Compilerbau – eine Einführung
SS 2019

Michael Jäger

30. April 2019

(Version 3.4.3 vom 30.4.2019)

Inhaltsverzeichnis

Vorbemerkung

Das vorliegende Skript wird ständig weiterentwickelt. Ich bitte darum, mich auf Fehler aufmerksam zu machen (am besten per E-Mail).

1 Einführung

Compilerbau ist sicher eines der am besten erforschten Gebiete innerhalb der Informatik. Solide theoretische Grundlagen, formale Beschreibungsmodelle, effiziente, gut dokumentierte Algorithmen und brauchbare Implementierungs-Tools (Compiler-Generatoren) stehen dem Anwender zur Verfügung.

Trotz der Verfügbarkeit bewährter Technologie ist der Compilerbau einem raschen Wandel unterzogen. Seit Jahrzehnten werden immer neue Programmiersprachen erfunden und die vorhandenen und bewährten weiterentwickelt. Als Beispiel mag Java dienen, das von Version zu Version mit neuer Funktionalität aufwartet. Nun müssen nicht nur immer wieder neue Sprachkonzepte implementiert werden, sondern die Anwendungsentwickler erwarten auch ständig verbesserte Effizienz bei der Ausführung.

Die heutigen Java-Implementierungen sind hocheffizient, weil die Compilerbau-Techniken parallel zur Sprachentwicklung erhebliche Fortschritte verzeichnet haben. Früher wurden Programme höherer Programmiersprachen in der Regel vor der Ausführung vom Compiler in Maschinenprogramme übersetzt. Dies nennt man heute „Ahead-of-Time compilation“ (AOT). Alternativ wurde bei interpretierten Sprachen auf die Übersetzung komplett verzichtet, was zu ineffizienter Ausführung führte.

Die Entwicklung von „Just-in-Time“-Compilern (JIT-Compiler), die zur Programm-Laufzeit lauffähigen Maschinencode erzeugen, hat dem Compilerbau neue Impulse gegeben und viel höhere Ausführungsgeschwindigkeit nicht nur bei Java und den auf der Java-VM lauffähigen anderen Sprachen wie Clojure, Scala oder Groovy ermöglicht, sondern auch bei den heute sehr wichtigen Sprachen JavaScript und Python. So ist es heute, nicht zuletzt infolge des Siegeszugs der Single-Page-Webanwendungen, für die Akzeptanz eines Webbrowsers von enormer Bedeutung, wie schnell JavaScript-Code im Browser ausgeführt werden kann. Ohne integrierte JIT-Compiler kommen moderne Webbrowser nicht mehr aus.

Wohin die Reise führt, ist noch nicht abzusehen. So hat Google für Android-Applikationen, die auf der Dalvik VM laufen, zunächst einen JIT Compiler eingesetzt, diesen ab Android 5.0. aber durch einen AOT-Compiler ersetzt, um höhere Ausführungsgeschwindigkeiten zu erreichen.

Der klassische Compilerbau ist also ein Gebiet, das getrieben durch die Effizienz-Anforderungen moderner dynamischer Programmiersprachen, einem rasantem Fortschritt unterworfen ist.

Das Anwendungsspektrum der Compilerbau-Techniken geht aber weit über die

Implementierung von Compilern hinaus. Insbesondere die Algorithmen zur automatisierten Analyse hierarchisch strukturierter Dokumente haben unzählige Anwendungsgebiete.

Darüber hinaus enthält das Wissen über die Implementierung einer Programmiersprache wichtige Aspekte zur Verwendung dieser Sprache. Beispielsweise sollte ein Programmierer, der immer mehrere Möglichkeiten zur Codierung hat, diese Möglichkeiten auch im Hinblick auf die Laufzeit-Effizienz gegeneinander abwägen können. Ohne gewisse Grundkenntnisse über den generierten Maschinencode ist diese Abwägung nicht praktikabel. (Man könnte theoretisch alle möglichen Varianten mit Zeitmessung ausprobieren).

Oft wird dem Programmierer auch die Semantik bestimmter Sprachkonzepte erst durch die Betrachtung der Implementierung vollends klar. (Zugegebenermassen ist die Semantik von Programmiersprachen oft genug nur durch deren Implementierung definiert: Die Wirkung des Sprachkonzepts „X“ ist das, was das übersetzte Programm bei Verwendung von „X“ tut !)

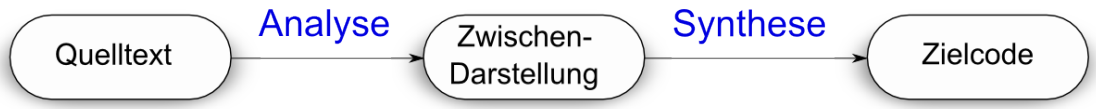
Einige Anwendungen für Übersetzerbau-Techniken:

- Klassischer (AOT-)-Compiler, z.B. für C oder C++
- Java VM mit JIT-Compiler
- Programmierumgebungen (z.B. Eclipse, Visual Studio)
- Webbrowser it Frameworks für die Entwicklung hybrider Smartphone-Applikationen
- XML-Werkzeuge (XML-Parser, XSLT-Prozessor usw.)
- SQL-Optimierer im DBMS
- Analysetools für Reverse-Engineering (automatisierte Gewinnung von Strukturinformation aus ungenügend oder gar nicht dokumentierter alter Software)

1.1 Phasenmodell eines Compilers

In dieser Einführung wird auf die Diskussion von JIT-Compilern und von anderen fortgeschrittenen Optimierungstechniken verzichtet. Stattdessen betrachten wir die Standard-Technologie, also die Übersetzung eines Programms vor dessen Ausführung in Maschinen- oder Assemblercode.

Ein Compiler transformiert einen Quelltext einer höheren Programmiersprache in Maschinencode (oder Assembler-Code). Eine sehr abstrakte Sicht auf diesen recht komplexen Prozess bietet folgendes Phasenmodell:



1.1.1 Analyse

In der **Analysephase** liest der Compiler die Quelltextdatei und erkennt die darin enthaltenen Strukturen. Dabei werden ggf. vorhandene Fehler wie nicht deklarierte Variablen oder fehlerhafte Funktionsaufrufe entdeckt und gemeldet. Falls das Programm formal korrekt ist, wird eine Zwischendarstellung als Grundlage für die weitere Verarbeitung erzeugt. Den für die Analyse zuständigen Teil des Compilers bezeichnet man auch als „Frontend“.

Die Analyse umfasst mehrere Teilfunktionen:

- Lexikalische Analyse (Scanner)

  Der Scanner muss aus einer Sequenz einzelner Zeichen primitive syntaktische Elemente, sogenannte Tokens, erkennen und klassifizieren: Zahlendarstellungen, Operatoren, Bezeichner (symbolische Namen), Schlüsselwörter, Klammern, Semikolon usw.

- Syntaxanalyse (Parser)

  Analyse des hierarchischen Programmaufbaus, z.B. Erkennung von Variablen-/Funktions-/Klassen-Definitionen, arithmetischen Ausdrücken, Anweisungen usw.

- Semantische Analyse

  Typprüfung, Berechnung von Typkonversionen u.a.

1.1.2 Zwischendarstellung

In der Zwischendarstellung werden typischerweise Listen bzw. Tabellen sowie abstrakte Syntaxbäume als Datenstrukturen verwendet.

- Symboltabellen für Bezeichner

  Ein **Bezeichner** (engl. „Identifier“) ist ein vom Programmierer gewählter symbolischer Name, der im Quelltext eines Programms ganz unterschiedliche Dinge repräsentieren kann, z.B. eine Klasse, einen Typ, eine Variable,

eine Methode oder Funktion. Typischerweise wird ein Bezeichner an einer Stelle im Programm definiert und an mehreren Stellen verwendet. In einer Symboltabelle wird zu jedem Bezeichner die Information aus der Definition abgelegt. Für jede Verwendungsstelle im Quelltext muss der Compiler anhand des Symboltabelleneintrags prüfen, ob die Verwendung im Einklang mit der Definition steht.

In den meisten Sprachen können Bezeichner in unterschiedlichen Kontexten definiert werden, z.B. innerhalb einer Klassendefinition, innerhalb einer Funktionsdefinition oder auf der äußersten Ebene („globaler“ Bezeichner). Für jeden solchen Kontext gibt es dann eine separate Symboltabelle.

- Abstrakter Syntaxbaum

  Bäume werden in der Informatik immer dort verwendet, wo hierarchische Strukturen repräsentiert werden müssen. Ein arithmetischer Ausdruck, eine Klassendefinition oder eine While-Schleife etwa sind hierarchich strukturiert und werden jeweils durch einen Syntaxbaum repräsentiert.

  Ein abstrakter Baum enthält nur solche Informationen, die für die Weiterverarbeitung nötig sind. Beispiel: Das zu einer C-Anweisung gehörende abschließende Semikolon muss zwar im Quelltext stehen, damit der Parser die Struktur erkennen kann. Für die Weiterverarbeitung ist das Semikolon jedoch überflüssig, daher taucht es im abstrakten Baum nicht auf.

1.1.3 Synthese

In der **Synthesephase** erzeugt der Compiler aus der Zwischendarstellung den Maschinencode für eine bestimmte Zielplattform. Dies kann direkt oder auf mehrere Teilphasen verteilt erfolgen. Aus der Zwischendarstellung kann der Compiler beispielsweise zunächst einen Zwischencode für eine besonders einfach strukturierte virtuelle Maschine erzeugen. Auf der Zwischcodeebene kann optimiert werden (z.B. Erkennung und Eliminierung bestimmter Rekursionsformen, Verkürzung von Sprungketten) bevor der eigentliche Zielcode generiert und danach (maschinenspezifisch) optimiert wird.

Der Zwischencode vereinfacht auch die Implementierung einer Sprache für unterschiedliche Zielmaschinen, denn nur der letzte Transformationsschritt hat maschinenspezifische Varianten.

Den für die Synthese zuständigen Teil des Compilers bezeichnet man auch als „Backend“.

1.1.4 Verschränkung der Phasen

Das beschriebene Phasenmodell könnte theoretisch in entsprechende Compiler-Phasen umgesetzt werden, die nacheinander aufgerufen werden, wobei jede Phase über eine Datei ihre Ergebnisse an die nachfolgende Phase weiterleitet:

Der Scanner liest den Quelltext und erzeugt eine Token-Datei, der Parser liest die Token-Datei und erzeugt eine erste interne Darstellung, die semantische Analyse erzeugt durch Berechnung der Typinformation eine erweiterte interne Darstellung usw.

Um die Übersetzung schneller zu machen, werden die Teilfunktionen jedoch weitgehend zeitlich verschränkt. So ruft der Parser bei der Analyse jedesmal, wenn ein neues Token benötigt wird, den Scanner auf. Der Scanner ist demnach keine eigenständige Übersetzungsphase, sondern eine Funktion, die vom Parser regelmäßig aufgerufen wird, um nur jeweils ein einziges Token zu erkennen.

Auch die Typprüfung und die Zwischencode-Generierung lassen sich oft während der Syntaxanalyse durchführen. Noch einen Schritt weiter gehen Einphasen-Compiler, die in einem Durchgang den Zielcode erzeugen. Gebräuchliche C-Compiler haben beispielsweise 3-4 Phasen.

1.1.5 Arbeitsteilung bei der Compilerentwicklung

Wenn man zunächst die interne Programmrepräsentation definiert, können Analyse- und Synthesephase parallel entwickelt werden.

Das gleiche Prinzip ist auch bei der Synthese über Zwischencode möglich: Während die Zwischencode-Generierung von einem Team entwickelt wird, kann ein zweites schon den Übersetzer vom Zwischen- in den Zielcode implementieren.

1.2 Bedeutung formaler Sprachbeschreibung

Ausgangspunkt sowohl für die Verwendung wie auch die Implementierung einer Programmiersprache ist eine Beschreibung der Syntax und der Semantik dieser Sprache.

Informale Beschreibungen, etwa Texte mit eingestreuten Diagrammen und Beispielen sind i.d.R. leicht verständlich und insbesondere zum Erlernen einer neuen Sprache gut geeignet.

Nachteile informaler Beschreibungen:

- Sie sind an vielen Stellen ungenau, lassen mehrere Interpretationsmöglichkeiten zu

- Sie sind lückenhaft

Formale Beschreibungen erfordern für den Leser zunächst die Vertrautheit mit dem Beschreibungsformalismus, der ggf. nur schwer verstehbar ist. Formale Sprachbeschreibungen sind für reale Programmiersprachen oft groß und kompliziert. Insbesondere werden für verschiedene Aspekte der Sprache unterschiedliche formale Modelle verwendet, z.B.

- reguläre Ausdrücke für lexikalische Ebene der Syntax
- kontextfreie Grammatiken für hierarchischen Programmaufbau
- Baumtransformations-Grammatiken zur Manipulation der internen Repräsentation
- Hardware-Beschreibungssprachen zur Spezifikation der Zielmaschine

Formale Beschreibungen sind zunächst einmal exakt, vollständig und widerspruchsfrei. (Die Exaktheit ist immanent, Vollständigkeit und Widerspruchsfreiheit erreicht man mit der gleichen Mühe, die es kostet, ein Programm zu entwickeln, das seinen Spezifikationen entspricht.)

Dann gibt es den einen entscheidenden Vorteil formaler Beschreibungen: **Formale Beschreibungen sind automatisierter Bearbeitung zugänglich.**

Dies ist im Zusammenhang mit Programmiersprachen von essentieller Bedeutung. Tatsächlich lassen sich heutzutage **alle** Phasen eines Compilers aus den formalen Beschreibungen der zu übersetzenden Sprache und der Zielmaschine vollautomatisch generieren. Aus der Quellsprachbeschreibung alleine lassen sich z.B. Strukturediteren, Interpretierer und Debugger generieren.

Zumindest für die Analysephase sind geeignete Generatoren (z.B. **flex** und **bison**) weit verbreitet und werden regelmäßig eingesetzt.

2 Programmiersprachliche Grundbegriffe

In diesem Abschnitt wird die notwendige Terminologie für die Verarbeitung von Programmen eingeführt. Die definierten Begriffe sind aus den Programmierpraktika zum großen Teil bekannt, sollen aber noch einmal im Hinblick auf Compilerbau-Aspekte diskutiert werden.

2.1 Laufzeit und Übersetzungszeit

Im Zusammenhang mit Compilerbau müssen allerlei Berechnungen durchgeführt werden. Im Blick auf die beteiligten Programme ist hier streng zu unterscheiden, ob der Compiler eine Berechnung durchführt („Übersetzungszeit“) oder aber das vom Compiler erzeugte Maschinenprogramm, sobald es aufgerufen wird („Laufzeit“).

2.2 Programm

Ein Programm kommt in unterschiedlichen Repräsentationen daher. Es ist daher immer notwendig, zu wissen, von welcher Programmrepräsentation die Rede ist:

- Der *Quelltext* des Programms wird vom Entwickler mittels eines Editors (einfacher Texteditor oder sprachspezifischer Strukturediteor innerhalb einer Entwicklungsumgebung) erstellt. Quelltexte werden aus technischen Gründen meist modularisiert, d.h. auf mehrere *Quelltextdateien* verteilt, abgespeichert. Der Begriff „Getrennte Übersetzung“ bedeutet, dass ein Compiler einzelne Quelltextmodule separat, also ohne Kenntnis der anderen Module, übersetzt.
- Ein *Objektmodul* ist das Ergebnis der Übersetzung eines Quelltextmoduls durch den Compiler. Das Objektmodul enthält unter anderem (grob betrachtet):
  - einen Kopfbereich mit Angaben über die Struktur des Moduls
  - Angaben über den zur Laufzeit benötigten Speicherplatz für statische nicht initialisierte Daten
  - die Werte der initialisierten Daten

  - Maschinencode, der aus den ausführbaren Anweisungen des Quelltexts erzeugt wurde
  - Symbolinformation zur Zuordnung der im Quelltext verwendeten symbolischen Information (Bezeichner, Zeilennummern u.ä.) zu den entsprechenden Speicheradressen.

    Diese Information erlaubt es, beim Debuggen des Programms die Quelltextsymbole zu verwenden. Dadurch kann man z.B. bei einer Zeilennummer (Bezug auf die Quelltextebene!) eine Unterbrechung der Ausführung vorsehen, obwohl im Maschinencode selbst keinerlei Zeilennummern vorhanden sind.

    Genauso kann man den Wert einer Variablen erfragen oder modifizieren, obwohl im Maschinencode selbst keinerlei Variablennamen vorkommen.

- Ein *ausführbares Programm* entsteht durch Zusammenführen der vom Compiler erzeugten Objektmodule mittels Binder (Linkage Editor, Linker) in Form einer Datei. Die obengenannten Bestandteile der einzelnen Objektmodule werden dabei quasi vereinigt, so dass z.B. die Programmdatei sämtlichen Maschinencode aus allen Objektmodulen und sämtliche initialisierten Daten enthält.

  Beim Binden müssen die vom Compiler vergebenen Hauptspeicheradressen, die wegen der separaten Übersetzung der einzelnen Module natürlich nur relativ zum Modulanfang vergeben werden können, umgewandelt werden in Adressen, die sich auf den Beginn des Programms beziehen.

  Dabei müssen auch in jedem Objektmodul O_1 externe Referenzen aufgelöst werden, das sind Verwendungen von Variablen, Objekten, Funktionen usw., die zwar vom Modul benutzt werden, aber innerhalb eines anderen Moduls O_2 definiert sind. Die Vergabe von (relativen) Speicheradressen für die in O_2 definierten Objekte erfolgt bei der Übersetzung von O_2. Bei der separaten Übersetzung von O_1 sind daher diese Adressen nicht verfügbar, der Maschinencode ist in dieser Hinsicht vor dem Binden unvollständig.

  Vom Binder werden ggf. auch Bibliotheksmodule mit eingebunden, die vom Programm benötigt werden (statisches Binden). Üblich ist aber heute eher dynamisches Binden, wobei die Programmdatei nur eine Liste der benötigten Bibliotheksmodule enthält.

