen (z.B. `alloca` in C) können bei Rückkehr der Prozedur ohne eigene Heap-Verwaltung implizit freigegeben werden. Falls die Lebensdauer einer Zelle unabhängig von Gültigkeitsbereichen sein soll, muß allerdings eine Heap-Verwaltung verwendet werden, die eine Speicherbereinigung (*garbage collection*) durchführt.

Operationen auf Referenzen

Modula erlaubt nur sehr eingeschränkte Operationen auf Zeiger, während in C Zeigerarithmetik erlaubt ist. Folgende Möglichkeiten sind zu unterscheiden:

- Test auf Gleichheit von Zeigern erlaubt (z.B. in Modula)
- Zeigerrichtung feststellbar (z.B. in C: `if (cp > dp)` ...)
- Zeigerarithmetik erlaubt (z.B. in C: `i = *(ip+5)`)

Falls der Zeigertyp der Programmiersprache zu mächtig ist (z.B. C, C++) oder Typinformationen zur Laufzeit nicht erkannt werden können, kann man einen sicheren „Zeigertyp“ als abstrakten Datentyp realisieren. In C++ können z.B. durch eine Klasse alle mit der Heap-Verwaltung verbundenen Operationen (Initialisierung, Zuweisung, Deallokierung) gekapselt werden.

Referenzstrukturen

Im einfachsten Fall enthalten die Zellen im Heap nur Werte und keine Referenzen auf weitere Zellen. Im allgemeinen aber können die Zellen Referenzen auf weitere Zellen mit Referenzen enthalten. Dabei kann man zwischen azyklischen Strukturen (z.B. Liste, Baum) und zyklischen Strukturen (z.B. Warteschlange) unterscheiden.

Kosten der Heap-Verwaltung

Zur Abschätzung von Antwortzeit und Durchsatz werden die Zellen des Heaps nach ihrer Lebensdauer bezüglich eines Zeitintervalls $(\mathtt{t}_1,\mathtt{t}_2)$ klassifiziert (Abb. 9.12).

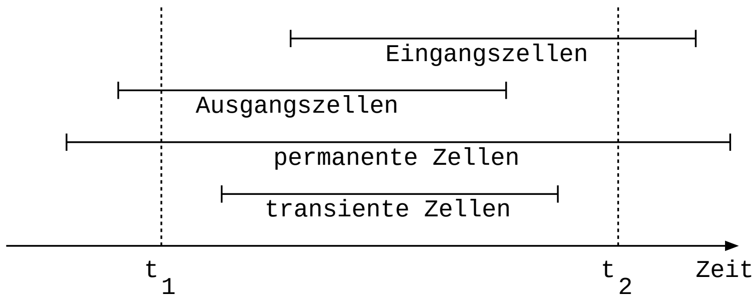


Abbildung 9.12: Klassifizierung von Zellen

Im betrachteten Intervall $(\mathtt{t}_1,\mathtt{t}_2)$ gibt es N_{ein} Eingangszellen, N_{aus} Ausgangszellen, N_{perm} permanente Zellen und N_{trans} transiente Zellen:

- vor $\mathtt{t}_1$ wurden $N_{perm} + N_{aus}$ Zellen vom Programm benötigt
- innerhalb von $(\mathtt{t}_1,\mathtt{t}_2)$ wurden $N_{ein} + N_{trans}$ Zellen neu angelegt
- nach $\mathtt{t}_2$ verbleiben $N_{perm} + N_{ein}$ vom Programm benötigte Zellen

Der Aufwand der Verfahren zur Speicherbereinigung ist im allgemeinen linear. Bei einem Verfahren, das z.B. nur von den permanenten und den Eingangszellen abhängt, beträgt der Aufwand (N_{perm}+ N_{ein}), wobei jeder Summand noch mit einem systemabhängigen Faktor zu versehen ist.

Die Werte N_{ein}, N_{aus}, N_{perm} und N_{trans} sind vom konkreten Programm abhängig: z.B. ist bei File-Verwaltungen N_{perm} sehr groß, bei Verwendung des Heaps für lokale Variablen einer Prozedur ist N_{trans} groß. Man hat meistens mit einer Häufung von sehr kurzlebigen (transienten) und sehr langlebigen (permanenten) Zellen zu rechnen. Vereinfachend betrachtet man oft nur diese beiden Klassen.

Algorithmen

Heap-Verwaltungen erfüllen im wesentlichen zwei Aufgaben: Zuerst sind die vom Programm noch benötigten (d.h. referenzierten) Zellen festzustellen (bei expliziter Freigabe fällt diese Aufgabe dem Programmierer zu), dann zu reorganisieren. Die entsprechenden Algorithmen sind weitgehend kombinierbar.