- Ein *Prozess* ist ein Programm, das gerade ausgeführt wird. Es ist eine Programmrepräsentation im Hauptspeicher, die aus der ausführbaren Programmdatei vom Betriebssystem erzeugt wird. Diesen Vorgang nennt man „Laden“ des Programms. Die Details sind Gegenstand der Lehrveranstaltung „Betriebssysteme“.

2.3 Syntax und Semantik

Die Syntax einer Programmiersprache ist das Aussehen bzw. die Struktur der Quelltexte. Jeder Quelltext muss syntaktisch korrekt strukturiert sein, damit er übersetzt werden kann. Die Syntax wird durch eine kontextfreie Grammatik formal spezifiziert. In der Syntaxanalyse muss der Compiler jeden logisch zusammengehörenden Quelltextabschnitt einer syntaxtischen Kategorie zuordnen, d.h. feststellen, ob sich bei dem Quelltextabschnitt um eine Klassendefinition, einen arithmetischen Ausdruck, einen Methodenaufruf usw. handelt. Diese Kategorien werden in der Grammatik durch Nonterminalsymbole repräsentiert. Für jede Kategorie erfolgt die Weiterverarbeitung in spezifischer Weise.

Die Semantik ist die Bedeutung, die den verschiedenen syntaktischen Gebilden zugeordnet wird. Von der Semantik hängt es ab, in welcher Weise die Weiterverarbeitung durch den Compiler erfolgt. Beispiel: Ein arithmetischer Ausdruck wie `x+3*sin(y)` repräsentiert einen Wert. Die Syntaxregeln geben an, wie er aussehen muss. Die Semantik ist der repräsentierte Wert. Die Weiterverarbeitung durch den Compiler muss also gemäß der Semantik auf die Berechnung des Werts ausgerichtet sein. Details dazu stehen weiter unten.

Für einige wichtige syntaktische Kategorien werden semantische Aspekte nachfolgend diskutiert. Dabei wird auf formale Semantikspezifikation (z.B. in Form einer denotationalen Semantik) bewusst verzichtet.

2.4 Literal, Konstante

Ein Literal repräsentiert im Quelltext einen festen Wert mit einer spezifischen Syntax. Beispiele für C++-Literale sind

```
0x10ff          // hexadezimales Ganzzahlliteral
-14.34e17       // Gleitpunkt-Literal
'a'             // Zeichen-Literal
"hallo"         // String-Literal
```

Der Begriff „Konstante“ wird nachfolgend in einem allgemeineren Sinn für alle festen Werte und deren Repräsentationen verwendet werden. Im Quelltext können Konstanten, z.B. durch Literale oder Bezeichner (der Bezeichnerkategorie „Konstantenbezeichner“) repräsentiert werden.

2.5 Bezeichner

Die syntaktische Kategorie „Bezeichner“ (engl. „identifier“) wird für die vom Programmierer im Quelltext vergebenen symbolischen Namen verwendet. Bezeich-

ner repräsentieren ganz unterschiedliche Dinge innerhalb eines Quelltexts, z.B. Konstanten, Variablen, Typen, Klassen, Objekte, Methoden usw.

Bezeichner werden (zumindest in den Ihnen bekannten Programmiersprachen) im Quelltext an einer Stelle (manchmal auch an mehreren Stellen) deklariert und können dann an anderer Stelle verwendet werden.

Aus der Art der Deklaration (Konstantendeklaration, Variablendeklaration usw.) geht hervor, was der Bezeichner im Quelltext repräsentiert. Wir nennen dies im folgenden die *Bezeichnerkategorie*.

Welche Bezeichnerkategorien es gibt, hängt immer von der betrachteten Programmiersprache ab. In fast allen Programmiersprachen gibt es Variablenbezeichner, Konstantenbezeichner und Funktionsbezeichner. Objekt- und Klassenbezeichner sind dagegen nur Quelltexten objektorientierter Sprachen zu finden.

Die Semantik (Bedeutung) einer Verwendungsstelle eines Bezeichners im und damit die Art der Verarbeitung durch den Compiler hängt natürlich von der Bezeichnerkategorie ab.

Beispiel:

Im Listing 2.1 findet man sowohl Deklarations- als auch Verwendungsstellen von Bezeichnern verschiedener Kategorien, z.B.

Zeile	**Bezeichner**	**Deklaration**	**Bezeichnerkategorie**
1	stringvektorgroesse	ja	Konstantenbezeichner
4	std	nein	Namespacebezeichner
7	main	ja	Funktionsbezeichner
8	puffer2	ja	Variablenbezeichner
9	kommandozeile	nein	Klassenbezeichner
9	zeile	ja	Objektbezeichner
13	ignoriere_sigint	nein	Funktionsbezeichner
19	zeile	nein	Objektbezeichner
19	lies	nein	Methodenbezeichner

2.6 Deklaration, Definition

In den meisten Programmiersprachen müssen Bezeichner deklariert werden. Damit werden dem Compiler die Bezeichnerkategorie und abhängig davon weitere Bezeichnerattribute mitgeteilt, z.B. in einer Variablendeklaration der Typ der Variablen oder in einer Funktionsdeklaration die Parametertypen und der Typ des Resultats.

Diese Information wird dann an jeder Verwendungsstelle des Bezeichners benötigt. Einerseits bestimmt die Bezeichnerkategorie die **Semantik der Verwendungsstelle** und damit die **Maschinencodeerzeugung**.

Listing 2.1 C++-Quelltextmodul mit Bezeichnern

```
 1      const int stringvektorgroesse=1024;
 2      const int puffer2groesse=4096;
 3
 4      using namespace std;
 5      using namespace signale;
 6
 7      main(){
 8        char puffer2[puffer2groesse];
 9        kommandozeile zeile;
10        char * stringvektor [stringvektorgroesse];
11        int string_anzahl, laenge;
12
13        ignoriere_sigint(); // CTRL-C ignorieren
14
15        string mein_prompt("Yo: ");
16
17        while(1){
18          zeile.prompt(mein_prompt);
19          laenge=zeile.lies();
20
21          string_anzahl=0;
22          if(laenge>0)
23            string_anzahl=zeile.zerlege (stringvektor,
24                                         stringvektorgroesse,
25                                         puffer2, puffer2groesse);
26
27          if (string_anzahl > 0) {// erfolgreiche Zerlegung
28            kommando *k=new kommando(stringvektor, string_anzahl);
29            k->bearbeiten();
30            delete k;
31          }
32        }
33      }
```

Beispiel: `x=2*y;`

Ohne genauer auf den Maschinencode einzugehen, den ein Compiler aus dieser Wertzuweisung erzeugen wird, kann man sich leicht vorstellen, dass abhängig von der Bezeichnerkategorie des Bezeichners `y` unterschiedliche Maschinenbefehle generiert werden müssen:

- Ist z.B. y ein Variablenbezeichner, repräsentiert die Verwendungsstelle den Variablenwert.

  Auf der Maschinencodeebene wird typischerweise zunächst ein Befehl benötigt, der den im Hauptspeicher gespeicherten Wert in ein Register des Prozessors lädt. Danach muss der Registerinhalt verdoppelt werden. Der **Maschinencode ist auch vom Typ der Variablen** abhängig: Man könnte sich z.B. vorstellen, dass bei einem Ganzzahltyp die Verdopplung durch eine Schiebeoperation des Registers (Arithmetischer Linksshift um ein Bit) erledigt wird. Andererseits wird bei einem Gleitkommatyp wohl ein Gleitkommamultiplikationsbefehl im Maschinencode verwendet werden. Abgesehen davon stehen für Gleitkomma- und Ganzahlwerte oft unterschiedliche Registersätze zur Verfügung, so dass auch die Ladeoperationen (d.h. der Transfer vom Hauptspeicher in das Register) unterschiedliche Maschinenbefehle benötigen.

- Ganz anders sollte der Compiler verfahren, wenn y als Konstante mit dem Wert 17 deklariert ist. Anstatt Maschinencode für die Multiplikation zu erzeugen, sollte der Compiler selbst das Ergebnis berechnen.

Außerdem wird die Information aus der Deklaration benötigt, um die Korrektheit der Verwendung zu überprüfen. Sehr viele Programmierfehler können somit durch den Compiler aufgedeckt werden.

Beispiel: Die Anweisung `x=2*y;` kann als fehlerhaft klassifiziert werden, falls die Bezeichnerkategorien und/oder die Typen von x oder y die Operation nicht erlauben, z.B. falls x ein Konstanten- oder Klassenbezeichner ist oder falls für den Typ von y keine Multiplikation definiert ist.

Der Compiler benötigt an einer Verwendungsstelle eines Bezeichners also immer die Information aus der Deklarationsstelle dieses Bezeichners, sowohl zur Fehlererkennung als auch zur korrekten Maschinencode-Erzeugung.

Die Begriffe *Definition* und *Deklaration* werden z.B. in C und C++ unterschieden, im Kontext anderer Sprachen machmal aber auch synonym verwendet. Wir werden den Begriff „Deklaration“ nachfolgend als den allgemeineren Begriff verwenden, während wir eine „Definition“ im Sinne von C/C++ als Spezialfall einer Deklaration auffassen wollen. Der Unterschied bezieht sich dabei auf die

Maschinencode-Erzeugung für statische Variablen und Objekte, sowie Methoden und Funktionen bei separater Übersetzung von Modulen:

Der Speicherplatz für eine statische Variable oder ein statisches Objekt muss bei der Übersetzung eines Moduls reserviert werden. Der Compiler muss wissen, in welchem Modul. Dazu dient die Definition.

Wird auf den Bezeichner in einem anderen Modul Bezug genommen, benötigt der Compiler zur Übersetzung dieses Moduls die Bezeichnerkategorie und Typinformation. Dazu dient eine Deklaration.

Beispiel:

Modul A:

```
int i=5;  // Variablen-Definition, Compiler reserviert Speicherplatz im
          // zugehörigen Objektmodul
....
```

Modul B:

```
extern int i;  // Variablen-Deklaration, Compiler reserviert keinen Speicherplatz
               // in B wird die Variable aus A verwendet
....
```

Bei modularisierten Programmen muss ein Bezeichner in **jedem** Modul, in dem er verwendet wird, auch deklariert sein. Er muss aber in **genau einem** Modul des Programms definiert werden.

Entsprechendes gilt für die Codeerzeugung für Funktionen und Methoden. Die Deklaration (oder der Prototyp) enthält die Schnittstelleninformation, die der Compiler an jeder Verwendungsstelle benötigt.

Die Definition enthält die Implementierung, also wird der Compiler bei der Verarbeitung Maschinencode erzeugen.

Da der Compiler die in einer Deklaration eines Bezeichners vorhandene Information an jeder Verwendungsstelle benötigt, wird sie in einer compilerinternen Datenstruktur, einer sogenannten Symboltabelle oder Symbolliste, abgespeichert. Die Analyse einer Deklaration führt also zu einem neuen Eintrag in der Symboltabelle, der den Bezeichner selbst, dessen Bezeichnerkategorie und alle sonstigen auf der Deklaration hervorgehenden Merkmale enthält.

2.7 Gültigkeitsbereich

Für die Erstellung komplexer, aus vielen separaten Modulen bestehender Softwaresysteme, an der i.d.R. viele Entwickler beteiligt sind, wäre es sehr hinderlich, wenn alle Bezeichner systemweit eindeutig sein müssten.

Stattdessen führt man das Konzept der Lokalität von Bezeichnern ein. So kann beispielsweise der Programmierer innerhalb einer Funktionsdefinition lokale Hilfsvariablen definieren und verwenden, ohne sich darum zu kümmern, ob in anderen Funktionen deren Bezeichner schon einmal verwendet wurden.

Gemäß dem Geheimnisprinzip sind die lokalen Bezeichner ausserhalb der Funktionsdefinition nicht sichtbar, d.h. nicht benutzbar.

Der Begriff „Gültigkeitsbereich“ bezieht sich auf den Quelltext(!) eines Programms und ist ein Attribut einer Bezeichners. In einer ersten Näherung könnte man sagen:

Der Gültigkeitsbereich eines Bezeichners ist der Bereich innerhalb des Quelltexts, in dem der Bezeichner verwendet werden kann.

Diese Definition ist noch verfeinerungsbedürftig, da in einem Quelltext mehrere Deklarationen mit dem gleichen Bezeichner vorkommen können, z.B. eine globale Konstantendefinition und eine bezüglich einer Funktionsdefinition lokale Variablendefinition des gleichen Bezeichners *x*.

```
const float x=3.5;

int func () {
  short x=1;
  int   y=x;
  ...
}
```

In der Tat bezieht sich der Gültigkeitsbereich nicht auf den Namen *x*, sondern auf eine bestimmte Deklaration von *x*. Wir können also sowohl den Gültigkeitsbereich der globalen Deklaration von *x* als auch den der lokalen Deklaration bestimmen.

Also, etwas genauer:

Der Gültigkeitsbereich einer Deklaration ist der Bereich innerhalb des Quelltexts, in dem der deklarierte Bezeichner verwendet werden kann.

Statt „Gültigkeitsbereich“ wird auch oft der Begriff „Sichtbarkeitsbereich“ bzw. der englische Begriff „Scope“ verwendet.

Wo der Gültigkeitsbereich einer Deklaration beginnt und wo er endet, wird für jede Programmiersprache abhängig von der Bezeichnerkategorie und der Deklarationsstelle im einzelnen genau spezifiziert.

Dabei sind manche „schwammigen“ Formulierungen, mit denen viele Programmierer vielleicht leben können, bei der Implementierung der Sprache ungeeignet. Nur **exakte** Spezifikationen helfen hier weiter.

Beispiel:

Eine ungenaue Spezifikation des Gültigkeitsbereichs eines Funktions- oder Methoden-lokalen Bezeichners könnte lauten:

„Der lokale Bezeichner ist von der Stelle seiner Deklaration bis zum Ende der Funktions- bzw. Methodendefinition gültig."

Betrachten wir C-Deklarationen, stellt sich die Frage, wo genau der Gültigkeitsbereich beginnt:

```
int i=17;

void func () {  int j=i, i=j+i, k=i+1;  ... }
```

Hier werden in derselben Definition innerhalb von *func* drei lokale Variablen definiert und initialisiert. Wo aber beginnt genau der Gültigkeitsbereich der lokalen Definition von i? Die schlampige Spezifikation lässt mehrere Alternativen zu.

1. Ist die lokale Deklaration von i schon ab dem Beginn der Gesamtdeklaration

   ```
   int j=i, i=j+i, k=i+1;
   ```

   gültig, so sind alle Initialwerte undefiniert, weil schon an j der noch undefinierte Wert des lokalen i zugewiesen wird.

2. Ist die lokale Deklaration von i ab dem Beginn der Teildeklaration

   ```
   i=j+i,
   ```

   gültig, so hat zwar j einen gültigen Initialwert, nämlich den Wert 17 des in

   ```
   j=i,
   ```

   gültigen globalen Bezeichners, aber die Initialisierung von i selbst wäre rekursiv und damit undefiniert.

3. Beginnt die Gültigkeit des lokalen i unmittelbar hinter der Teildeklaration von i, dann ist im Initialwertausdruck noch das wohldefinierte globale i gültig und der Initialwert ist 34. Im Initialwertausdruck für k ist in diesem Fall mit i aber schon das lokale i gemeint, so dass der Initialwert von k 35 wäre.

4. Beginnt die Gültigkeit des lokalen i erst nach der Gesamtdeklaration dann bezieht sich jede Verwendung von i in den Initialwertausdrücken auf die globale Deklaration. Überlegen Sie, welche Initialwerte die lokalen Variablen in diesem Fall annehmen! (Tipp: j=17, i=34, k=18 ist falsch!)

Im Compiler wird ein Gültigkeitsbereich durch eine Symboltabelle repräsentiert (ggf. durch eine Untertabelle).

2.7.1 Benannte Gültigkeitsbereiche und qualifizierte Bezeichner

Ein qualifizierter Bezeichner wird gebildet aus Gültigkeitsbereichsnamen und einem (im Hinblick auf den Gültigkeitsbereich) lokalem Namen. Beispiele in C++ für benannte Gültigkeitsbereiche sind Klassen und Namespaces.

```
namespace Parser {
  int expr    (void);
  int term    (void);
  int factor  (void);
  int primary (void);
}
```

Parser ist ein benannter Gültigkeitsbereich, *expr* und die anderen Funktionsbezeichner sind lokal dazu. Die Verwendung der lokalen Bezeichner erfolgt in qualifizierter Form, z.B.

```
int Parser::expr (void) { ... }  // qualifizierter Name in der Deklaration
```

oder

```
main() {
  ...
  result = Parser::expr();  // qualifizierter Name im Funktionsaufruf
  ...
}
```

2.7.2 Verschachtelung von Gültigkeitsbereichen

Gültigkeitsbereiche sind meist hierarchisch ineinander verschachtelt, so dass eine lokale Deklaration eines Bezeichners eine im umfassenderen Bereich vorhandene andere Deklaration des selben Bezeichners *verdeckt*.

```
namespace n {
  int i;
  ...
  class  c {
    ...
    int i;
    class  c1 {
      int i;
```

```
        ...
        class c11 {
          ...
        };
      };
    };
  }
```

Während der Übersetzung muss der Compiler Buch führen über die an einer bestimmten Quelltextstelle gültige Hierarchie der Gültigkeitsbereiche. Die Verwendungstelle eines Bezeichners wird i.d.R. zu einer Durchsuchung aller Gültigkeitsbereiche führen, in denen der Verwendungstelle liegt, angefangen bei den lokalen Bereichen.

Im obigen Beispiel muss bei einer Verwendung von *i* innerhalb von c11 die Deklarationsstelle in c11 selbst und dann von innen nach außen in den umfassenderen Gültigkeitsbereichen gesucht werden.