Feststellung der Referenzierbarkeit

Markierungsverfahren. Von den adressierbaren Programmvariablen aus werden alle erreichbaren Zellen und ihre Nachfolger markiert. Man benötigt pro Zelle zusätzlich ein Markierungsbit. Bei tief geschachtelten Referenzstrukturen wird der Stack der Markierungsprozedur sehr groß. Deswegen verwendet man oft nichtrekursive Varianten, die ohne Stack auskommen. Der Aufwand beträgt (N_{perm} + N_{ein}).

Referenzzähler. Um nicht mehr referenzierte Zellen möglichst früh zu erkennen, müßte man nach jeder Referenzmanipulation ein Markierungsverfahren anwenden. Es ist einfacher, die Veränderungen direkt zu beobachten. Dazu wird pro Zelle ein Zähler verwendet, dessen Wert die Anzahl der Referenzen auf diese Zelle ist. Bei jeder Manipulation einer Referenz auf die Zelle wird ihr Zähler entsprechend erhöht oder erniedrigt. Ist der Zähler = 0, wird die Zelle nicht mehr benötigt. Allerdings werden durch dieses Verfahren zyklische Strukturen nicht erkannt, da die Zähler der Zellen eines nicht referenzierten Zyklus $\neq 0$ bleiben.

Der Aufwand beträgt mindestens (N_{trans}+ N_{aus}), da pro Zuweisung eines Zeigers zusätzlich der Zähler verändert wird. Dieser Algorithmus kommt daher für Systeme, für die N_{perm} sehr groß (z.B. Files) oder eine geringe maximale Antwortzeit gefordert ist, in Frage.

Werden viele Zellen (in funktionalen Sprachen rund 70%) nur einmal referenziert, kann ein aus nur einem Bit bestehender Referenzzähler verwendet werden, der im *Zeiger* auf die Zelle abgespeichert wird. Beim Überschreiben des Zeigers wird die referenzierte Zelle freigegeben und ihr Speicherplatz kann zugleich für eine neue Zelle verwendet werden. Wird ein Zeiger aber einmal kopiert, entsteht ein Überlauf (Zählerbit=0) und die Zelle kann nur mehr durch andere Verfahren freigegeben werden. Diese Variante wird von einigen Lisp-Maschinen verwendet.

Reorganisation

Reorganisation der freien Zellen. Der Aufwand für diese Verfahren beträgt mindestens (N_{trans} + N_{aus}). Die frei gewordenen Zellen werden z.B. über **Freispeicherlisten** verwaltet. Nachteile sind der bei Zellen unterschiedlicher Größe entstehende „Speicherverschnitt“ und der Verlust an Lokalität, weil neue Zellen an einem zufälligen Speicherplatz angelegt werden. Das ist bei virtuellen Speichern besonders ungünstig.

Mit dem Markierungsverfahren kombiniert ergibt diese Freispeicher-Verwaltung im einfachsten Fall den Algorithmus **mark and sweep**:

- Initialisiere alle Markierungsbits
- Markiere alle benötigten Zellen
- Hänge alle unmarkierten Zellen in die Freispeicherliste

Dieses Verfahren ist das älteste (1960 für Lisp) und auf heutigen Systemen (viel realer und virtueller Speicher) das langsamste. Es ist in dieser Form nur noch bei nicht verschiebbaren Zellen (z.B. C) sinnvoll. Geringfügig besser sind inkrementelle bzw. parallelisierte Varianten.

Reorganisation der benötigten Zellen. Die folgenden beiden Klassen von Algorithmen kompaktieren die benötigten Zellen durch Verlegung in einen geschlossenen Bereich. Die Zellen müssen verschiebbar sein. Bei virtuellem Speicher sind Kompaktieralgorithmen aus Gründen der **Lokalität** vorteilhaft - die Arbeitsmenge (*working set*) ist nach jeder Kompaktierung minimal. Der Aufwand hängt nur von den benötigten Zellen ab (N_{perm}+N_{ein}).

Ordnungsbewahrende Kompaktierer. Diese verschieben die Zellen unter Bewahrung ihrer Ordnung. Durch sie werden auch Stacks, die redundante Einträge besitzen, kompaktiert. Derartige Stacks treten z.B. in Prolog und Lisp auf.

Der folgende Algorithmus benötigt zu jeder Zelle zusätzlich Platz für einen Hilfszeiger. Weitere Algorithmen benötigen weniger Speicherplatz bzw. nur einen Pass.

1. Verschiebe alle markierten Zellen an ihre neue Position, speichere die neue Adresse in einem Hilfszeiger der alten Zelle ab.
2. Setze alle Zeiger in den kompaktierten Zellen auf die neuen Positionen, die in den Hilfszeigern der alten Zellen stehen.