Für die Buchführung bietet sich an, jeden Gültigkeitsbereich als Symboltabelle zu speichern und die aktuelle (d.h. an einer bestimmten Quelltextstelle gültige) Verschachtelung in Form einer verketteten Liste der Symboltabellen darzustellen.

2.8 Schlüsselwörter

Ein Schlüsselwort ist eine sprachspezifisch fest vorgegebene Zeichenfolge mit einer rein syntaktischen Funktion: Es ermöglicht bzw. erleichtert dem Compiler die Syntaxanalyse. Typische Beispiele sind die C-Schlüsselwörter **if**, **else**, **while**, **do**, **class**, **typedef** usw. Sie stehen gleich zu Beginn eines Sprachkonstrukts, so dass der Compiler daran sofort erkennen kann, um welches Konstrukt es sich handelt.

Es gibt Sprachen mit allzuvielen Schlüsselwörter, vor allem COBOL, die den Quelltext unnötig aufblähen und andererseits Sprachen völlig ohne Schlüsselwörter, z.B. LISP, deren einfache Syntax manchem etwas gewöhnungsbedürftig erscheint.

Schlüsselwörter nennt man *reserviert*, wenn die Zeichenfolge nicht für Bezeichner verwendbar ist. Dies ist der Normalfall, das Konzept nichtreservierter Schlüsselwörter, wie es etwa in PL/I eingeführt wurde ist schlichtweg unsinnig: Welche Vorteile bringt es, wenn Bezeichner genauso lauten können wie Schlüsselwörter? Betrachten Sie dazu die PL/I-Anweisung:

```
if if=then then then:=if else else:=then;
```

Fazit: Unnötiges Kopfzerbrechen beim Compilerbauer, Schimpf und Schande über den Programmierer, der davon Gebrauch macht!

2.9 Ausdruck

Ein Ausdruck (engl. „expression“) repräsentiert (als Bestandteil eines Quelltexts) einen Wert.

Syntaktische Bestandteile eines Ausdrucks sind z.B:

- Atomare Ausdrücke wie Literale und Bezeichner
- Operatoren
- Klammern

Wie verarbeitet der Compiler eine Ausdruck?

- Da der Ausdruck einen Wert darstellt, muss dieser letzlich ermittelt werden. Man nennt dies die „Auswertung“ des Ausdrucks.

  Im einfachsten Fall, wenn keine Variablen und Funktionsaufrufe im Ausdruck enthalten sind, kann der Compiler den Ausdruck selbst auswerten.

  Andernfalls muss der Compiler Maschinencode erzeugen, damit der Wert zur Laufzeit, d.h. beim Aufruf des vom Compiler generierten Maschinenprogramms, berechnet werden kann.

- Bei der syntaktischen Analyse komplexer Ausdrücke geht es um die korrekte Zuordnung der Operanden zu den Operatoren. Dies ist deswegen nicht trivial, weil üblicherweise keine vollständige Klammerung gefordert wird, sondern Präzedenz- und Assoziativitätsregeln zur Zuordnung berücksichtigt werden müssen, z.B. „Punktrechnung vor Strichrechnung“ im arithmetischen Ausdruck

  ```
  3*4+5*6
  ```

- Die sogenannte „Semantische Analyse“ prüft die Kompatibilität der Typen von Operatoren und Operanden. Dabei wird zu jedem (Teil-)Ausdruck ein Typ berechnet.

2.10 Speicherplatz, Adresse, Referenz, Dereferenzierung

Den Begriff „Speicherplatz“ wollen wir nachfolgend verwenden für einen Bereich im Hauptspeicher, der zum Abspeichern eines Werts genutzt wird, und dem eine (Anfangs-) Adresse und eine Größe (in Bytes) zugeordnet ist.

Beispiel: Der 4-Byte Speicherplatz mit der Anfangsadresse 1000 erstreckt sich über die Bytes 1000-1003.

Als Referenz bezeichnen wir jeden mit Typinformation versehenen Repräsentanten eines Speicherplatzes im Quelltext. Die Typinformation bestimmt einerseits die konkrete Codierung der abgespeicherten Werte und andererseits die Menge der erlaubten Operationen. Ohne die Typinformation lässt sich ein auf einem Speicherplatz abgespeicherte Bytesequenz nicht interpretieren. Umgekehrt muss beim Abspeichern eines Werts die Typinformation benutzt werden, um die Codierung zu bestimmen, beispielsweise könnte man die Zahl 4 als vorzeichenbehaftete Ganzzahl in Zweierkomplementdarstellung abspeichern oder als char-Wert `'4'`, als C-String `"4"`, als C++-String-Objekt, als float- oder double-Gleitkommawert usw.

Eine wichtige Operation für Referenzen ist die **Dereferenzierung**, die den im Speicherplatz gespeicherten Wert liefert. Die Dereferenzierung erfolgt zur Laufzeit. Auf der Maschinencodeebene erfolgt eine Dereferenzierung typischerweise in Form von Maschinenoperationen, die Hauptspeicher-Lesezugriffe durchführen.

Auf der Quelltextebene ist zwischen *impliziter* und *expliziter* Dereferenzierung zu unterscheiden: Bei der impliziten Form gibt es keinen Operator! Das bekannteste Beispiel für eine Referenz ist ein Variablenbezeichner. Er repräsentiert im Quelltext einen Speicherplatz und ihm ist Typinformation zugeordnet, die die Interpretation des Speicherplatz-Inhalts ermöglicht. Betrachten wir zwei Verwendungsstellen einer Variablen *i* innerhalb einer Wertzuweisung

```
int i=5;  // Definitionsstelle
i=i+1;    // zwei Verwendungsstellen
```

Auf der linken Seite der Wertzuweisung repräsentiert der Bezeichner *i* den Speicherplatz. Die Wertzuweisungsoperation dient dazu, einen neuen Wert dort abzulegen. Auf der rechten Seite dagegen repräsentiert derselbe Bezeichner *i* nicht den Speicherplatz, sondern den darin abgespeicherten Wert 5. Wir haben es hier mit einer impliziten Dereferenzierung zu tun: Abhängig von der Verwendungsstelle, hier z.B. linke oder rechte Seite der Wertzuweisung, wird ohne sichtbaren Operator dereferenziert oder auch nicht.

Beispiel für eine explizite Dereferenzierung ist der einstellige $*$-Operator:

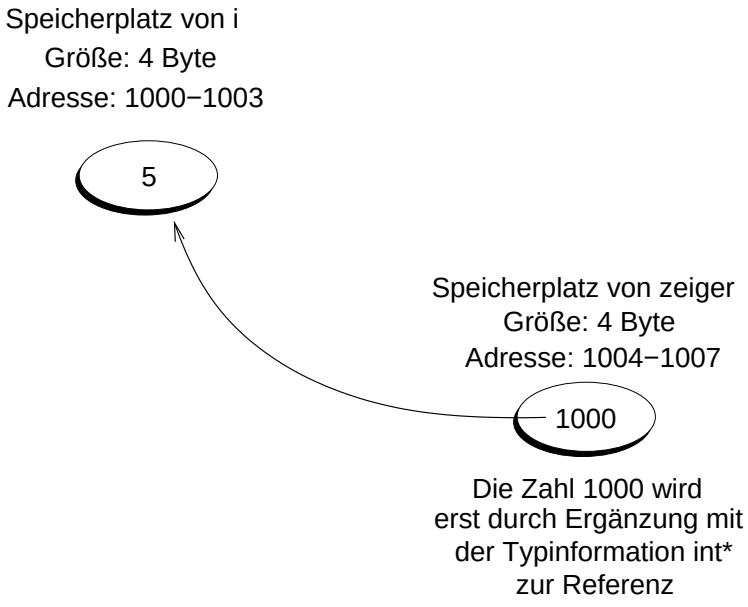
```
int i=5;
int *zeiger= &i;

*zeiger = *zeiger * 2;
```

Bei der zweiten Verwendungsstelle von *zeiger* auf der rechten Seite der Wertzuweisung liegt sowohl eine implizite als auch eine explizite Dereferenzierung vor:

- *zeiger* repräsentiert als Variablenbezeichner einen Speicherplatz
- weil die Verwendungsstelle auf der rechten Seite ist, wird implizit dereferenziert. Ergebnis ist der Wert: die „Adresse“ der Variablen *i*.

  Genauer betrachtetet ist der Wert einer Zeigervariablen natürlich nicht eine „nackte“ Adresse, mit der man nicht viel anfangen könnte, sondern eine Referenz: wir haben die Typinformation und damit die Größe des Speicherplatzes und die Spezifikation der Codierung.
- Mit dem vorangestellten Dereferenzierungsoperator $*$ wird das Ergebnis der impliziten Dereferenzierung, in diesem Fall eine Referenz, noch einmal explizit dereferenziert: Ergebnis ist der Wert 5.



2.11 Typen, Datentypen und Klassen

2.11.1 Typsysteme

Ein *Typ* repräsentiert eine Wertemenge zusammen mit den dazugehörigen Operationen. Aus Implementierungssicht ist Typinformation wichtig zur Bestimmung der Größe der Speicherplätze, der konkreten Codierung für die Werte und der Überprüfung der korrekten Verwendung von Operatoren, Methoden und Funktionen.

Es gibt typisierte und untypisierte Programmiersprachen. Typisierte Sprachen haben ein Typsystem mit vordefinierten elementaren Typen (auch *Basistypen*), z.B. in C „unsigned int“, sowie Typkonstruktoren zur Konstruktion neuer benutzerdefinierter Typen. Beispielsweise bietet C Konstruktoren für Aufzählungstypen (*enum*), Feldtypen, Verbundtypen (*struct*), Variantentypen (*union*), Zeigertypen oder Funktionstypen. Mit einem Typkonstruktor sind in der Sprache jeweils vordefinierte Operationen zur Konstruktion von komplexen Werten und zur Selektion von Komponenten komplexer Werte assoziiert. Ein typisches Beispiel ist die Selektion von Feldkomponenten mit dem `[]`-Operator, z.B. in `f[i+1]`. Die Konstruktoren lassen nahezu beliebig kombinieren.

In einer typisierten Sprache gibt es ein Regelwerk, das für Operatoren, Funktionen und Methoden jeweils die erlaubten Operanden- bzw. Argumenttypen festlegt. Diese Regeln sind meist nicht ganz einfach, wie man am Beispiel des Additionsoperators leicht sehen kann. Grob gesagt, müssen die Operanden Zahlen sein. Nun gibt es aber in einer Programmiersprache oft eine ganze Reihe von Typen für Zahlen (in Java z.B. `byte`, `short`, `int`, `long`, `float`, `double`, `Integer`, `Long`), die im Rechner unterschiedlich codiert werden. Eine weitere Komplikation der Regeln entsteht dadurch, dass ein Prozessor zwischen Ganzzahl- und Gleitpunktarithmetik unterscheidet, dass also auf der Hardwareebene zwei Sorten von Arithmetik verfügbar sind. Der Compiler muss daher für jeden Ausdruck zunächst den Typ bestimmen um dann zu prüfen, ob dieser Typ im Kontext erlaubt ist, ob ggf. irgendwelche Konvertierungen (z.B. von `int` nach `float`) erfolgen müssen.

Bei einer statisch typisierten Sprache kann dies der Compiler während der Analyse vollständig erledigen. In modernen objektorientierten Sprachen sind die Typen zur Übersetzungszeit nicht vollständig bestimmbar, so dass ein Teil der Typberechnung erst während des Programmlaufs erfolgen kann.

2.11.2 Konversionen

Eine *Konversion* (genauer gesagt *Typkonversion*) ist die Transformation eines Ausdrucks oder Werts in einen äquivalenten Ausdruck bzw. Wert eines anderen Typs.

Konversionen erfolgen explizit durch Konversionsoperationen (*type cast*) oder implizit gemäß sprachspezifischen Konversionsregeln.

In C/C++/Java sind sowohl explizite Konversionsoperationen („typecast“-Operator) als auch implizite Konversionen, z.B. zwischen den arithmetischen Typen, verfügbar.

```
int i=1;
float x;
```

```
x=i;           // implizite Konversion von i
i = (int) x;   // explizite Konversion von x
```

In Java kommen noch die Konversionen zwischen einfachen Typen wie *int* und den mathematisch äquivalenten Klassen (*Integer*) hinzu (autoboxing und unboxing).

Automatische Typanpassungen ersparen dem Programmierer etwas Schreibaufwand, erschweren andererseits aber auch die Fehlererkennung durch den Compiler.

Sind die zu konvertierenden Werte Konstanten, erfolgt die Konversion bei der Übersetzung, ansonsten muss entsprechender Maschinencode erzeugt werden.

2.11.3 Datentypen, Datenkapselung und Geheimnisprinzip

Wenn Typinformation und die Definition der zugehörigen Operationen eine syntaktische Einheit bilden, spricht man von einem „Datentyp“. In diesem Sinn ist auch eine Klassendefinition in einer objektorientierten Sprache ein Datentyp.

Ein Datentyp repräsentiert eine Menge von Objekten in der realen Welt zusammen mit den zur Verarbeitung notwendigen Operationen. Oft sind die Objekte der realen Welt komplex („Rechnung“) und die Zuordnung eines geeigneten Typs der Implementierungssprache ist eine nichttriviale Aufgabe, bei der der Programmierer zwischen verschiedenen Alternativen entscheiden muss.

Es macht daher Sinn, zwischen zwei Sichten auf den Datentyp zu unterscheiden:

- Die Sicht des Implementierers, der das Datenmodell mit geeigneten Typkonstruktoren der Implementierungssprache umsetzen muss.
- Die Sicht des Nutzers, den die Implementierungsdetails i.d.R. nicht interessieren.

Die Verwendung des Datentyps soll unabhängig von der konkreten Implementierung erfolgen. Dazu erzwingt eine Sprache mit geeigneten Sichtbarkeitsregeln, dass ein Zugriff auf Objekte des Datentyps nur über eine Reihe von dafür vorgesehenen Operationen erfolgen darf. Diese werden als „Schnittstelle“ des Datentyps bezeichnet. Von dem konkreten Wissen über die verwendeten Typkonstruktoren darf der Nutzer keinen Gebrauch machen. Dieses sogenannte „Geheimnisprinzip“ erlaubt eine spätere Änderung der Implementierung ohne Änderungsbedarf beim Nutzer, was insbesondere bei Bibliotheken für Container-Datentypen wie Listen, Mengen usw. ein unschätzbarer Vorteil ist.

Wenn beim Nutzer keinerlei Information über die Implementierung vorhanden ist, spricht man auch von einem „abstrakten“ Datentyp.

2.11.4 Klassen

In objektorientierten Sprachen treten Klassen als Ersatz bzw. Erweiterung benutzerdefinierter Datentypen auf. Gegenüber einem Datentyp kommt hier noch ein Vererbungskonzept hinzu, das für eine bessere Wiederverwendbarkeit sorgt.

Die Typverarbeitung im Compiler wird insbesondere durch die Vererbung und die damit im Zusammenhang stehende Polymorphie (s. unten) komplexer.

2.11.5 Generische Typen

In vielen modernen Sprachen gibt es generische Typen. Dabei handelt es sich meist um Container-Typen, z.B. Listen oder Mengen, die man unabhängig vom Typ der Elemente definieren will. Für den Elementtyp wird daher eine Typvariable verwendet, die einen beliebigen Typ repräsentiert. Ggf. kann man auch die Menge der gültigen Elementtypen durch sprachspezifische Restriktionen einschränken, z.B. wenn man für Listenelemente eine Sortiermethode definieren will und dafür voraussetzen muss, dass der Elementtyp einen Größenvergleich unterstützt. Der Compiler wird in seiner internen Typrepräsentation dann ebenfalls Typvariablen benötigen.

2.11.6 Typrepräsentation im Quelltext

Im Quelltext werden Typen unterschiedlich repräsentiert:

- **Typausdrücke**

  Ein Typausdruck (auch „anonymer“ oder „unbenannter“ Typ) repräsentiert einen komplexen benutzerdefinierten Typ. Zur Bildung werden die Typkonstruktoren verwendet.

  C++-Beispiele: `int* , float& (int*, double[])`

- **Typbezeichner**

  Ein Typbezeichner (auch „benannter Typ“) wird in einer Typdeklaration als Repräsentant eines Typausdrucks definiert, z.B.

  ```
  typedef int t_intpaar[2];
  typedef int** t_indirekter_intzeiger;
  ```

  Der Typbezeichner kann dann wie jeder andere Typrepräsentant für die Typangaben z.B. innerhalb von Variablendefinitionen, Funktionsdefinitionen und anderen Typdefinitionen verwendet werden.

```
t_intpaar intpaar, *intpaarzeiger;
typedef t_intpaar *t_intpaarzeiger;
t_intpaarzeiger f(t_intpaar);
```

- **Klassenbezeichner**

  In einer OO-Sprache wird bei jeder Klassendefinition ein Klassenbezeichner angegeben (zumindest, wenn es sich nicht um eine anonyme Klasse handelt). Dieser Klassenbezeichner repräsentiert ebenso einen Typ wie ein Typbezeichner in C.

- **Schlüsselwort**

  In C/C++/Java werden die Basistypen durch Schlüsselwörter repräsentiert, z.B. `short int` oder `char`, in anderen Sprachen (z.B. Pascal und Abkömmlinge) werden hier vordefinierte Bezeichner verwendet.

2.11.7 Compiler-interne Typrepräsenation

Damit der Compiler Typen verarbeiten kann, muss er sie in geeigneter Weise speichern können. Dies ist wegen der Typkonstruktoren nicht trivial: Der Programmierer kann die Typkonstruktoren beliebig kombinieren und so beliebig komplexe Typen konstruieren. Da komplexe Typen einen hierarchischen Aufbau haben, werden hier komplexe Datenstrukturen, insbesondere Baum- und Listenstrukturen benötigt. Typischerweise wird der Compiler für jeden Typausdruck einen abstrakten Syntaxbaum konstruieren.