Ordnungszerstörende Kompaktierer – Kopieralgorithmen. Die folgenden Algorithmen sind einsetzbar, wenn Zellen verschiebbar sind und die Reihenfolge zwischen ihnen keine Rolle spielt. Sie zählen zu den schnellsten Verfahren, da die Feststellung der Referenzierbarkeit und die Reorganisation in einem Pass erfolgen.

Die grundlegende Idee ist, den Heap in zwei Hälften zu teilen: den *Arbeitsspeicher* und den *Freispeicher*. Benötigte Zellen werden im Arbeitsspeicher sequentiell allokiert. Ist dieser voll, werden alle erreichbaren Zellen in den Freispeicher kopiert. Dabei durchwandert man – wie beim Markierungsverfahren – alle erreichbaren Zellen. Jede Zelle wird in den Freispeicher kopiert, an ihrer Stelle im Arbeitsspeicher wird die neue Position der Zelle eingetragen, damit andere auf sie zeigende Zellen ihre Referenzen entsprechend umsetzen können. Danach wird der Freispeicher zum Arbeitsspeicher und umgekehrt.

Die Unterbrechung durch (N_{perm}+N_{ein}) Kopiervorgänge bestimmt die maximale Antwortzeit. Zur Reduktion bieten sich zwei kombinierbare Vorgehensweisen an:

- inkrementelles Kopieren
- Unterteilung des Arbeitsspeichers in Segmente (durch Anwender oder System definiert) bzw. Generationen (nach bisherigem Alter einer Zelle), die unterschiedlich oft reorganisiert werden

Kopieralgorithmen werden vor allem für Smalltalk und Lisp verwendet. Einige Prozessoren (z.B. Explorer) unterstützen inkrementelle Kopieralgorithmen.

Literaturverzeichnis

[App98] A.W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

> Das Buch beschreibt umfassend und klar die aktuellen Konzepte und Verfahren. Es werden auch objektorientierte Sprachen behandelt.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986.

> Dieses internationale Standardwerk bietet eine umfassende Einführung in den Übersetzerbau und eine Vertiefung vieler Themenbereiche.

[FH95] Ch. Fraser and D. Hanson. *A Retargetable C compiler: Design and Implementation.* Benjamin/Cummings Publishing, 1995.

> Das Buch enthält einen kompletten ANSI C Compiler mit Codegeneratoren für Intel, SPARC und MIPS Prozessoren

[Kan89] Gerry Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[Kas90] U. Kastens. *Übersetzerbau.* R. Oldenbourg Verlag, 1990.

> Das Buch ist vergleichsweise kurz und vollständig

[Mot84] Motorola. *M68000, Programmer's Reference Manual.* Prentice Hall, 1984.

[Par92] Th.W. Parsons. *Introduction to Compiler Construction, Computer Science Press.* W.H.Freeman, 1992.

> Der Verfasser legt besonderen Wert auf Klarheit und weniger auf Tiefe. Das Buch behandelt auch theoretische Grundlagen sowie die Generatoren Lex und Yacc.

[SF85] A.T. Schreiner and G. Friedman. *Compiler bauen mit UNIX - Eine Einführung.* Carl Hanser Verlag, 1985.

> Mithilfe der Übersetzer-Generatoren Lex und Yacc wird ein Compiler für ein C-Subset erstellt.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1994.

Das Buch enthält einen kleinen in Prolog geschriebenen Compiler (Kapitel 24).

[Sta89] R.M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1989.

Das Manual beschreibt einen der besten portablen C-Compiler. Er ist unter UNIX und MS-DOS als freie Software verfügbar.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

Der Oberon-Compiler und die gesamte interaktive Systemumgebung werden beschrieben. Das Buch enthält die vollständigen Programmlisten.

[Wir86] N. Wirth. *Compilerbau - Eine Einführung.* Teubner, 1986.

Es wird ein kleiner Compiler-Interpreter für ein Pascal-Subset entwickelt.

[WM92] R. Wilhelm and D. Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.

Das Buch enthält theoretische Grundlagen und praktische Verfahren nach dem aktuellen Stand der Technik. Auch funktionale und logische Programmiersprachen werden berücksichtigt.