2.11.8 Vor- und Nachteile der Typisierung

Vorteile:

- Der Compiler kann bei allen Operatoren, Funktionen und Variablen überprüfen, ob die Verwendung typkonform ist. Dabei wird ein großer Teil der Codierfehler aufgedeckt, die sonst durch mühsames Debuggen zur Laufzeit ermittelt werden müssten. Die Programmentwicklung wird also deutlich sicherer. Vor allem bei grossen Programmsystemen, die aus vielen getrennt übersetzten Modulen bestehen, ist die (Modul-übergreifende) Typprüfung ein außerordentlich wertvolles Hilfsmittel für die Entwickler.
- Mit der Typinformation sind Laufzeitoptimierungen möglich, die Programme werden schneller.

Nachteile:

- In vielen Sprachen ist das Typsystem für gehobene Ansprüche nicht flexibel genug. Insbesondere parametrisierbare Typen sind oft gar nicht oder nur eingeschränkt benutzbar.

  Dies ist kein Nachteil der Typisierung an sich, sondern zeugt schlichtweg von einer gewissen technischen Rückständigkeit dieser Sprachen.

- Durch die Typangaben wird die Programmierung etwas aufwändiger.

- Der Compiler wird komplexer und der Übersetzungsprozess dauert etwas länger. Dies ist i.d.R. vernachlässigbar.

2.12 Variable, Lebensdauer

Das Variablenkonzept prozeduraler Programmiersprachen ist komplex. Um es zu verstehen, betrachten wir einzelne Aspekte zunächst etwas vereinfacht, um sie später zu verfeinern.

- Eine Variable ist ein programmiersprachliches Konzept zur Repräsentantion von Speicherplätzen. Im einfachsten Fall gehört zu einer Variablen im Programmquelltext ein Hauptspeicherplatz während der Programmausführung.

- Eine Variable hat einen Typ, aus dem die Größe des Speicherplatzes, die Codierung der Werte und die erlaubten Operationen hervorgehen.

- Syntaktisch werden Variablen im Quelltext durch Bezeichner vertreten. Ein Variablenbezeichner ist an seiner Verwendungsstelle eine Referenz und wird ggf. implizit dereferenziert.

  Der Bezeichner hat einen wohldefinierten Gültigkeitsbereich.

- Eine Variable hat einen Wert, der sich während der Programmausführung ändern kann. Genauer gesagt kann man zu jedem Zeitpunkt der Programmausführung in dem der Variable zugehörigen Speicherplatz eine Bitsequenz sehen, die mittels der Typinformation als Wert interpretiert werden kann.

  Ist die Variable nicht ordnungsgemäß initialisiert, steht natürlich im Speicherplatz dennoch eine solche Bitsequenz, es geht aus dem Programm nur nicht hervor, welche. Man betrachtet den Wert in diesem Fall als undefiniert.

- Eine Variable kann je nach Programmiersprache unterschiedliche weitere Merkmale haben, so kann in C/C++ beispielsweise das Attribut *volatile* vergeben werden. Es besagt, dass der Speicherplatz nicht nur durch das Programm, sondern auch von anderer Seite verändert werden kann. Der Compiler ist in diesem Fall bei den Maschinencode-Optimierung eingeschränkt.

Dies wird z.B. genutzt, wenn Register eines Hardware-Controllers in den Adressraum eines Prozesses eingebunden werden, damit diese Register im Programm durch Variablen repräsentiert werden können.

Wichtig ist eine genauere Betrachtung der Speicherplatzzuordnung und in diesem Zusammenhang die Klärung des Begriffs „Lebensdauer“ der Variablen.

In modernen Programmiersprachen muss zwischen statischer und dynamischer Speicherplatzverwaltung unterschieden werden:

- Bei statischer Verwaltung steht der Speicherplatz während der gesamten Programmausführung zur Verfügung.
- Der Grundgedanke dynamischer Verwaltung ist, dass ein Speicherplatz nur so lange reserviert werden sollte, wie er auch gebraucht wird. Das ist nicht unbedingt immer die gesamte Ausführungszeit.

  Während der Programmausführung wird also zu einem wohldefinierten Zeitpunkt der Speicherplatz reserviert, steht dann für einen bestimmten Zeitraum zur Verfügung und wird schließlich wieder zur anderweitigen Verwendung wieder freigegeben.

  Diese Zeitspanne heißt „**Lebensdauer**“ der Variablen. Das gängigste Konzept dynamischer Verwaltung ist die Beschränkung der Lebensdauer von Variablen, die im Rahmen eines Funktions- oder Methodenaufrufs benötigt werden, auf die Zeitspanne, die eben dieser Aufruf dauert.

  Ebenfalls bekannt ist die Verwendung von expliziten Hochsprachoperatoren zur Reservierung und Freigabe der Speicherplätze, z.B. *malloc* und *free* in C-Programmen oder *new* und *delete* in C++-Programmen.

Die dynamische Speicherplatzverwaltung erlaubt rekursive Funktions- und Methodenaufrufe, wobei für jede einzelne Rekursionebene ein separater Speicherbereich verfügbar ist. Somit ist der Bezug zwischen Variable und Speicherplatz noch einmal zu überdenken:

```
...
long fakultaet (int n) {
  return n<=1 ? 1 : n*fakultaet(n-1);
}
main() {  cout << fakultaet(10) << endl;  }
```

Die funktionslokale Variable *n* hat in jeder Rekursionebene einen separaten Speicherplatz, in dem jeweils ein anderer Wert gespeichert ist. Wir denken natürlich gerne in den Kategorien des Quelltexts, reden also von *einer* Variablen *n*. Zur Laufzeit sind dieser Variablen aber, abhängig vom Beobachtungszeitpunkt 0, 1,

2 oder noch mehr Speicherplätze zugeordnet. Der Begriff „Lebensdauer“ bezieht auf den Speicherplatz und muss damit für jede Rekursionebene separat betrachtet werden.

In dieser Hinsicht spielt es keine Rolle, dass die betrachtete Variable ein formaler Parameter der Funktion ist. Letzlich handelt es sich dabei doch nur um eine lokale Variable, die bei Funktionsaufruf mit dem aktuellen Parameterwert initialisiert wird.

2.13 Objekt

Die oben für Variablen genannten semantischen Aspekte gelten genauso für Objekte in OO-Sprachen. Die Rolle des Typs übernimmt bei einem Objekt die Klasse. Besondere Betrachtung erfordert die Polymorphie (s. unten).

Es gibt zusätzliche Aspekte, die aus den Programmierlehrveranstaltungen vertraut sind, z.B. dass zu Beginn der Lebensdauer eines Objekts ggf. eine benutzerdefinierte Konstruktormethode und zum Ende der Lebensdauer entsprechend eine Destruktormethode implizit aufgerufen wird.

2.14 Anweisung

Wie wollen den Begriff „Anweisung“ (engl. „statement“) bezogen auf die Quelltextebene für jeden Repräsentanten einer „Laufzeitaktion“ bzw. einer Zustandsänderung benutzen. Eine wesentliche Teilkomponente des damit gemeinten Zustands ist der Adressraum des Prozesses, der sich auf der Quelltextebene in den Variablenwerten bzw. Objektzuständen widerspiegelt.

Es ist ein Grundmerkmal prozeduraler Programmiersprachen, dass die Ausführung als eine Sequenz von zustandsmodifizierenden Anweisungen aufgefasst werden kann. Andere Zustandskomponenten, die durch Anweisungen modifizierbar sind, könnten externe Geräte sein, z.B. ein Bildschirm, der seinen Zustand infolge der Ausführung einer Ausgabeanweisung ändert, oder eine Datei, deren Zustand durch eine Dateioperation modifiziert wird.

2.14.1 Zwielichtige Gestalten

Die Abgrenzung syntaktischer Kategorien wie *Audruck* und *Anweisung* durch Zuordung einer klaren Semantik ist allerdings in vielen Sprachen, insbesondere auch in C++, problematisch. Die Unterscheidung erscheint dann machmal „an den Haaren herbeigezogen“.

Dazu einige Beispiele:

1. In einem C-Quelltext ist

   ```
   3*5+i
   ```

   ohne große Probleme als Ausdruck einzuordnen, da es einen Wert repräsentiert, der ausgewertet werden soll.

2. Ebenso können in einem Pascal-Quelltext die beiden folgenden "Befehle"

   ```
   println('hallo');
   x:=5;
   ```

   klar als Anweisungen klassifiziert werden, da es sich hier um Zustandsmodifikationen handelt: In der ersten Anweisung wird die Standardausgabedatei, in der zweiten der Variablenwert (Hauptspeicher) modifiziert.

3. Problematischer ist eine Wertzuweisung in C, da die Wertzuweisung als Ganzes selbst wieder einen Wert repräsentiert:

   ```
   x=3
   ```

   Die Wertzuweisung verändert den Zustand von *x*, repräsentiert darüber hinaus aber den Wert 3. So praktisch dies für den Programmierer ist, kann er doch dadurch Ketteninitialisierungen wie

   ```
   x=y=z=0
   ```

   unaufwändig formulieren, so schwierig ist jetzt die semantische Einordnung: Anweisung oder Ausdruck? In C/C++ gibt es viele solche ZwitterKonstrukte, so sind beispielsweise `i++` und `++i` als Anweisungen gleichwertig, da die gleiche Zustandsmodifikation erfolgt, während sie als Ausdrücke unterschiedliche Werte repräsentieren.

   Formal wird die Abgrenzung wie folgt gemacht: Eine „Ausdrucksanweisung" ist ein Ausdruck gefolgt von einem Semikolon. Aus jedem Ausdruck lässt sich auf diese Weise eine Anweisung machen. Die durch die Anweisung repräsentierte „Aktion" ist die Auswertung des Ausdrucks. Sinnvoll ist eine solche Aktion natürlich nur, wenn die Auswertung tatsächlich einen Seiteneffekt hat, d.h. eine semantisch relevante Zustandsänderung erfolgt. Dies ist bei Wertzuweisungen, ++-Opertoren und dergleichen der Fall. Rein formal ist aber auch folgendes völlig korrekt:

```
main() {
  int x=17;

  3+x;
  sin(x);
  -25.3;
}
```

Nicht immer sieht man einem Ausdruck an, ob die Auswertung einen Seiteneffekt hat (i.a. ist es auch nicht entscheidbar), nämlich bei Funktions- und Methodenaufrufen: Der im o.g. Beispiel vorkommende Funktionsaufruf

```
sin(x)
```

ist bei erster Betrachtung ein Ausdruck, dessen Auswertung keine Seiteneffekte hat, der ausschließlich als Wertrepräsentant gedacht ist. Andererseits kann aber niemand dem Ausdruck ansehen, ob er seiteneffektfrei ist, schließlich könnte die *sin*-Funktion durchaus Variablen manipulieren, Bildschirmausgaben erzeugen oder gar die Festplatte löschen.

Ein anderes Beispiel sind Objekt-Definitionen in C++. Obwohl grundsätzlich eine Definition dazu dient, dem Compiler die Bezeichnerkategorie, Typinformation und sonstige Attribute bekanntzugeben, kann auch eine „harmlos“ aussehende Definition wie

```
rationale_zahl z;
```

beliebige Seiteneffekte repräsentieren, da dahinter ein (impliziter) Methodenaufruf versteckt ist, nämlich der des Konstruktors.

2.15 Überladung

Ein Bezeichner ist **überladen**, wenn an einer Verwendungsstelle zwei (oder mehr) unterschiedliche Definitionen des Bezeichners gültig sind. Der Compiler muss in diesem Fall der Verwendungsstelle *eindeutig* eine Definition zuordnen können. Genau wie ein Bezeichner kann auch ein Operator überladen sein.

Ein klassisches Beispiel für einen überladenen Operator ist der Additionsoperator „+“, der in den meisten Programmiersprachen sowohl für die Ganzzahladdition als auch für die Gleitkommaaddition verwendet wird, in manchen Sprachen auch für die Konkatenation von Zeichenketten.

In C++ kann der Programmierer selbst Bezeichner und Operatoren überladen, in anderen Sprachen gibt es Überladungen nur bei den vordefinierten Operatoren.

Unabhängig davon ist die Verarbeitung der Verwendungsstelle die gleiche: Der Compiler muss zuerst anhand der Typinformationen nach bestimmten sprachspezifischen Regeln die „richtige“ Definition bestimmen und damit die Überladung auflösen. Die sonstige Verarbeitung erfolgt wie sonst auch.

Beispiel: Beim Additionsoperator ist die Auflösung von den Operandentypen abhängig und erfolgt durch Bestimmung der Operandentypen und Fallunterscheidung:

a) Beide Operanden sind ganzzahlig: Der Operator steht für die Ganzzahladdition.

b) Beide Operanden sind Gleitpunktzahlen: Der Operator steht für die Gleitpunktaddition.

c) Ein Operand ist ganzzahlig, der andere ist eine Gleitpunktzahl. Der Operator steht für die Gleitpunktaddition, bei der Auswertung kommt zur Addition selbst aber noch eine weitere Operation hinzu: Der ganzzahlige Operand muss vor der Addition in einen entsprechenden Gleitpunktwert konvertiert werden. Dies kann zur Übersetzungszeit durch den Compiler selbst erfolgen, z.B. in

   ```
   float x; ...    x = x+3; ...
   ```

   oder erst zur Laufzeit durch entsprechende Maschinenbefehle, z.B. in

   ```
   float x;
   int   i;
   ...
   x = x + i;
   ...
   ```

2.16 Polymorphismus

Polymorph heißt „vielgestaltig“, wobei im Kontext von Programmiersprachen verschiedene Arten solcher „Vielgestaltigkeiten“ auftauchen. Wir begnügen uns mit einer kurzen Betrachtung der Polymorphie im Sinne von OO-Sprachen.

Hier spricht man von Polymorphismus, wenn der Compiler den Typ eines Objekts nicht vollständig bestimmen kann, weil die zugrunde liegende Klasse andere Klassen beerbt, deren Objekte subklassenspezifische Operationen benötigen.

Ein Beispiel:

Eine Klasse *2D-Objekt* hat zwei speziellere Subklassen: *Rechteck* und *Kreis*. Sowohl für *Rechteck*-Objekte als auch für *Kreis*-Objekte sei eine spezifische Methode *zoom* definiert, die die Darstellung des Objekts auf dem Bildschirm in der Grösse verändert.

```
class zwei_D_objekt {     // Basisklasse
  ...
public:
  ...
  virtual void zoom (float faktor);  // Darstellungsgröße ändern
  ...
};

class rechteck : public zwei_D_objekt {
  ...
public:
   void zoom   (float faktor);  // Redefinition für Rechtecke
  ...
};

class kreis : public zwei_D_objekt {
  ...
public:
   void zoom   (float faktor);  // Redefinition für Kreise
  ...
};
```

Für Rechtecke und Kreise existieren nun spezifische *zoom*-Methoden. Betrachten wir nachfolgende Verwendung:

```
main() {
  zwei_D_objekt *z2o_zeiger;
  rechteck      r;
  kreis         k;
  ...

  if ( ... )
     z2o_zeiger = &r;
  else
     z2o_zeiger = &k;

  z2o_zeiger -> zoom(1.8);   // Aufruf der spezifischen Methode!
                             // erst zur Laufzeit bestimmbar
```

. . .

Der Compiler kann die aufzurufende Methode nicht bestimmen, da die Typinformation für das Objekt `*z2o_zeiger` dazu nicht ausreicht. Das Prinzip der „späten Bindung“ (auch „dynamische Bindung“ genannt) verlangt einen Aufruf der subklassenspezifischen Methode. Abhängig vom Wert des Auswahlausdruck innerhalb der *if*-Anweisung, käme die rechteck- oder die kreisspezifische Zoom-Methode in Frage.

Um das Problem zur Laufzeit lösen zu können, müssen die Objekte ihren (Sub-) Typ mit sich tragen. Der Compiler wird bei der Codeerzeugung eine Tabelle generieren, in der jeder zur Laufzeit möglichen Ausprägung die Adresse der zugehörigen Methode zugeordnet wird. Der aktuelle Typ des Objekts wird zur Laufzeit benutzt, um in dieser Tabelle die Adresse der aufzurufenden Methode zu ermitteln. Der Rest ist analog zu einem Funktionsaufruf zu behandeln.

3 Einführung in Formale Sprachen

3.1 Mathematische Grundlagen

Definition

Sei A ein *Alphabet*, d.h. eine endliche Menge von Symbolen.

A^+ ist die *Menge aller Wörter über A*, die aus endlicher Aneinanderreihung (Konkatenation) von Symbolen aus A gebildet werden können.

ε bezeichne das *leere Wort*.

$A^* = A^+ \cup \{\varepsilon\}$

Beispiel:

Sei $A = \{a, B, \$\}^*$

aaaB$BB ist ein Wort über A, d.h. ein Element von A^*

Für u, v $\in A^*$ ist auch die Konkatenation uv $\in A^*$, z.B. u=$$, v=aBBa, uv=$$aBBa.
ε ist das neutrale Element der Konkatenation:

εw = wε = w ,

für alle w $\in A^*$

Anmerkung

Eine Algebra mit einer Menge M und einer assoziativen Verknüpfung $\circ$ auf M heißt Halbgruppe. Falls darüber hinaus noch ein neutrales Element bezüglich $\circ$ existiert, nennt man die Algebra ein Monoid.

Betrachten Sie die Menge aller (nichtleeren) Wörter über einem Alphabet A: A^+ bzw. A^* .

Beides sind Halbgruppen, denn die Konkatenation ist eine assoziative Verknüpfung,

z.B. (ab)c = a(bc) = abc.

A^* enthält mit dem leeren Wort ε darüberhinaus noch ein neutrales Element, A^* ist also ein Monoid. (A^* ist übrigens keine Gruppe, denn bezügl. der Konkatenation existieren keine inversen Elemente).

A^+ ist die freie von A frei erzeugte Halbgruppe, A^* das freie Monoid über A.

3.2 Kontextfreie Grammatiken

Grammatiken dienen zur endlichen Beschreibung formaler Sprachen, etwa von Programmiersprachen. Für den Übersetzerbau sind insbesondere kontextfreie Grammatiken wichtig.

Definition
Eine *kontextfreie Grammatik* besteht aus

- einer endlichen Menge T von *Terminalsymbolen* (oder Tokens)
- einer endlichen Menge N von *Nonterminalsymbolen* (oder Variablen), $N \cap T = \{\}$
- einer endlichen Menge P von Ableitungsregeln (oder Produktionen) der Form

  $\alpha \to w$, wobei $a \in N, w \in (N \cup T)^*$

  a heißt linke Seite, w rechte Seite der Regel.
- einem Startsymbol S $\in$ N.

Beispiel:
G = (T, N, P, A) mit T = { a,b,c } , N = { A,B } , P = $\{A \to aAb, A \to B, B \to c\}$
und Startsymbol A ist eine kontextfreie Grammatik.

Anmerkung
Kontextfreie Grammatiken sind insofern gegenüber allgemeineren Grammatiken eingeschränkt, als die linke Regelseite immer aus genau einem Nonterminal besteht. Produktionen, wie Ba$\to$aB oder abcd$\to \varepsilon$ sind nicht erlaubt.

Beispiel für eine „richtige“ Grammatik:

(in Anlehnung an die Pascal-Programmstruktur)

Programm	$\rightarrow$	Programmkopf Deklarationen Anweisung **.**
Programmkopf	$\rightarrow$	"`program`" Id OptProgParListe ";"
OptProgParListe	$\rightarrow$	"`(`" IdListe "`)`"
OptProgParListe	$\rightarrow$	ϵ
IdListe	$\rightarrow$	Id
IdListe	$\rightarrow$	Id "**,**" IdListe
Deklarationen	$\rightarrow$	OptConstDekl OptTypDekl OptVarDekl OptUpDekl
OptConstDekl	$\rightarrow$	ϵ
OptConstDekl	$\rightarrow$	"`const`" ConstDeklListe
	...	
VarDekl	$\rightarrow$	IdListe "**:**" Typ
	...	

3.2.1 Ableitbarkeit

Zur Beschreibung der Syntax formaler Sprachen gibt man in der Regel eine Grammatik an und definiert die Sprache dann als Menge aller Wörter, die sich aus der Grammatik erzeugen (oder ableiten) lassen. Was heißt das aber genau ?

Definition

Sei G = (T,N,P,S) eine Grammatik, u, x, y, w $\in$ (T $\cup$ N)*.

Aus *uxw* ist *uyw* **direkt ableitbar** (Notation: $uxw \rightarrow_G uyw$), wenn $(x \rightarrow y) \in$ P.

Aus *uxw* ist *uyw* **ableitbar** (Notation: $uxw \stackrel{*}{\Rightarrow}_G uyw$), wenn für ein $n > 0$ Wörter $v_0, \ldots, v_n$ existieren, so dass

$uxw = uv_0w \rightarrow_G uv_1w \cdots \rightarrow_G uv_nw = uyw$

(Die Ableitbarkeitsrelation ist demzufolge der transitive und reflexive Abschluss der direkten Ableitbarkeit.)

Definition

Die von G erzeugte Sprache L(G) ist die Menge aller aus dem Startsymbol ableitbaren Wörter, die nur aus Terminalsymbolen bestehen:

L(G) = $\{w \in T^* \mid S \stackrel{*}{\Rightarrow}_G w\}$

Anmerkung

Man beachte, dass mit Grammatiken dem Informatiker also eine endliche Beschreibungsmöglichkeit für unendliche Mengen - insbesondere die Menge aller Programme einer bestimmten Programmiersprache - zur Verfügung steht.

Anmerkung

Es gibt Sprachen, die sich nicht durch kontextfreie Grammatiken, wohl aber mittels allgemeinerer Grammatiken definieren lassen,

z.B. $\{a^n b^n c^n \mid n \in N\}$

Definition

Sei G = (N, T, P, S) eine Grammatik. Ein Wort $w \in (N \cup T)^*$, das sich aus S ableiten lässt, $S \stackrel{*}{\Rightarrow}_G w$, heißt **Satzform** (zu G).

Falls in einer Satzform mehrere Nonterminal-Zeichen vorkommen, ist die Reihenfolge der Ableitungsschritte nicht mehr eindeutig festgelegt. Wir definieren zwei kanonische Reihenfolgen für die Ableitungschritte:

Definition

Eine Ableitung, bei der in jedem Ersetzungsschritt das äußerst linke (rechte) Nonterminal-Zeichen ersetzt wird, heißt **Linksableitung** (bzw. **Rechtsableitung**).

Eine Darstellung, die von der Reihenfolge der Ableitungsschritte abstrahiert ist der Ableitungsbaum (auch Syntaxbaum oder Zerlegungsbaum).

Definition

Ein **Ableitungsbaum** ist ein geordneter Baum, der aus einer Ableitung A eines Worts w bezüglich einer Grammatik G = (N, T, P, S) wie folgt konstruiert wird:

- Der Wurzelknoten wird mit dem Startsymbol S markiert
- Zu jeder in der Ableitung angewandten Regel
  $X \to x_1 ... x_n$ werden dem mit X markierten Knoten im Baum, der das abgeleitete Nonterminal repräsentiert, n neue, mit $x_1, ..., x_n$ markierte Knoten als Nachfolgeknoten zugeordnet.

Der Baum hat folgende Eigenschaften:

- Die inneren Knoten des Baums sind mit Nonterminalsymbolen aus N markiert.
- Die Blätter sind mit Terminalsymbolen aus T markiert.

- Die Blätter ergeben von links nach rechts angeordnet das abgeleitete Wort w.

Beispiel:

Wir betrachten die Grammatik

G = (T, N, P, A) mit T = { a,b,c } , N = { A,B,C }.

Die Regelmenge P enthalte folgende Regeln:

1. A$\rightarrow$ aBC
2. B$\rightarrow$ Bb
3. B$\rightarrow \epsilon$
4. C$\rightarrow$ c

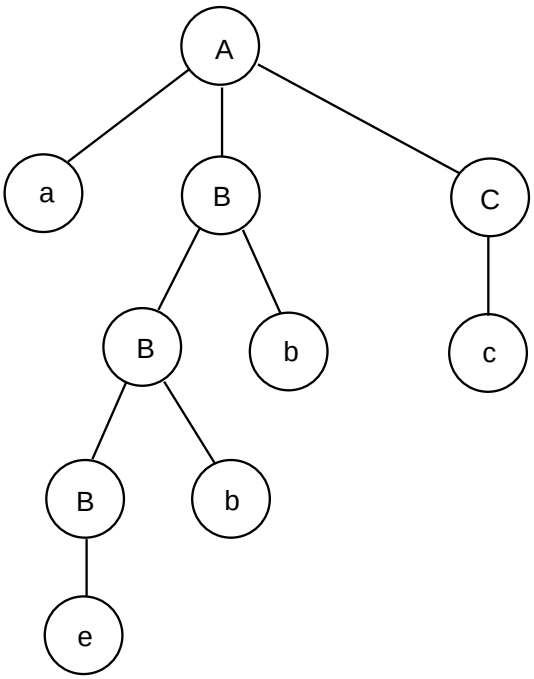
Das Wort w=abbc liegt in L(G). Die Linksableitung für w ist

$A \stackrel{1}{\Rightarrow} aBC \stackrel{2}{\Rightarrow} aBbC \stackrel{2}{\Rightarrow} aBbbC \stackrel{3}{\Rightarrow} abbC \stackrel{4}{\Rightarrow} abbc$

Die Rechtsableitung:

$A \stackrel{1}{\Rightarrow} aBC \stackrel{4}{\Rightarrow} aBc \stackrel{2}{\Rightarrow} aBbc \stackrel{2}{\Rightarrow} aBbbc \stackrel{3}{\Rightarrow} abbc$

Der (eindeutig bestimmte) Ableitungsbaum:



3.2.2 Notationsvarianten für Grammatiken

Für Grammatik-Regeln gibt es mehrere gebräuchliche Schreibweisen, die insbesondere der Tatsache Rechnung tragen, dass man Grammatiken mit dem Rechner verarbeitet, auf gebräuchlichen Tastaturen aber weder „$\rightarrow$“ noch „ϵ“-Tasten zu finden sind.

- Um linke und rechte Seite einer Regel zu trennen wird statt $\rightarrow$ auch ”::=” oder (vgl. *yacc/bison*) einfach nur ”:”verwendet.
- Oft gibt es für ein Nonterminalsymbol *X* mehrere syntaktische Varianten $V_1 \ldots V_k$. Statt für jede Variante eine eigene Regel zu formulieren

  $$\begin{array}{lcl} \mathsf{X} & \rightarrow & V_1 \\ & \ldots & \\ \mathsf{X} & \rightarrow & V_k \end{array}$$

  schreibt man auch

  $\mathsf{X} \rightarrow V_1 \mid \cdots \mid V_k$
- Zur Abgrenzung zwischen Terminal- und Nonterminalsymbolen werden einzelne Terminalsymbole auf der rechten Seite einer Grammatik in einfache Apostrophe eingeschlossen.

Beispiel:

Die oben definierte Grammatik

G = (T, N, P, A) mit T = { a,b,c } , N = { A,B,C } mit den Regeln

A$\rightarrow$ aBC
B$\rightarrow$ Bb
B$\rightarrow \epsilon$
C$\rightarrow$ c

sieht in bison-Notation so aus:

```
A : 'a' B C ;
B : B 'b' | /* epsilon */ ;
C : 'c' ;
```

„`/* epsilon */`“ ist dabei nur ein Kommentar (C-Syntax), der für den Leser klar macht, dass B bei dieser Regelalternative ganz entfällt.

3.3 EBNF

Kontextfreie Grammatiken werden auch als BNF-Notation (Backus-Naur-Form) bezeichnet.

EBNF steht für „erweiterte Backus-Naur-Form“. EBNF bietet gegenüber der einfacheren BNF oder der „klassischen“ Notation für kontextfreie Grammatiken einige Schreibabkürzungen. Insbesondere gibt es einen Wiederholungsoperator ($\{\ldots\}$), der die umständlichere rekursive Definition von Listen unnötig macht.

Eine EBNF-Beschreibung besteht aus Regeln der Form

Symbol **::=** *rechte Seite*

Auf der rechten Seite können Terminalzeichenketten stehen, die in Apostrophe eingeschlossen sind, z.B. `"hallo"` oder `">="`, (Nonterminal-)Symbole oder komplexe Ausdrücke.

Für die Bildung komplexer Ausdrücke verwendet EBNF diverse Meta-Zeichen, z.B. eckige, geschweifte und runde Klammern. Falls diese Zeichen auch in der zu beschreibenden Sprache vorkommen, sind sie dort in Apostrophe einzuschließen, um Verwechslungen zu vermeiden.

- Für die Auslassung eines Elements verwenden wir **EMPTY**.
- die rechte Seite kann mehrere durch den Alternativ-Operator `|` getrennte syntaktische Varianten enthalten, z.B.

```
declaration ::= label_decl | const_decl | type_decl |
                var_decl | function_decl | procedure_decl
```

  Die Präzedenz dieses Operators ist schwächer als die der Konkatenation, ggf. ist zu klammern :

```
name ::=  "HANS"  | "PETER" "MAIER"
```

  ist äquivalent zu

```
name ::=  "HANS"  | ( "PETER" "MAIER" )
```

- auf der rechten Seite kann eine beliebige (auch eine 0-fache) Wiederholung eines Elements durch Einschließen dieses Elements in geschweifte Klammern spezifiziert werden

```
function_call  ::= identifier  ( parameter_list | EMPTY )
parameter_list ::= "(" expression  { "," expression } ")"
```

- für eine Beschränkung der Wiederholungen nach unten durch *min* oder nach oben durch *max* verwendet man

  $$\{ \quad \ldots \quad \}_{min}^{max}$$

- statt $\{ \quad \ldots \quad \}_0^1$ schließt man optionale Elemente in eckige Klammern ein.

3.4 Transformation von EBNF in Grammatiken

Die Transformation einer EBNF-Syntax in eine kontextfreie Grammatik lässt sich sowohl von Hand als auch per Programm ohne große Probleme bewerkstelligen.

Im wesentlichen wird man dabei folgende Ersetzungen benötigen:

- Einen Listenbestandteil einer rechten Regelseite der Form $\{Listenelement\}$ ersetzt man durch ein neues Nonterminalsymbol, hier z.B. *Liste*, und definiert rekursiv:

  $Liste \rightarrow ListenelementListe \mid \epsilon$ oder

  $Liste \rightarrow ListeListenelement \mid \epsilon$

- Für eine optionale syntaktische Struktur der Form $[X]$ wird gleichfalls ein neues Nonterminalsymbol definiert, hier z.B. *OptionalesX*. Die Regeln lauten dann

  $OptionalesX \rightarrow X \mid \epsilon$

- Auch für geklammerte komplexe Bestandteile der rechten Regelseite wird ein neues Nonterminalsymbol definiert, dessen Definition dann abhängig ist von den in den Klammern stehenden Operatoren.

Das EBNF-Beispiel von oben

```
function_call  ::= identifier  ( parameter_list | EMPTY )
parameter_list ::= "(" expression  { "," expression } ")"
```

kann in folgende Grammatik transformiert werden:

function_call	$\rightarrow$	*identifier OptParameterList*
OptParameterList	$\rightarrow$	*parameter_list* $\mid \epsilon$
parameter_list	$\rightarrow$	*"(" expression Expressions ")"*
Expressions	$\rightarrow$	*"," expression Expressions* $\mid \epsilon$

Listing 3.1 EBNF-Beispiel: Syntax von Oberon-0

```
ident = letter {letter|digit}.
integer = digit {digit}.

selector = { "." ident |"[" expression "]"}.
number = integer.
factor = ident selector | number
         | "(" expression ")" | "~" factor.
term = factor {("*" | "DIV" | "MOD" | "&") factor}.
SimpleExpression = ["+"|"-"] term {("+"|"-"|"OR") term}.
expression = SimpleExpression
             ("=" | "#" | "<" | "<=" | ">" | ">=")
             SimpleExpression.
assignment = ident Selector ":=" expression.
ActualParameters = "(" [expression {"," expression }]")".
ProcedureCall = ident [ActualParameters].
IfStatement = "IF" expression "THEN" StatementSequence
             {"ELSIF" expression "THEN" StatementSequence}
             [ELSE StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence "END".
statement = [ assignment | ProcedureCall
            | IfStatement | WhileStatement ].
StatementSequence = statement {";" statement }.

IdentList = ident {"," ident}.
ArrayType = "ARRAY" expression "OF" type.
FieldList = [IdentList ":" type].
RecordType = "RECORD" FieldList {";" FieldList} "END".
type = ident | ArrayType | RecordType.
FPSection = ["VAR"] IdentList ":" type.
FormalParameters = "(" [FPSection {";" FPSection}] ")".
ProcedureHeading = "PROCEDURE" ident [FormalParameters].
ProcedureBody = declarations
                ["BEGIN" StatementSequence]
                "END" ident.
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody.
declarations = ["CONST" {ident "=" expression ";"}]
               ["TYPE"  {ident "=" type ";"}]
               ["VAR" {IdentList ":" type ";"}]
               {ProcedureDeclaration ";"}.
module = "MODULE" ident ";" declarations
         [ "BEGIN" StatementSequence ] "END" ident ".
```

3.5 Formale Sprachen und Compilerbau

Die Theorie der Formalen Sprachen, die Automaten- und die Komplexitätstheorie haben entscheidend die Compilerbau-Theorie geprägt.

Innerhalb der Theorie der Formalen Sprachen klassifiziert die Chomsky-Hierarchie verschiedene Komplexitätsstufen für Grammatiken und die dadurch definierten Sprachen:

- Bei Chomsky-0-Grammatiken gibt es keine Restriktionen für die Ableitungsregeln. Ob ein Wort w mit den Regeln der Grammatik aus dem Startsymbol ableitbar ist, ist **nicht entscheidbar**.
- Chomsky-1-Grammatiken nennt man auch **kontextsensitive** Grammatiken. Für diese Grammatiken gibt es bestimmte Restriktionen für die Ableitungsregeln und die theoretische Aussage, dass die Zugehörigkeit eines Wort zu einer Chomsky-1-Sprache entschieden werden kann.

  Für den Compilerbau sind Chomsky-1-Grammatiken zu komplex.
- Chomsky-2-Grammatiken sind die **kontexfreien** Grammatiken.

  Die kontexfreien Sprachen eignen sich zur Spezifikation der hierarchischen Programmstruktur für alle gängigen Programmiersprachen. Zur Syntaxanalyse können Automaten mit einem Stack (Kellermaschinen) verwendet werden.
- Chomsky-3-Grammatiken nennt man auch **reguläre** Grammatiken. Eine reguläre Grammatik $G = (T, N, S, P)$ ist entweder **linksregulär** mit Regeln der Form $X \rightarrow w, X \in N, w \in T^*$ oder $X \rightarrow Yw, X, Y \in N, w \in T^+$, oder **rechtsregulär** mit Regeln der Form $X \rightarrow w, X \in N, w \in T^*$ oder $X \rightarrow Yw, X, Y \in N, w \in T^+$.

  Die regulären Sprachen eignen sich zur Spezifikation der lexikalischen Ebene der Sprachsyntax. Zur Erkennung können Endliche Automaten verwendet werden, ein Stack ist im Gegensatz zu den Chomsky-2-Grammatiken nicht notwendig

Eine wesentliche Rolle spielen im Compilerbau die kontextfreien Sprachen, deren Syntax durch kontextfreie Grammatiken beschrieben werden kann. Dies mag bei näherem Hinsehen verwundern, denn Kontextabhängigkeiten gibt es in den gebräuchlichen Programmiersprachen genügend viele. Ein Beispiel:

In C gilt das Prinzip „declare before use“: Jeder *Bezeichner* muss vor seiner ersten Verwendung deklariert werden (vgl. 4.1, S. 50).