Anhang A

Mini-Compiler in Java

```
/* ============================= Mini.java ================================

The Mini language is a modified small subset of Java/C.
A Mini program consists of a single possibly recursive function.
The language has no declarations (implicit type is integer).

Mini.java is a compiler-interpreter for Mini written in Java.
The compiler generates code for a virtual machine, which is a modified
small subset of the Java VM (integer code instead of byte code).

The one-pass compiler is implemented as a top-down recursive descent
parser calling the methods of lexical analysis and code generation.
The parser routines correspond to the the grammar rules in EBNF Notation.
The lexical analysis takes advantage of the Java class StreamTokenizer.

====================== source language syntax (EBNF) ======================

Program    = Function
Function   = identifier "(" identifier ")" Block
Block      = "{" [Statements] "}"
Statements = Statement Statements
Statement  = identifier "=" Expression ";" |
             "if" Condition Statement "else" Statement |
             "while" Condition Statement |
             "return" Expression ";" |
             Block |
             ";"
Condition  = "(" Expression ("=="|"!="|">"|"<") Expression ")"
Expression = Term {("+"|"-") Term}
Term       = Factor {("*"|"/") Factor}
Factor     = number |
             identifier |
             "(" Expression ")" |
             identifier "(" Expression ")"

================================= VM code =================================

 0    do nothing
 1 c  push constant c onto stack
```

```
 2 v  load variable v onto stack
 3 v  store stack value into variable v
 4    add two top elements of stack, replace by result
 5    subtract ...
 6    multiply ...
 7    divide ...
 8 a  jump to a if the two top elements of stack are equal
 9 a  jump if ... not equal
10 a  jump if ... less or equal
11 a  jump if ... greater or equal
12 a  unconditional jump to a
13 a  jump to subroutine start address a
14    return from function
15    stop ececution

================================ example =================================

source file "fac.mini":
-----------------------
fac(n) {
    if (n == 0)
        return 1;
    else
        return n * fac(n-1);
}

run:
----
<Java Compiler> Mini.java
<Java VM> Mini fac.mini 8

VMCode:13 3 15 2 1 1 0 9 14 1 1 14 12 25 2 1 2 1 1 1 5 13 3 6 14 0
Result:40320

========================================================================= */

//////////////////////////////////////////////////////////////////////////

import java.io.*;

public class Mini implements Globals {
    static String stab[] = new String[10]; // symbol table
    static int nv = 0;                     // number of variables
    static int code_size = 1000;
    static int stack_size = 100000;

    public static void main(String args[]) {
        Lex.init(args[0]);                  // init lexer
        Gen.init(code_size);                // init generator
        Parser.program();                   // call parser
        Gen.show();                         // show VM code
        VM.init(Gen.code,nv,stack_size);    // initialize VM
        int arg = Integer.parseInt(args[1]); // input argument
        int res = VM.exec(arg);             // call VM
```

```
        System.out.println("Result:"+res);   // print result
    }

    protected static void error(String s,int n) {
        System.out.println("error "+s+" "+n);
        System.exit(0);
    }
}

//////////////////////////////////////////////////////////////////////////

interface Globals {

    // token

    final static int T_num = 1; // number
    final static int T_id  = 2; // identifier
    final static int T_eql = 3; // ==
    final static int T_neq = 4; // !=
    final static int T_grt = '>';
    final static int T_les = '<';
    final static int T_add = '+';
    final static int T_sub = '-';
    final static int T_mul = '*';
    final static int T_div = '/';
    final static int T_lbr = '(';
    final static int T_rbr = ')';
    final static int T_clb = '{';
    final static int T_crb = '}';
    final static int T_com = ',';
    final static int T_sem = ';';
    final static int T_ass = '=';
    final static int T_eof = '$';

    final static String kw[]  = {"if","else","while","return"};
    final static int kw0 = 10;
    final static int T_if     = kw0+0;
    final static int T_else   = kw0+1;
    final static int T_while  = kw0+2;
    final static int T_return = kw0+3;

    // target code (VM)

    final static int M_nop      =  0;
    final static int M_push     =  1;
    final static int M_load     =  2;
    final static int M_store    =  3;
    final static int M_add      =  4;
    final static int M_sub      =  5;
    final static int M_mul      =  6;
    final static int M_div      =  7;
    final static int M_if_cmpeq =  8;
    final static int M_if_cmpne =  9;
    final static int M_if_cmple = 10;
```

```
    final static int M_if_cmpge = 11;
    final static int M_goto     = 12;