Diese Regel ist syntaktischer Natur, geht aber aus keiner Grammatik für die Sprache hervor. Warum nicht ?

Der Spezifikationsmechanismus der kontextfreien Grammatiken ist dafür nicht mächtig genug. Nur mit komplizierten kontextabhängigen Regeln ließe sich das „declare before use“-Prinzip spezifizieren. Für kontextabhängige Grammatiken gibt es aber keine praktikablen Analyseverfahren (zu kompliziert, zu langsam). Der einzige ernsthafte Versuch, kontextabhängige Grammatiken für eine reale Programmiersprache zu verwenden, die ALGOL68-Definition, war nicht gerade erfolgreich.

Daher beschreibt man die Syntax einer Programmiersprache wie folgt: In einer kontextfreien Grammatik beschreibt man den Großteil der syntaktischen Anforderungen. Alles, was sich nicht mit kontextfreien Regeln beschreiben lässt, wird außerhalb der Grammatik, quasi als „Nebenabrede“ (meist informal) ergänzt. Manchmal werden auch Dinge, die sich an für sich kontextfrei beschreiben liessen, aus der Grammatik herausgenommen, um diese nicht zu komplex zu machen.

4 Lexikalische Analyse (Scanner)

Die Lexikalische Analyse ist ein Teil der Syntaxanalyse. Ziel ist die Erkennung von lexikalischen Quelltextbestandteilen wie Literalen, Schlüsselwörtern, Bezeichnern, Operatoren, Kommentaren.

Wir nennen diese syntaktischen „Grundbausteine“ eines Programms nachfolgend **Tokens**. Ein Tokenerkennungsprogramm heißt auch **Scanner**.

Tokens haben eine besonders einfache Syntax, die mit **regulären Ausdrücken** beschrieben wird. Diese sind kompakter als Grammatiken.

Der Grund für die Abgrenzung zwischen Lexikalischer Analyse und Syntaxanalyse liegt darin, dass für die Tokens einfachere und effizientere Analyseverfahren eingesetzt werden.

4.1 Lexikalische Elemente einer Programmiersprache

In höheren Programmiersprachen muss die Lexikalische Analyse typischerweise folgende Tokens erkennen:

- **Bezeichner** (engl.: Identifier)

  Ein Bezeichner ist ein üblicherweise vom Programmierer definierter symbolischer Name. Er wird nach festen Regeln gebildet, z.B. Buchstabe oder Unterstrich gefolgt von beliebigen Buchstaben, Ziffern oder Unterstrichen.

  Der Bezeichner kann im Programmtext sehr unterschiedliche Dinge repräsentieren, z.B.

  - Variablen
  - Konstanten
  - Typen
  - Funktionen
  - Sprungziele
  - Klassen

  In vielen Programmiersprachen muss ein Bezeichner vor der ersten Verwendung **deklariert** werden.

- **Operator**

  Ein Operator repräsentiert in kompakter Schreibweise eine häufig benötigte „Operation“. Syntaktisch besteht er meist nur aus einem oder zwei Sonderzeichen, z.B. `+`, `+=`, `->`, `.`, `:=`.

  In vielen Sprachen stehen Operatoren immer für vordefinierte Funktionen, in anderen (z.B. C++) können Sie auch Programmierer-definierte Funktionen repräsentieren.

  - **Präfix-Operatoren** werden ihren Operanden vorangestellt, z.B. `- x`
  - **Postfix-Operatoren** werden hinter ihre Operanden gestellt, z.B. `p ++`
  - **Infix-Operatoren** stehen zwischen ihren Operanden, z.B. `a += b`
  - **Mixfix-Operatoren** nennt man andere syntaktische Formen, z.B. in C der Index-Operator `[]`, der eine Feldkomponente selektiert, oder der `? ... :`-Operator

- **Literal**

  Ein Literal ist eine vordefinierte spezielle Schreibweise zur Repräsentation eines bestimmten Wertes, z.B. `-123e5`, `0Xff`, `'x'`, `"hallo\n"`

- **Kommentar**

  Ein Kommentar mag aus der Sicht seines Verfassers oder seines Lesers ein langer, komplexer, strukturierter (und wichtiger) Programmbestandteil sein.

  Vom Compiler wird er allerdings i.d.R. nicht weiter analysiert (sondern meist schlichtweg ignoriert) und daher als Token behandelt.

- **Schlüsselwort**

  Schlüsselwörter sind von der Sprache fest vorgegebene Wörter, die dem Compiler die Erkennung bestimmter Strukturen ermöglichen (z.B. **if**, **repeat**, **typedef**).

  Es gibt Sprachen mit extrem vielen Schlüsselwörtern (z.B. COBOL) und Sprachen völlig ohne Schlüsselwörter (z.B. LISP).

  Schlüsselwörter sind in der Regel **reserviert**, es ist also verboten einen syntaktisch identischen Bezeichner zu verwenden.

  In PL/I sind sie nicht reserviert, was einer einfachen Syntaxanalyse nicht gerade dienlich ist, vor allem aber nicht der Lesbarkeit von Programmen:

```
IF IF=THEN THEN THEN:=IF ELSE ELSE:=THEN ...
```

4.2 Spezifikation der Token-Syntax

Zur Spezifikation der Token-Syntax ist EBNF geeignet. Es gilt (ohne Beweis): **Eine Sprache ist genau dann regulär, wenn man sie durch eine einzige nichtrekursive EBNF-Regel beschreiben kann.**

Compilerbau-Werkzeuge und andere UNIX-Tools verarbeiten statt EBNF-Regeln allerdings **reguläre Ausdrücke**. Diese Ausdrücke sind im Prinzip zu den nichtrekursiven EBNF-Regeln äquivalent. In der UNIX-Notation gibt es allerdings zusätzliche Operatoren, die die Syntaxbeschreibung noch weiter vereinfachen.

Nachfolgend werden reguläre Ausdrücke und reguläre Sprachen formal definiert.

4.2.1 Reguläre Sprachen

Operationen für Sprachen

Seien L, L_1 und L_2 Sprachen. Wir definieren vier Operationen für Sprachen:

1. **Vereinigung**
   $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ oder } w \in L_2\}$
2. **Konkatenation**
   $L_1 L_2 = \{w \mid w = uv, u \in L_1, v \in L_2\}$
3. **Kleenesche Hülle**
   $L^* = \bigcup_{i=0}^{\infty} L^i$
4. **positive Hülle**
   $L^+ = \bigcup_{i=1}^{\infty} L^i$

Anmerkung

$L^0 = \{\epsilon\}, L^1 = L, L^2 = LL$ usw.

L^* besteht aus ϵ und den Wörtern, die sich in Bestandteile aus L zerlegen lassen.

Zu L^+ gehört ϵ nur dann, wenn es auch zu L gehört.

Es gilt: $L^* = L^+ \cup \{\epsilon\}$

Beispiel:

$L_1 = \{AA, AAA, B\}$
$L_2 = \{cc, dadada\}$
$L_1 \cup L_2 = \{AA, AAA, B, cc, dadada\}$
$L_1 L_2 = \{AAcc, AAdadada, AAAcc, AAAdadada, Bcc, Bdadada\}$
$L_2^2 = L_2 L_2 = \{cccc, ccdadada, dadadacc, dadadadadada\}$

Sprachen, die aus endlichen Wortmengen nur mittels dieser Operationen konstruiert werden, nennt man reguläre Sprachen. Sie sind besonders einfach strukturiert und eignen sich für die Spezifikation der lexikalischen Ebene einer Programmiersprache.

4.2.2 Reguläre Ausdrücke

Mit den oben definierten Sprachoperationen können reguläre Ausdrücke als formale Beschreibungsmethode für die lexikalische Ebene der Syntax von Programmiersprachen (und andere Zwecke) definiert werden.

Definition
Sei A ein Alphabet. Die Menge der regulären Ausdrücke über A ist induktiv definiert. Jeder reguläre Ausdruck *r* repräsentiert eine Sprache $L(r) \subseteq A^*$.

1. Für jedes Symbol $a \in A$ ist a ein regulärer Ausdruck. $L(a) = \{a\}$
2. ϵ ist ein regulärer Ausdruck. $L(\epsilon) = \{\epsilon\}$
3. Seien r_1 und r_2 reguläre Ausdrücke (über A). Dann ist $r_1 r_2$ ein regulärer Ausdruck. $L(r_1 r_2) = L(r_1)L(r_2)$
4. Seien r_1 und r_2 reguläre Ausdrücke. Dann ist $r_1 \mid r_2$ ein regulärer Ausdruck. $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$
5. Seien r ein regulärer Ausdruck. Dann ist r^* ein regulärer Ausdruck. $L(r^*) = L(r)^*$

Definition
Seien *r* und *s* reguläre Ausdrücke über A. *r* und *s* heißen **äquivalent** (Notation: *r=s*), genau dann, wenn $L(r) = L(s)$.

Notation

Um eine eindeutige Operator-Operanden-Zuordnung zu ermöglichen, lassen sich reguläre Ausdrücke klammern. Für ungeklammerte Ausdrücke wird vereinbart:

- $*$ ist linksassoziativ und hat höchste Präzedenz
- Die Konkatenation ist linksassoziativ und hat zweithöchste Präzedenz
- $|$ ist linksassoziativ und hat niedrigste Präzedenz

Beispiel:

Das Alphabet: $A = \{a, b\}$

$r = (a \mid b)^*abb$ ist ein regulärer Ausdruck über A. *r* ist nicht vollständig geklammert, wegen der Linksassoziativität der Konkatenation entspricht dies einer Klammerung „von links“:

$(a \mid b)^*abb = ((((a \mid b)^*)a)b)b$

Welche Sprache repräsentiert **r**? Wir betrachten dazu einige Teilausdrücke:

$L(a \mid b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$

$L((a \mid b)^*) = (L(a \mid b))^* = \{a, b\}^*$
$= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, baa, abb, bab, bba, bbb, aaaa \dots\}$

$L(((((a \mid b)^*)a)b)b) = \{a, b\}^*\{a\}\{b\}\{b\}$

Die Sprache besteht aus allen Wörtern über dem Alphabet A, die mit *abb* enden.

4.2.3 Reguläre Ausdrücke – UNIX-Notation

Für den praktischen Umgang mit Programmen, die reguläre Ausdrücke verarbeiten, sind weitere Operatoren nützlich, die eine kompaktere Notation ermöglichen. (Daneben gibt es auch noch das Problem mit der fehlenden ϵ-Taste an der Rechnertastatur!)

Wir benutzen die in UNIX-Systemen übliche Schreibweise, wie sie vom Scannergenerator *flex*, aber (zumindest annähernd) auch von einer Menge anderer Programme (*grep*, *egrep*, *sed*, *emacs* usw.) benutzt wird. Man beachte, dass in vielen Betriebssystemen auch eine POSIX-Programmierschnittstelle zur Verwendung dieser regulären Ausdrücke verfügbar ist, vgl. *regcomp*, *regexec* u.a.

Die Notation

Man unterscheidet Sonderzeichen und normale Zeichen. Sonderzeichen sind z.B. : `* + [ ] ? ^ ( ) . $`

- ein „normales“ Zeichen steht für sich selbst
- **.** steht für ein beliebiges Zeichen außer `'\n'`
- Falls ein Sonderzeichen nicht als solches interpretiert werden soll, ist ein „\“ voranzustellen
- Auch in Apostrophe eingeschlossene Strings werden verbatim interpretiert.
- Klammerung erfolgt durch (und)
- Konkatenation zweier reg. Ausdrücke erfolgt ohne expliziten Operator
- Alternativen werden mittels | gebildet
- ein nachgestellter ***** steht für beliebige Wiederholung
- ein nachgestelltes **+** steht für nichtleere Wiederholung
- ein nachgestelltes **?** bezeichnet einen optionalen Anteil
- `^` am Anfang eines regulären Ausdrucks steht für Zeilenanfang
- **$** am Ende eines regulären Ausdrucks steht für Zeilenende
- eine **Zeichenklasse** steht für ein Zeichen.

  Sie kann durch **Zeichen-Aufzählung** $x_1 x_2 \ldots x_n$ und **Bereichsangaben** $x_1 - x_n$ gebildet werden:

  - $x_1 - x_n$ steht für ein Zeichen aus dem Bereich, z.B. `[0-9]`
  - $x_1 x_2 \ldots x_n$ steht für ein Zeichen aus der Menge der angegebenen Zeichen, z.B. `[abcx]`
  - beide Schreibweisen können kombiniert werden z.B. `[0-9a-zA-Z_]`
  - Eine `^`-Zeichen am Anfang einer Zeichenklasse `[^...]` spezifiziert die komplementäre Zeichenmenge, z.B. steht `[^0-9]` für ein beliebiges Zeichen außer einer Ziffer

Beispiele:

a) Alle mit kleinem „a“ beginnenden Zeichenketten: `a.*`

b) Alle nichtleeren Dezimalziffernfolgen: `[0-9]+`

c) Alle Wörter, die aus genau 3 Zeichen bestehen und nicht mit einer Ziffer enden: `..[^0-9]`

d) Pascal-Bezeichner = `[A-Za-z][A-Za-z0-9]*`

e) C-Float-Literale = `-?[0-9]+((\.[0-9]+)|((\.[0-9]+)?[eE]-?[0-9]+)`

C-Float-Literale bestehen aus

- einem „Vorkomma-Anteil“ (ggf. mit Minuszeichen) (Syntax: `-?[0-9]+`)
- einem optionalen Nachkomma-Anteil (Syntax: `\.[0-9]+`)
- und einem ebenfalss optionalen Exponenten-Anteil (Syntax: `[eE]-?[0-9]+`).

Dabei ist zu beachten, dass entweder der Nachkomma-Anteil oder der Exponent vorhanden sein muss. Wenn beides fehlt, liegt eine Integer-Konstante vor. Daher ist die folgende Spezifikation nicht korrekt:

`-?[0-9]+(\.[0-9]+)?([eE]-?[0-9]+)?`

(Viel komplizierter wird es in der Praxis nur selten !)

4.3 Scanner-Implementierung

Scanner werden i.d.R. entweder

- „ad hoc“ programmiert oder
- mittels Scanner-Generator automatisch erzeugt

Systematische Scanner-Implementierung (im Gegensatz zur „ad hoc“-Vorgehensweise) kann basierend auf endlichen Akzeptoren nach dem im Abschnitt 4.4, S. 61 erklärten Schritten durchgeführt werden. Wegen der Größe der Automaten ist dies allerdings sehr aufwändig.

Alle Schritte der systematischen Scanner-Implementierung sind voll automatisierbar. Wenn man diese Schritte programmiert, hat man einen Scanner-Generator implementiert.

Betrachten wir zunächst „ad hoc“-Verfahren.

Folgende Punkte sind zu beachten:

- Typischerweise wird der Scanner als Funktion implementiert, die vom Parser aufgerufen wird, um das nächste im Quelltext stehende Token zu erkennen.

Als Resultat muss der Scanner zumindest eine eindeutige Token-Identifikation liefern. Bei Operatoren und Schlüsselwörtern genügt diese Information dem Parser, die konkrete Zeichenfolge, aus denen die Tokens bestehen ist irrelevant.

Hat der Scanner einen Bezeichner gefunden, genügt die Aussage, dass es sich um einen solchen handelt, natürlich nicht. Hier wird der Bezeichner selbst benötigt. Das gleiche gilt für Literale aller Art.

Ein von *bison* aufgerufener Scanner erwartet die Token-Klassifikation (Typ: int) als Funktionsresultat. Die Zeichenkette, aus der das Token besteht, muss in der globalen Variable *yytext* stehen und deren Länge in *yylen*.

- Der Scanner muss sehr effizient programmiert werden, da große Programme aus hunderttausenden von Tokens bestehen können und entsprechend viele Scanner-Aufrufe im Lauf einer Compilierung erfolgen. Insbesondere darf er pro eingelesenem Zeichen im Durchschnitt nicht allzuviel Bearbeitungszeit benötigen.

- Es ist unbedingt darauf zu achten, das beim Einlesen des Quelltexts von einer Plattendatei mit Pufferung gearbeitet wird und nicht pro eingelesenem Zeichen ein physikalischer Plattenzugriff erfolgt.

  Bei Verwendung der „Streams“-Funktion der Standard-C-Bibliothek (*getchar*, *fgetc*, *fread* usw.) ist Pufferung verfügbar (ggf. mit *setbuf*, *setvbuf* einstellen).

- In der Regel wird der Scanner beim Aufruf das erste Zeichen des nächsten Tokens lesen. Anhand dieses Zeichens kann er eine Fallunterscheidung treffen zwischen Operatoren, Literalen, Bezeichnern oder Schlüsselwörtern usw. In jedem der Fälle wird dann das Token vollständig gelesen.

  Man beachte, dass manche Operatoren mit dem gleichen Zeichen beginnen.

- Zur Erkennung des Token-Endes ist oft eine Vorausschau erforderlich.

  Um das Ende eines Bezeichners zu erkennen, muss der Scanner das Zeichen hinter dem Bezeichner gelesen haben, also das erste Zeichen nach dem Bezeichner-Anfang, das nach den Syntaxregeln kein Bezeichnerbestandteil mehr sein darf (z.B. Space, Newline, Operator).

  Für viele Sprachen genügt eine Vorausschau um ein Zeichen.

  Am einfachsten wird die Einleseschleife dann, wenn folgende Vor- und Nachbedingung für den Scanneraufruf erfüllt werden:

  – Beim Scanner-Aufruf und beim Rücksprung aus dem Scanner ist das aktuelle Zeichen im Eingabestrom entweder

- das erste Zeichen des nächsten Token oder
- ein Whitespace-Zeichen vor dem nächsten Token oder
- das Eingabeende.

Um dies zu erfüllen, ist immer ein Token und das dahinter stehende Zeichen zu lesen.

Eine andere Möglichkeit besteht darin, ein zur Vorausschau gelesenes Zeichen, das nicht mehr zum Token gehört, wieder in den Eingabestrom zurückzuschieben (*unget*- oder *putback*-Operation).