    final static int M_jsr      = 13;
    final static int M_return   = 14;
    final static int M_stop     = 15;
}

///////////////////////////////////////////////////////////////////////////

// lexical analysis

class Lex extends Mini {
    static int num_val;   // attribute of number
    static int id_val;    // attribute of id (index of symbol table)

    private static FileInputStream source; // imported class
    private static StreamTokenizer st;     // imported class
    private static int tok;                // tokenizer token

    static void init(String file_name) {
        try {
            source = new FileInputStream(file_name);
        }
        catch (FileNotFoundException e) {
            error("filename"+" "+e.getMessage(),0);
        }
        st = new StreamTokenizer(source);
        st.ordinaryChar('/');
        st.ordinaryChar('-');
    }

    static int scan() {
        try {
            tok = st.nextToken();
            switch(tok) {
            case StreamTokenizer.TT_EOF:
                return T_eof;
            case StreamTokenizer.TT_NUMBER:
                num_val = (int)st.nval;
                return T_num;
            case StreamTokenizer.TT_WORD:
                int i = look_kw(st.sval);
                if (i >= 0)
                    return kw0+i;
                else {
                    id_val = enter_id(st.sval);
                    return T_id;
                }
            default:
                char c = (char)tok;
                switch(c) {
                case '=':
                    if ((char)st.nextToken() == '=')
                        return T_eql;
```

```
                    else
                        st.pushBack();
                    break;
                case '!':
                    if ((char)st.nextToken() == '=')
                        return T_neq;
                    else
                        st.pushBack();
                    break;
                }
            }
        }
        catch (IOException e) {
            error("i/o"+" "+e.getMessage(),0);
        }
        return tok;
    }

    private static int look_kw(String s) {
        int i;
        for (i = 0;i < kw.length && !s.equals(kw[i]);i++) {}
        if (i < kw.length)
            return i;
        else
            return -1;
    }

    private static int enter_id(String s) {
        int i;
        for (i = 0;i < nv && !s.equals(stab[i]);i++) {}
        if (i == nv) {
            stab[i] = s;
            nv++;
        }
        return i;
    }
}

///////////////////////////////////////////////////////////////////////////

// parser (generates VM code)

class Parser extends Mini {
    private static int token;

    static void program() {
        next();
        Gen.start();
        function();
    }

    private static void function() {
        match(T_id);
        match(T_lbr);
```

```
    match(T_id);
    match(T_rbr);
    block();
    Gen.instr(M_nop);
}

private static void block() {
    match(T_clb);
    statements();
    match(T_crb);
}

private static void statements() {
    if (token != T_crb) {
        statement();
        statements();
    }
}

private static void statement() {
    int p1,p2;
    switch(token) {
    case T_id:
        int adr = Lex.id_val;
        next();
        match(T_ass);
        expression();
        Gen.instr(M_store,adr);
        match(T_sem);
        break;
    case T_if:
        next();
        condition();
        p1 = Gen.pc-1;
        statement();
        Gen.instr(M_goto,0);
        p2 = Gen.pc-1;
        match(T_else);
        Gen.setjump(p1);
        statement();
        Gen.setjump(p2);
        break;
    case T_while:
        next();
        p1 = Gen.pc;
        condition();
        p2 = Gen.pc-1;
        statement();
        Gen.instr(M_goto,p1);
        Gen.setjump(p2);
        break;
    case T_return:
        next();
        expression();
```

```
            Gen.instr(M_return);
            match(T_sem);
            break;
        case T_clb:
            block();
            break;
        case T_sem:
            next();
            break;
        default:
            error("syntax",token);
        }
    }

    private static void condition() {
        match(T_lbr);
        expression();
        int rop = token;
        next();
        expression();
        match(T_rbr);
        switch(rop) {
        case T_eql:
            Gen.instr(M_if_cmpne,0);
            break;
        case T_neq:
            Gen.instr(M_if_cmpeq,0);
            break;
        case T_grt:
            Gen.instr(M_if_cmple,0);
            break;
        case T_les:
            Gen.instr(M_if_cmpge,0);
            break;
        default:
            error("syntax",token);
        }
    }

    private static void expression() {
        term();
        while (token == T_add || token == T_sub) {
            switch (token) {
            case T_add:
                next();
                term();
                Gen.instr(M_add);
                break;
            case T_sub:
                next();
                term();