Schlüsselwörter und Bezeichner

Der Scanner selbst wird sehr viel einfacher, wenn alle Schlüsselwörter von ihm vorläufig als „Bezeichner“ klassifiziert werden. In der Symboltabelle steht zu jedem (echten oder vermeintlichen) Bezeichner dann die korrekte Klassifikation.

Dazu initialisiert man vor dem Scannen einfach die Tabelle mit je einem Eintrag für jedes Schlüsselwort.

Findet der Scanner einen „Bezeichner“, wird in der Tabelle ein passender Eintrag gesucht. Falls keiner vorhanden ist, handelt es sich um einen echten Bezeichner, der mit genau dieser Klassifikation neu eingetragen wird.

Ist dagegen schon ein Eintrag vorhanden, so kann die korrekte Klassifikation der Tabelle entnommen werden, einerlei, ob es sich um ein bei der Initialisierung eingetragenes Schlüsselwort oder einen während der Analyse vorher schon einmal gefundenen echten Bezeichner handelt.

Wer sucht in der Symboltabelle nach der korrekten Klassifikation ?

Für die Aufgabe ist es egal, ob der Scanner selbst oder die aufrufende Parser-Simulation dies tut. Bei einem vollständigen System wird man dies besser dem Parser überlassen, der ja bei echten Bezeichnern ohnehin Symboltabellenzugriffe durchführen muss.

Falls in den Quelltexten viele Schlüsselwörter vorkommen, ergibt sich aus dieser Methode natürlich ein Effizienznachteil wegen der vielen unnötigen Symboltabellenzugriffe.

Grobskizze für „ad hoc“-Scanner:

```
int gettoken(){
  ...
  /* Leerzeichen, TABs usw. ueberlesen: */
  do
    c=lieszeichen();
  while ( IS_WHITE_SPACE (c) );

  switch (c) {
  /* Operatoren, die aus einem Zeichen bestehen,
     und am ersten Zeichen erkannt werden koennen */
  case '.':
  case ',':
     ...    token=c;
            c=lieszeichen();
            return token;
  /* sonstige Operatoren: */
  case ':': /* Test ob ":" oder ":=" */
            if((c=lieszeichen())=='=') {
              c=lieszeichen();
              return TK_WERTZUWEISUNG;
              }
            else
              return ':';
  ...
  /* Kommentare */
  case '{':  ...

  /* Zahlen */
  case 0:
   ...
  case 9:  ...

  /* Bezeichner (bzw. Schluesselwoerter) */
  default:
           if (BEZEICHNER_ZEICHEN(c)) {
              ...
              }
           else
              /* weitere Möglichkeiten ?, Fehler ? */
  }
```

Die einzelnen Fälle der switch-Anweisung können dabei direkt ausprogrammiert

werden. Eine Alternative ist ein kleiner endlicher Automat, der mehrere Zustände unterscheidet, für die jeweils ein separates Verhalten programmiert wird.

4.3.1 Minuszeichen: Operator oder Vorzeichen?

Ein in fast allen Programmiersprachen auftretendes Problem ist die Klassifikation des Minuszeichens. Einerseits tritt es als (ein- oder zweistelliger) Operator und damit als selbständiges Token auf, z.B. in

```
x = -(y+1); // einstelliger Operator
x = x - y;  // zweistelliger Operator
```

Andererseits kann es auch Bestandteil eines Literals sein, dann ist es kein eigenständiges Token:

```
x = x + -15.3;     // Bestandteil des Literals -15.3
```

Das Problem liegt darin, dass der Scanner diese Fälle nicht anhand einer Vorausschau unterscheiden kann:

```
x = y-3;
x = y * -3;
```

In beiden Fällen liefert die Vorausschau nach dem '-' dasselbe.

Der Scanner kann die Unterscheidung nur treffen, wenn er sich bestimmte Information über die „Vorgeschichte“, also die voranstehenden Tokens, merkt. Einfacher und eleganter wird der Scanner, wenn er negative Literale nicht als solche erkennt, sondern in jedem Fall einen '-'-Operator liefert.

Der Parser hat dann zwei Möglichkeiten:

- Er erkennt aus dem Kontext, dass ein einstelliger Minusoperator und eine nachfolgende Zahl zusammen eine negative Zahl bilden und fügt beides zusammen.
- Er unterscheidet nicht zwischen einem negativen Vorzeichen und einstelligem Operator. Bei der Codeerzeugung muss natürlich beachtet werden, dass kein Maschinencode erzeugt wird, der die (positive) Zahl mit -1 multipliziert.

  Wenn ohnehin alle zur Übersetzungszeit berechenbaren Ausdrücke vom Compiler ausgewertet werden, erfordert dies keine Sonderbehandlung.

4.4 Systematische Scanner-Implementierung

Grundlage für die Automatisierung der Scanner-Entwicklung ist das Konzept der Endlichen Akzeptoren, die zur Prüfung der Token-Syntax bzw. Erkennung von Tokens verwendet werden.

4.4.1 Endliche Automaten / Endliche Akzeptoren

Endliche Automaten (EA) sind besonders einfache formale Maschinenmodelle, die im Compilerbau zur Spezifikation der lexikalischen Analyse dienen. Daneben gibt es viele weitere Anwendungsmöglichkeiten in der Informatik.

Das Verhalten eines EA wird durch Eingabe und Zustand bestimmt. Charakteristisch ist dabei die Endlichkeit der Zustandmenge, die einen EA von komplexeren Maschinenmodellen unterscheidet.

Definition

Ein (nichtdeterministischer) endlicher Automat (NEA) besteht aus

- einer endlichen Menge S von Zuständen
- einem Eingabealphabet Σ
- einer Übergangsrelation
  $next \subseteq (S \times (\Sigma \cup \{\varepsilon\})) \times S$
- einem Startzustand $s_0 \in S$
- einer Menge von Endzuständen $F \subseteq S$

Ist $(s, x, s') \in next$, so nennen wir s' Folgezustand zu s bei Eingabe x.

Definition

Ein deterministischer endlicher Automat (DEA) ist ein endlicher Automat $(S, \Sigma, next, s_0, F)$, bei dem *next* eine Funktion

$$next : (S \times \Sigma) \to S$$

ist.

Betrachten wir zunächst das einfachere DEA-Modell:

Der DEA befindet sich zunächst in seinem Startzustand s_0, liest in jedem Berechnungsschritt ein Eingabesymbol und geht über in den durch *next* definierten

Folgezustand. Wenn nach dem Lesen der gesamten Eingabe ein Endzustand erreicht ist, war die „Berechnung“ erfolgreich. Eine „nicht erfolgreiche“ Berechnung liegt vor, wenn

- zum aktuellen Zustand und zum nächsten Eingabesymbol kein Folgezustand definiert ist oder
- nach dem Lesen der gesamten Eingabe ein Zustand erreicht ist, der kein Endzustand ist.

Das NEA-Modell ist in zweierlei Hinsicht allgemeiner:

- Zu einem Zustand s und einem Eingabesymbol x kann es mehrere Folgezustände $s_1, s_2, \ldots, s_n$ geben:

  $(s, x, s_1) \in next$
  $(s, x, s_2) \in next$
  ...
  $(s, x, s_n) \in next$

  Der NEA entscheidet sich in nichtdeterministischer Weise für einen der Folgezustände.

- Zu einem Zustand s kann es sogenannte ε-Übergänge geben:

  $$(s, \epsilon, s_1) \in next$$

  Der NEA kann dann unabhängig vom nächsten Eingabesymbol in einen anderen Zustand wechseln.

4.4.2 Darstellung von endlichen Automaten

Die Zustandsübergänge eines EA lassen sich durch eine Zustandsübergangstabelle oder durch ein Zustandsübergangsdiagramm repräsentieren.

Während das Diagramm der menschlichen Auffassungsgabe besser Rechnung trägt, ist die Tabellen-Darstellung für die maschinelle Verarbeitung gut geeignet.

Beispiel:

Sei S={s1, s2, s3}die Zustandsmenge, s1 der Startzustand und s3 der einzige Endzustand, d.h. F={s3}

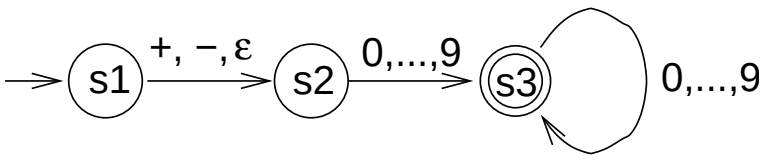
Repräsentation durch die Übergangstabelle

Zustand	Eingabe	Folgezustand
s1	+	s2
s1	-	s2
s1	ε	s2
s2	0	s3
s2	1	s3
⋮	⋮	⋮
s2	9	s3
s3	0	s3
s3	1	s3
⋮	⋮	⋮
s3	9	s3

Repräsentation durch Zustandsübergangsdiagramm

Das Diagramm ist ein Graph, in dem die Zustände durch Knoten und die Übergange durch Kanten repräsentiert werden, die mit den Eingabesymbolen bzw. mit ε beschriftet sind.

Falls der Automat von einem Zustand *s* aus mit unterschiedlichen Eingabesymbolen (bzw. ϵ) den selben Folgezustand s_1 erreicht, wird im Graph nur eine Kante von *s* nach s_1 dargestellt, die mit den entsprechenden Eingabesymbolen beschriftet ist.



Endzustände und Anfangszustand sind besonders markiert.

4.4.3 Automaten als Akzeptoren für formale Sprachen

Definition

Ein NEA $A = (S, \Sigma, next, s_0, F)$ akzeptiert eine Eingabe $w \in \Sigma^*$ genau dann, wenn ein Pfad $k_0, \ldots, k_n$ mit Kantenmarkierungen $x_1, \ldots, x_n$ im Übergangsdiagramm von A existiert, so dass

$k_0 = s_0$
$k_n \in F$ und
$x_1 \ldots x_n = w$

Die Menge aller von A akzeptierten Wörter ist die von A akzeptierte Sprache.

Für den oben angegebenen Automaten A ist die akzeptierte Sprache durch folgenden regulären Ausdruck gegeben: `[+|-]?[0-9]+`

Satz

Zu jedem regulären Ausdruck *r* gibt es effektiv einen endlichen Akzeptor *A*, so dass L(*r*) = L(*A*)

Es gibt diverse Algorithmen zur Konstruktion passender (nichtdeterministischer oder deterministischer) Akzeptoren, wir stellen in 4.4.4 den Thompson-Algorithmus vor.

Systematische Vorgehensweise zur Scannerkonstruktion

Sei T die Menge der lexikalischen Atome der Sprache (Tokens).

Sprachabhängig:

1. Beschreibe für alle Tokens $t \in T$ die Syntax durch je einen regulären Ausdruck r_t
2. Berechne zu jedem Token $t \in T$ ausgehend von r_t einen nichtdeterministischen endlichen Akzeptor A_t (Thompson-Algorithmus). Die Zustandsmengen aller Akzeptoren seien disjunkt.
3. Berechne den NEA, der sich durch Parallelschaltung aller nach Schritt 2 berechneten Akzeptoren A_t ergibt. Dabei wird die Zuordnung eines Tokens *t* zu jedem Endzustand von A_t festgehalten.
4. Berechne den äquivalenten DEA (Die Teilmengen, die Endzustände repräsentieren, müssen einelementig sein, sonst ist die Token-Syntax mehrdeutig)

5. Minimiere den nach 4 berechneten DEA, behalte aber die Unterscheidung solcher Endzustände bei, die an unterschiedliche Tokens gebundenen sind
6. Repräsentiere den minimalen DEA mittels komprimierter Zustands-Übergangs-Matrix

Sprachunabhängig:
Implementiere einen Tabellen-gesteuerten DEA-Simulator

Pseudocode-Schema für DEA-Simulation (ohne vernünftige Fehlerbehandlung):

```
zustand := startzustand ;
token := scanner();
while (nexttoken <> EOF) do
  zustand := folgezustand [ zustand, token ];
  if (zustand = fehlerzustand) then
    break;
  nexttoken := scanner();
done

if (zustand = fehlerzustand) or
   (zustand nicht Endzustand) then
  Fehlerbehandlung
else
  return tokenklasse[zustand];
fi
```

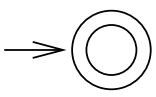
4.4.4 Vom regulären Ausdruck zum nichtdeterministischen Akzeptor – Thompson-Algorithmus

Eingabe: ein regulärer Ausdruck *r* über dem Alphabet Σ

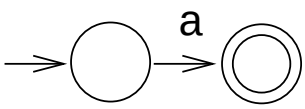
Ausgabe: NEA *A* für L(*r*)

Fallunterscheidung:

1. $r = \varepsilon$



2. $r = a \in \Sigma$



3. $r = r_1 \mid r_2$

   Parallelschaltung:

   (a) Konstruiere NEA A_1 zu r_1, A_2 zu r_2

   (b) Schalte A_1 und A_2 zu A wie folgt parallel:

   - alle Zustände und Übergänge aus A_1 und A_2 werden übernommen
   - ein neuer Startzustand s für A wird eingeführt
   - von s aus werden ε-Übergänge zu den beiden Zuständen hinzugefügt, die in A_1 und A_2 Startzustände sind.
   - die Endzustände von A_1 und A_2 bleiben Endzustände

4. $r = r_1 r_2$

   Serienschaltung:

   (a) Konstruiere NEA A_1 zu r_1, A_2 zu r_2

   (b) Schalte A_1 und A_2 zu A wie folgt hintereinander:

   - alle Zustände und Übergänge aus A_1 und A_2 werden übernommen
   - Startzustand von A ist der Startzustand von A_1.
   - Alle Endzustände von A_2 sind Endzustände von A
   - ein Endzustand von A_1 ist nicht länger Endzustand
   - zu jedem Endzustand von A_1 gibt es einen ε-Übergang in den Startzustand von A_2

5. $r = r_1^*$

   Rückführung:

   (a) Konstruiere NEA A_1 zu r_1

   (b) Konstruiere A aus A_1 wie folgt:

   - alle Zustände und Übergänge aus A_1 werden übernommen
   - ein neuer Startzustand s für A wird eingeführt, der auch gleichzeitig ein Endzustand ist.

- von s aus wird ein ε-Übergang in den Startzustand von A_1 eingefügt
- Alle Endzustände von A_1 sind Endzustände von A
- zu jedem Endzustand von A_1 gibt es einen ε-Übergang zu s

Anmerkung

- Zur Implementierung benötigt man natürlich einen Parser für reguläre Ausdrücke
- Die Implementierung wird einfacher, wenn man nur einen Endzustand zulässt (ggf. neuen Endzustand und ε-Übergänge von den ursprünglichen Endzuständen in den neuen einführen)
- Man kann aus dem NEA durch Teilmengen-Konstruktion einen DEA ableiten
- Es gibt Verfahren, die direkt einen DEA konstruieren

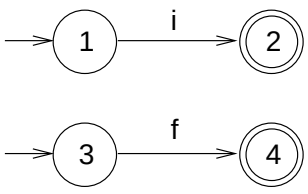
Beispiele

Um zu demonstrieren, wie der Automat, der alle Tokens erkennt, konstruiert wird, betrachten wir der besseren Übersicht wegen drei Tokenkategorien:

- Bezeichner, Syntax: `r1 = [A-Za-z][A-Za-z0-9]*`
- Das Schlüsselwort **if**, Syntax: `r2 = if`
- Ganzzahlliterale, Syntax: `r3 = -?[0-9]+`

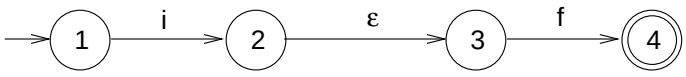
Beispiel für Serienschaltung

Als erstes wird die Serienschaltung am Beispiel des Schlüsselworts **if** gezeigt. Der reguläre Ausdruck `r2 = if` ist, wie man sofort sieht, durch Konkatenation der beiden atomaren regulären Ausdrücke `i` und `f` gebildet. Die beiden Automaten für diese Basisausdrücke ergeben sich aus Fall 2 der obigen Fallunterscheidung wie folgt:



Die Zustandsmengen der Ausgangsautomaten sind durch fortlaufende Nummerierung disjunkt gewählt, damit es bei der Zusammenführung keine Benennungskonflikte gibt.

Gemäß Fall 4 muss die Konkatenation der regulären Ausdrücke in eine Serienschaltung der Automaten umgesetzt werden:



Das Ergebnis:

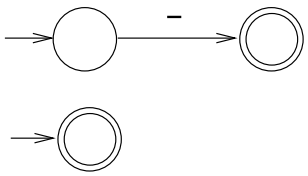
- Zustand 2 ist in der Serienschaltung kein Endzustand mehr.
- Zwischen die Zustandknoten 2 und 3 wird ein ϵ-Übergang neu eingefügt, so dass der neu entstandene NEA zunächst den ersten Ausgangsautomaten simuliert und danach den zweiten.
- Man sieht schon an diesem einfachen Beispiel, dass die systematisch konstruierten Automaten unnötig groß werden.

Beispiel für Parallelschaltung

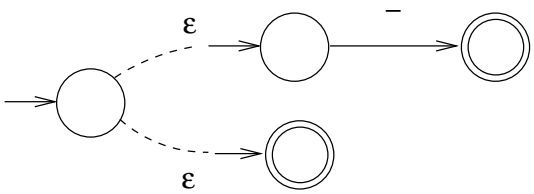
Betrachten wir als nächstes den Automaten für `-?[0-9]+`. Da in der Thompsonkonstruktion nur die Basisoperationen für reguläre Ausdrücke berücksichtigt werden, ist die UNIX-Notation zunächst auf diese Basisoperationen zurückzuführen. Wir erhalten:

$$(-|\epsilon)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

Betrachten wir als Beispiel für die Parallelschaltung die Konstruktion des Automaten für $(-|\epsilon)$ aus den folgenden beiden primitiven Automaten:



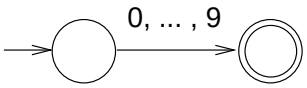
Das Ergebnis:



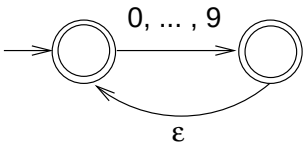
Der NEA kann von seinem neuen Startzustand aus durch die gestrichelt dargestellten neuen ϵ-Übergänge wahlweise beide Ausgangsautomaten simulieren und akzeptiert daher die Vereinigung der beiden Ausgangssprachen.