                Gen.instr(M_sub);
                break;
            }
```

```
    }
}

private static void term() {
    factor();
    while (token == T_mul || token == T_div) {
        switch (token) {
        case T_mul:
            next();
            term();
            Gen.instr(M_mul);
            break;
        case T_div:
            next();
            term();
            Gen.instr(M_div);
            break;
        }
    }
}

private static void factor() {
    switch(token) {
    case T_num:
        Gen.instr(M_push,Lex.num_val);
        next();
        break;
    case T_id:
        if (Lex.id_val != 0) {
            Gen.instr(M_load,Lex.id_val);
            next();
        }
        else {
            next();
            match(T_lbr);
            expression();
            match(T_rbr);
            Gen.instr(M_jsr,Gen.start_adr);
        }
        break;
    case T_lbr:
        next();
        expression();
        match(T_rbr);
        break;
    default:
        error("syntax",token);
    }
}

private static void next() {
    token = Lex.scan();
}
```

```
    private static void match(int x) {
        if (token == x)
            next();
        else
            error("syntax",token);
    }
}

//////////////////////////////////////////////////////////////////////////

// code generator

class Gen extends Mini {
    static int code[];    // target code
    static int pc;        // program counter
    static int start_adr; // start address

    static void init(int code_size) {
        code = new int[code_size];
        pc = 0;
    }

    static void start() {
        instr(M_jsr,3);
        instr(M_stop);
        start_adr = pc;
    }

    static void instr(int operator) {
        code[pc] = operator;
        pc = pc+1;
    }

    static void instr(int operator,int operand) {
        code[pc] = operator;
        code[pc+1] = operand;
        pc = pc+2;
    }

    static void setjump(int adr) {
        code[adr] = pc;
    }

    static void show() {
        System.out.print("VMCode:");
        for (int i = 0;i < pc;i++)
            System.out.print(code[i]+" ");
        System.out.println();
    }
}

//////////////////////////////////////////////////////////////////////////

// virtual machine
```

```
class VM extends Mini {
    private static int p[]; // program code
    private static int ip;  // instruction pointer
    private static int s[]; // stack
    private static int sp;  // stack pointer
    private static int fp;  // frame pointer
    private static int fs;  // frame size

    static void init(int code[],int nv,int stack_size) {
        p = code;
        ip = 0;
        s = new int[stack_size];
        sp = 0;
        fp = 0;
        fs = nv;
    }

    static int exec(int arg) {
        s[0] = arg;
        sp++;
        while (p[ip] != M_stop) {
            switch(p[ip]) {
            case M_push:
                s[sp] = p[ip+1];
                sp++;
                ip = ip+2;
                break;
            case M_load:
                s[sp] = s[fp+p[ip+1]];
                sp++;
                ip = ip+2;
                break;
            case M_store:
                s[fp+p[ip+1]] = s[sp-1];
                sp--;
                ip = ip+2;
                break;
            case M_add:
                sp--;
                s[sp-1] = s[sp-1] + s[sp];
                ip++;
                break;
            case M_sub:
                sp--;
                s[sp-1] = s[sp-1] - s[sp];
                ip++;
                break;
            case M_mul:
                sp--;
                s[sp-1] = s[sp-1] * s[sp];
                ip++;
                break;
            case M_div:
```

```
            sp--;
            s[sp-1] = s[sp-1] / s[sp];
            ip++;
            break;
        case M_if_cmpeq:
            sp = sp-2;
            ip = s[sp] == s[sp+1] ? p[ip+1]:ip+2;
            break;
        case M_if_cmpne:
            sp = sp-2;
            ip = s[sp] != s[sp+1] ? p[ip+1]:ip+2;
            break;
        case M_if_cmple:
            sp = sp-2;
            ip = s[sp] <= s[sp+1] ? p[ip+1]:ip+2;
            break;
        case M_if_cmpge:
            sp = sp-2;
            ip = s[sp] >= s[sp+1] ? p[ip+1]:ip+2;
            break;
        case M_goto:
            ip = p[ip+1];
            break;
        case M_jsr:
            s[sp] = ip+2;      // save return address
            s[sp+1] = fp;      // save fp
            fp = sp+2;         // set fp
            sp = fp+fs;        // set sp
            s[fp+1] = s[fp-3]; // copy argument
            ip = p[ip+1];      // goto start address
            break;
        case M_return:
            s[fp-3] = s[sp-1]; // copy return value
            sp = fp-2;         // reset sp
            fp = s[sp+1];      // reset fp
            ip = s[sp];        // goto return address
            break;
        default:
            error("runtime",p[ip]);
        }
    }
    return s[0];
  }
}

//////////////////////////////////////////////////////////////////////////
```

Anhang B

Mini-Compiler in Prolog

```
/* ============================= mini.pl ================================

The Mini language is a modified small subset of Java/C.
A Mini program consists of a single possibly recursive function.
The language has no declarations (implicit type integer).

mini.pl is a compiler-interpreter for Mini written in Prolog.
The compiler generates a syntax tree which is executed by a recursive
interpreter.

The compiler has two passes (lexical analysis and syntax analysis).
Both are implemented with definite clause grammars (DCG).
The target language (syntax tree) is specified by Prolog predicates.

====================== source language syntax (DCG) =====================

program    --> function
function   --> identifier,"(",identifier,")",block.
block      --> "{",statements,"}".
statements --> statement,statements.
statements --> "".

statement  --> identifier,"=",expression,";".
statement  --> "if",condition,statement,"else",statement.
statement  --> "while",condition,statement.
statement  --> "return",expression,";".
statement  --> block.
statement  --> ";".

condition  --> "(",expression,("==";"!=";">";"<"),expression,")".

expression --> term,exp_rest.
exp_rest   --> ("+";"-"),term,exp_rest.
exp_rest   --> "".

term       --> factor,term_rest.
term_rest  --> ("*";"/"),factor,term_rest.
term_rest  --> "".
```

```
factor     --> number.
factor     --> identifier.
factor     --> identifier,"(",expression,")".
factor     --> "(",expression,")".

============================ target language ========================== */

% syntax tree

is_int_prog(p(F)) :-
    is_fun(F).

is_fun(fun(Id,Pm,S)):-
    is_id(Id),
    is_id(Pm),
    is_stm(S).

is_stm(seq(S1,S2)) :-
    is_stm(S1),
    is_stm(S2).
is_stm(asm(Id,E)) :-
    is_id(Id),
    is_exp(E).
is_stm(ifs(E,S1,S2)) :-
    is_exp(E),
    is_stm(S1),
    is_stm(S2).
is_stm(whs(E,S)) :-
    is_exp(E),
    is_stm(S).
is_stm(ret(E)) :-
    is_exp(E).
is_stm(emp).

is_exp(con(N)) :-
    is_con(N).
is_exp(var(Id)) :-
    is_id(Id).
is_exp(fct(Id,E)) :-
    is_id(Id),
    is_exp(E).
is_exp(bin(Op,E1,E2)) :-
    is_opr(Op),
    is_exp(E1),
    is_exp(E2).

is_id(Id):- atom(Id).
is_con(N):- number(N).
is_opr(Op):- member(Op,[=,#,>,<,+,-,*,/]).

/* ============================== example ================================

source file "fac.mini":
------------------------
```

```
fac(n) {
    if (n == 0)
        return 1;
    else
        return n * fac(n-1);
}

run:
----
mini('fac.mini',8,Result,SynTree,_).

Result = 40320
SynTree =
    p(fun(fac,n,
        ifs(bin(=,var(n),con(0)),
            ret(con(1)),
            ret(bin(*,var(n),fct(fac,bin(-,var(n),con(1))))))))

======================================================================== */
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% main

mini(SourceFile,In,Out,SynTree,Msg) :-
    compiler(SourceFile,SynTree,CompMsg),
    (CompMsg='compilation ok' ->
        interpreter(SynTree,In,Out,Msg)
        ;
        Msg=CompMsg).

compiler(P0,P,M) :-
    run(lex_analysis(P0,P1),M='lexical error',
    run(syn_analysis(P1,P), M='syntax error',
    run(fail,               M='compilation ok',
    true))).

interpreter(P,In,Out,M) :-
    run(is_int_prog(P),     M='target program error',
    run(number(In),         M='input data error',
    run(execution(P,In,Out),M='execution error',
    run(fail,               M='execution ok',
    true)))).

run(Pass,_Error,Next) :-
    Pass,Next.
run(Pass,Error,_Next) :-
    \+Pass,Error.

:- use_module(library(lists)).
:- use_module(library(assoc)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% lexical analysis
```

```
lex_analysis(SourceFile,Tokens) :-
    reader(SourceFile,Chars),
    phrase(tokens(Tokens),Chars).

tokens(Ts)     --> whitespace,
                   tokens(Ts).
tokens([T|Ts]) --> tok(T),
                   !, % single solution:longest input match
                   tokens(Ts).
tokens([])     --> "".

tok('if')      --> "if".
tok('else')    --> "else".
tok('while')   --> "while".
tok('return')  --> "return".
tok('{')       --> "{".
tok('}')       --> "}".
tok(';')       --> ";".
tok(',')       --> ",".
tok('(')       --> "(".
tok(')')       --> ")".
tok(rop(=))    --> "==".
tok(rop(#))    --> "!=".
tok(rop(>))    --> ">".
tok(rop(<))    --> "<".
tok(aop(+))    --> "+".
tok(aop(-))    --> "-".
tok(mop(*))    --> "*".
tok(mop(/))    --> "/".
tok('=')       --> "=".
tok(id(I))     --> ident(Cs),{name(I,Cs)}.
tok(num(N))    --> number(Cs),{name(N,Cs)}.

ident([C|Cs])  --> letter(C),identr(Cs).
identr([C|Cs]) --> letter(C),identr(Cs).
identr([C|Cs]) --> digit(C),identr(Cs).
identr([])     --> "".

number([C|Cs]) --> digit(C),number(Cs).
number([C])    --> digit(C).

letter(C)      --> [C],{0'A=<C,C=<0'Z;0'a=<C,C=<0'z}.
digit(C)       --> [C],{0'0=<C,C=<0'9}.
whitespace     --> [C],{C=<0' }.

% ------------------------------------------------------------------------

reader(F,P) :-
    open(F,read,S),
    read_chars(S,P),
    close(S),
    !.
```

```
read_chars(S,[]) :-
    at_end_of_stream(S).
read_chars(S,[C|Cs]):-
    \+at_end_of_stream(S),
    get0(S,C),
    read_chars(S,Cs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% syntax analysis

syn_analysis(Tokens,SynTree) :-
    phrase(prog(SynTree),Tokens).

prog(p(F))          --> func(F).

func(fun(Id,Pm,S))  --> [id(Id)],['('],[id(Pm)],[')'],block(S).

block(S)            --> ['{'],stms(S),['}'].

stms(S)             --> stm(S1),stmr(S1,S).
stms(emp)           --> [].
stmr(S1,seq(S1,S))  --> stm(S2),stmr(S2,S).
stmr(S,S)           --> [].

stm(asm(Id,E))      --> [id(Id)],['='],exp(E),[';'].
stm(ifs(E,S1,S2))   --> ['if'],cond(E),stm(S1),['else'],stm(S2).
stm(whs(E,S))       --> ['while'],cond(E),stm(S).
stm(ret(E))         --> ['return'],exp(E),[';'].
stm(S)              --> block(S).
stm(emp)            --> [';'].

cond(bin(Op,E1,E2)) --> ['('],exp(E1),[rop(Op)],exp(E2),[')'].

exp(E)              --> term(E1),expr(E1,E).
expr(E1,E)          --> [aop(Op)],term(E2),expr(bin(Op,E1,E2),E).
expr(E,E)           --> [].

term(E)             --> factor(E1),termr(E1,E).
termr(E1,E)         --> [mop(Op)],factor(E2),termr(bin(Op,E1,E2),E).
termr(E,E)          --> [].

factor(con(N))      --> [num(N)].
factor(var(Id))     --> [id(Id)].
factor(fct(Id,E))   --> [id(Id)],['('],exp(E),[')'].
factor(E)           --> ['('],exp(E),[')'].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% execute target program (syntax tree)
% Env..Environment, V..Value
% return statements produce a return value replacing the environment

execution(p(F),Vin,Vout) :-
```

```
    exec(F,Vin,Vout).

exec(fun(Id,Pm,S),Vin,Vout) :-
    new_env(fun(Id,Pm,S),Env0),
    put_env(Pm,Vin,Env0,Env1),
    exec(S,Env1,Vout),
    number(Vout). % fail if no return value

exec(seq(S1,S2),Env0,Env) :-
    exec(S1,Env0,Env1),
    (number(Env1) -> % return value ?
        Env = Env1   % don't execute S2
        ;
        exec(S2,Env1,Env)).
exec(asm(Id,E),Env0,Env) :-
    exec(E,Env0,V),
    put_env(Id,V,Env0,Env).
exec(ifs(E,S1,_),Env0,Env) :-
    exec(E,Env0,1),
    exec(S1,Env0,Env).
exec(ifs(E,_,S2),Env0,Env) :-
    exec(E,Env0,0),
    exec(S2,Env0,Env).
exec(whs(E,S),Env0,Env) :-
    exec(E,Env0,1),
    exec(S,Env0,Env1),
    exec(whs(E,S),Env1,Env).
exec(whs(E,_),Env,Env) :-
    exec(E,Env,0).
exec(ret(E),Env0,V) :-
    exec(E,Env0,V).
exec(emp,Env,Env).

exec(con(V),_,V).
exec(var(Id),Env,V) :-
    get_env(Id,Env,V).
exec(fct(_,E),Env,Vout) :-
    exec(E,Env,Vin),
    Env = env(F,_),
    exec(F,Vin,Vout).
exec(bin(Op,E1,E2),Env,V) :-
    exec(E1,Env,V1),
exec(E2,Env,V2),
exec(Op,V1,V2,V).

exec(+,V1,V2,V) :- V is V1+V2.
exec(-,V1,V2,V) :- V is V1-V2.
exec(*,V1,V2,V) :- V is V1*V2.
exec(/,V1,V2,V) :- V is V1/V2.
exec(=,V1,V2,V) :- V1=V2 -> V=1;V=0.
exec(#,V1,V2,V) :- V1=V2 -> V=0;V=1.
exec(>,V1,V2,V) :- V1>V2 -> V=1;V=0.
exec(<,V1,V2,V) :- V1<V2 -> V=1;V=0.
```

```
new_env(F,env(F,B)) :- % B..Bindings of Variables
empty_assoc(B).

put_env(Id,V,env(F,B0),env(F,B)) :-
put_assoc(Id,B0,V,B).

get_env(Id,env(_,B),V) :-
get_assoc(Id,B,V). % fail if not assigned

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```