Beispiel für Schleifenschaltung

Betrachten wir als Beispiel für die Schleifenschaltung zur Implementierung des $*$-Operators die Konstruktion des NEA für $(0|1|2|3|4|5|6|7|8|9)^*$. Dabei gehen wir aber von folgendem minimierten NEA für $(0|1|2|3|4|5|6|7|8|9)$ aus:



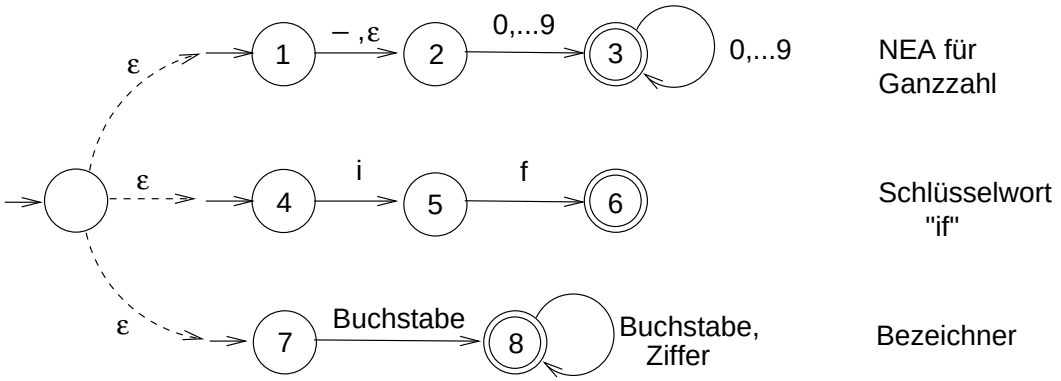
Das Ergebnis:



Der Anfangszustand ist jetzt zugleich Endzustand, damit ϵ akzeptiert wird. Die ϵ-Rückführung vom Endzustand in den Anfangszustand erlaubt beliebig häufige Simulation des Ausgangsautomaten, so dass beliebige Ziffernfolgen akzeptiert werden.

Beispiel für einen NEA, der unterschiedliche Tokens erkennt

Nun zur Parallelschaltung der Automaten für die einzelnen Tokenkategorien der Sprache. Auch hier gehen wir von optimierten Ausgangsautomaten für die einzelnen Tokens aus:



Wichtig ist dabei, dass die Zuordung der Endzustände der Ausgangsautomaten zu den jeweiligen Tokenkategorien gespeichert wird:

Zustand	**Tokenkategorie**
3	Ganzzahlliteral
6	**if**
8	Bezeichner

Dadurch kann der entstandene neue NEA, wenn er einen Endzustand erreicht, entscheiden, welches Token er erkannt hat.

4.4.5 Vom nichtdeterministischen zum deterministischen Akzeptor

Nichtdeterministisches Verhalten zu implementieren ist immer aufwändig. Um der Effizienz willen werden daher die nichtdeterministischen Akzeptoren in deterministische überführt. Diese lassen sich sehr einfach simulieren.

Satz

Zu jedem (nichtdeterministischen) endlichen Akzeptor A existiert ein äquivalenter deterministischer endlicher Akzeptor A' (d.h. A und A' akzeptieren die gleiche Sprache).

(ohne Beweis)

Konstruktion deterministischer endlicher Automaten (Teilmengenkonstruktion)

Eingabe: Ein NEA $A = (S, \Sigma, next, s_0, F)$

Ausgabe: Ein DEA $A' = (S', \Sigma, next', s_0, F')$ mit L(*A*) = L(A').

Idee: Der nichtdeterministische Ausgangsautomat A kann mit einer Eingabe w ggf. unterschiedliche Zustände aus S erreichen. Der Zustand, den der äquivalente DEA A' mit der Eingabe w erreicht, repräsentiert die Menge aller Zustände, die A mit der Eingabe w erreichen kann. Jeder Zustand T des DEA A' ist also eine Teilmenge von S.

Basis-Operationen:

zu jedem Zustand $s \in S$, zu jeder Zustandsmenge $T \in 2^S$ und jedem $a \in \Sigma$ sei definiert:

- ε-Abschluss(s)=$\{s' \in S \mid s' = s$ oder s' ist von s aus durch einen oder mehrere ε-Übergänge zu erreichen $\}$
- ε-Abschluss(T)=$\bigcup_{s \in T}$ ε-Abschluss(s)
- $move(T, a) = \{s' \in S \mid \exists_{s \in T} : (s, a, s') \in next\}$

Algorithmus:

Die Zustandsmenge *S* und die Übergangsfunktion *next* werden sukzessive erweitert, alle in S übernommenen Zustände seien anfangs unmarkiert.

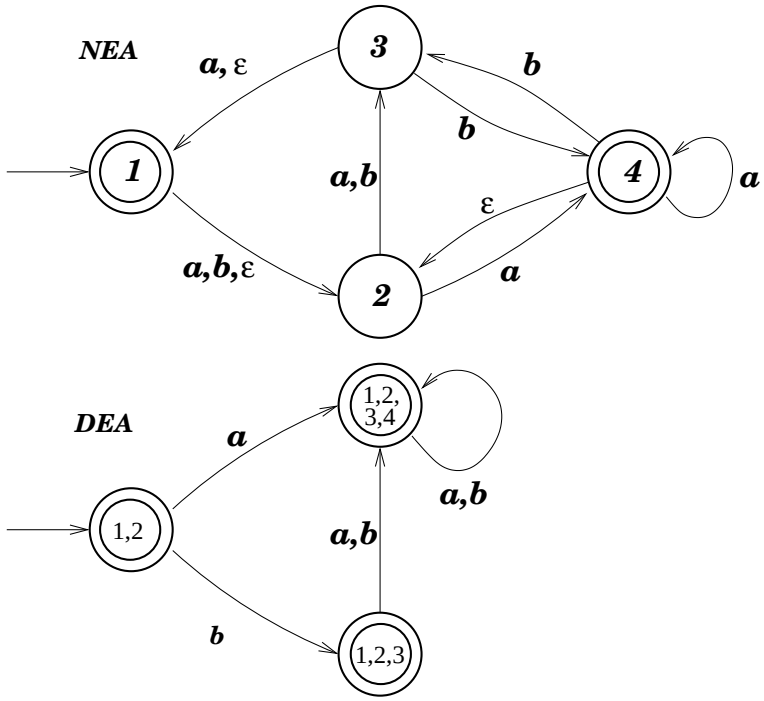
S := {ε-Abschluss(s_0) }
while (S enthält unmarkierten Zustand T) **do**
  markiere T
  for $a \in \Sigma$ **do**
    U:=ε-Abschluss(*move(T,a)*)
    if not $U \in S$ **then**
      $S := S \cup \{U\}$
    end if
    next(T,a) := U
  end for
end while

Der Startzustand des DEA ist der ε-Abschluss(s_0). Endzustand des DEA ist jeder Zustand *T*, der einen Endzustand *f* des NEA enthält. Dies bedeutet ja, dass der NEA (ggf. neben anderen Möglichkeiten) in den Endzustand *f* hätte gelangen können. Gemäß der NEA-Definition akzeptiert er in diesem Fall die Eingabe, somit muss auch der DEA sie akzeptieren.

Anmerkungen:

- Der NEA kann mit einer Eingabe w ggf. verschiedene Zustände erreichen. Mit dem Algorithmus wird berechnet, welche das sind. Die Menge dieser Zustände bildet einen Zustand im DEA.
- Der Startzustand des DEA ist die Menge der Zustände, die der NEA mit ϵ als Eingabe erreichen kann.
- Zu einem DEA-Zustand z wird der Folgezustand z' für ein Eingabesymbol x in zwei Schritten berechnet:
  - zuerst bestimmt man alle Zustände, die der NEA für die Eingabe x hätte erreichen können: $z1 = move(z, x)$
  - Dann nimmt man noch alle Zustände hinzu, die der NEA durch nachfolgende ϵ-Übergänge erreichen könnte: $z' = \epsilon - Abschluss(z1)$
- Der ϵ-Abschluss einer Zustandmenge z beinhaltet neben den Zuständen aus z auch alle die Zustände des NEA, die von einem Zustand aus z mit einem oder mehreren ϵ-Übergängen erreichbar sind.

Beispiel



Berechnung der Zustände:

Schritt	Bezeichnung	Berechnung	NEA-Zustände	Ergebnis
1	A	$\epsilon - Abschluss(1)$	$\{1,2\}$	DEA-Startzust.
2	A'	$move(A,a)$	$\{2,3,4\}$	Zwischenschritt
3	B	$\epsilon - Abschluss(A')$	$\{1,2,3,4\}$	neuer DEA-Zust.
4	A''	$move(A,b)$	$\{2,3\}$	Zwischenschritt
5	C	$\epsilon - Abschluss(A'')$	$\{1,2,3\}$	neuer DEA-Zust.
6	$B' = B$	$move(B,a)$	$\{1,2,3,4\}$	Zwischenschritt
7	$B'' = B' = B$	$\epsilon - Abschluss(B')$	$\{1,2,3,4\}$	bekannter DEA-Zust.
8	$B''' = A'$	$move(B,b)$	$\{2,3,4\}$	Zwischenschritt
9	$B'''' = B$	$\epsilon - Abschluss(B''')$	$\{1,2,3,4\}$	bekannter DEA-Zust.
10	$C' = B$	$move(C,a)$	$\{1,2,3,4\}$	Zwischenschritt
11	$C'' = B$	$\epsilon - Abschluss(C')$	$\{1,2,3,4\}$	bekannter DEA-Zust.
10	$C''' = A'$	$move(C,b)$	$\{2,3,4\}$	Zwischenschritt
11	$C'''' = B$	$\epsilon - Abschluss(C''')$	$\{1,2,3,4\}$	bekannter DEA-Zust.

4.4.6 Minimierung endlicher Automaten

Satz

Zu jedem (nichtdeterministischen) endlichen Akzeptor existiert ein äquivalenter deterministischer endlicher Akzeptor mit minimaler Zustandsmenge. Bis auf die Zustandsbenennung ist der minimale Akzeptor eindeutig bestimmt.

(ohne Beweis)

Konstruktion minimaler endlicher Automaten

Eingabe: Ein DEA $A = (S, \Sigma, next, s_0, F)$
(o.B.d.A sei *next* auf $(S \times \Sigma)$ totale Funktion – gegebenenfalls „tote“ Zustände einführen)

Ausgabe: Der minimale DEA $A' = (\mathcal{S}', \Sigma, next', s_0, F')$ mit L(A') = L(A).

Idee: Ein Wort $w \in S^*$ unterscheidet zwei Zustände, falls A mit Eingabe w aus genau einem dieser Zustände einen Endzustand erreicht.

Zwei Zustände sind äquivalent, falls kein $w \in S^*$ diese unterscheidet. Der Algorithmus prüft im i-ten Schritt für alle Zustände und für alle Eingabewörter der Länge i, ob zwei bislang nicht unterscheidbare Zustände durch das i-te Zeichen der Eingabe unterschieden werden.

Der Zustand, den A mit der Eingabe w erreicht, repräsentiert die Äquivalenzklasse der nicht unterscheidbaren Zustände, die A mit der Eingabe w erreicht.

Algorithmus:

Die Zustandsmenge von A' wird mit $\mathcal{S}$ bezeichnet. Jeder Zustand $z \in \mathcal{S}$ repräsentiert eine Klasse äquivalenter Zustände aus der Zustandsmenge S des Ausgangsautomaten A.

Die Äquivalenzklasseneinteilung wird sukzessive durch Verfeinerung berechnet:

1. (Initialisierung der Äquivalenzklassen) $\mathcal{S} := \{F, S \setminus F\}$

   (Für w=ε sind alle Endzustände und alle Nicht-Endzustände unterscheidbar)

2. (Verfeinerung der Äquivalenzklassen)

   Berechne $\mathcal{S}_{NEU}$ aus $\mathcal{S}$ (siehe unten)

3. (Solange neue Klasseneinteilung feiner, wiederholte Verfeinerung)

   if $(\mathcal{S}_{NEU} \neq \mathcal{S})$ **then**
   $(\mathcal{S} := \mathcal{S}_{NEU})$

goto 2;
end if

4. (Zustandstransformation eines Repräsentanten übernehmen)

   Wähle aus jeder Äquivalenzklasse einen Repräsentanten. Die Zustandsübergänge sind durch die Übergange der Repräsentanten definiert:

   Sei $next(s1, x) = s2$, *s1* und *s2* Repräsentanten der Äquivalenzklassen von *s1* bzw. *s2*, so sei definiert:

   $next(s1, x) = s2$

5. Entferne tote und nicht erreichbare Zustände

Verfeinerung der Äquivalenzklassen:

für jeden Zustand $z \in \mathcal{S}$: (*z* repräsentiert eine Klasse von Zuständen aus S)

- Verteile die Zustände in der Klasse *z* so auf Teilklassen, dass zwei Zustände des Ausgangsautomaten $s_1, s_2 \in S$ genau dann in der gleichen Teilklasse z' sind, wenn für alle Eingabesymbole $x \in \Sigma$ gilt: $next(s_1, x)$ und $next(s_2, x)$ sind in der gleichen Klasse von $\mathcal{S}$.
- Ersetze *z* in $\mathcal{S}$ durch die so gebildeten Teilklassen.

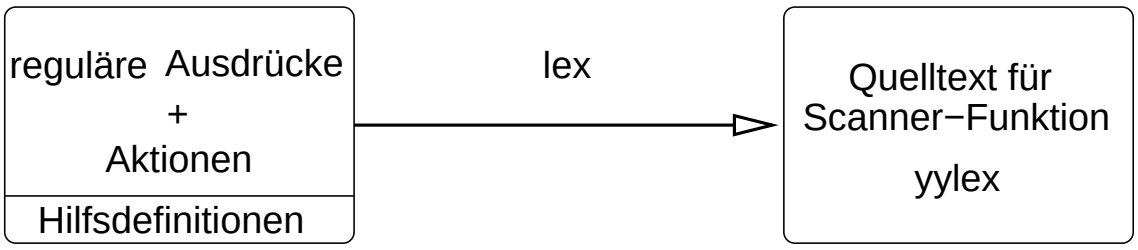
4.5 Scanner-Generatoren

Scanner-Generatoren erzeugen aus einer formalen Beschreibung der lexikalischen Syntax (reguläre Ausdrücke) einen Scanner. Die dazu notwendigen Schritte sind in 4.4, S. 61 beschrieben.

Es lohnt sich nicht, im Verlauf eines Compilerprojekts eigens einen Scanner-Generator zu entwickeln, um dann den Scanner generieren zu können: Der Aufwand ist unverhältnismäßig groß. (Bei der Parser-Entwicklung ist ein entsprechendes Vorgehen durchaus überlegenswert !)

Es lohnt sich dagegen durchaus, über den Einsatz eines Scannergenerators nachzudenken, denn gerade *lex*, *flex* und Konsorten sind auf vielen Rechnerplattformen verfügbar. Diese Generatoren erzeugen den Scanner in Form eines C-Quelltexts, so dass man an dem generierten Scanner noch Modifikationen auf der C-Ebene vornehmen kann. Auch für C++ und Pascal sind Generator-Varianten verfügbar.

4.5.1 Verwendung von *flex*



Die Verwendung von *flex* ist einfach:

- In der Eingabedatei gibt man für jedes Token an
  - die Syntax in Form eines regulären Ausdrucks
  - eine *Aktion* (C-Anweisung), die bei Erkennung des Tokens vom Scanner ausgeführt werden soll
- Der Generator erzeugt aus der Eingabedatei den Quelltext des Scanners, der aus der Definition der Scanner-Funktion *yylex* und aller benötigten Datenstrukturen und Hilfsfunktionen besteht.
- Der generierte Scanner wird compiliert und das erzeugte Objektmodul zusammen mit dem Parser und den anderen Modulen zum lauffähigen Compiler gebunden.

Die Schnittstelle zum einem von *yacc/bison* generierten Parser betehen im wesentlichen aus folgenden Konventionen:

- Der Rückgabewert der *yylex*-Funktion identifiziert das erkannte Token in Integer-Codierung.
- Die globale Variable *yytext* enthält die Zeichen, aus denen das Token besteht.
- Die globale Variable *yyleng* gibt die Länge von *yytext* an.

In den Aktionen, die man den Tokens zuordnet, muss die gewünschte Parser-Schnittstelle explizit programmiert werden. Dadurch lässt sich ein *flex*-generierter

Scanner mit einem beliebigen Parser leicht kombinieren. Das Kopieren des Tokens nach *yytext* geschieht allerdings automatisch, auch *yyleng* wird immer entsprechend angepasst.

Eingabedatei-Format

Eine typische *flex*-Eingabedatei hat folgenden Aufbau:

```
%{

    hier stehen C-Deklarationen, die in den generierten Scanner
    ohne Modifikation kopiert werden, dazu gehören alle
    Deklarationen, die in den Aktionen benötigt werden.

%}

    hier können reguläre (Hilfs-)Ausdrücke definiert werden,
    entweder, weil sie mehrfach benötigt werden, oder zur
    Strukturierung besonders komplizierter Ausdrücke

    Format:       Name  Ausdruck
    Verwendung:   {Name}

%%

    hier steht eine beliebige Liste von regulären Ausdrücken
    und zugehörigen Aktionen. Eine Aktion ist eine beliebige
    C-Anweisung.

%%

    hier stehen Definitionen von Datenstrukturen und Funktionen,
    die in den Scanner kopiert werden.
```

Bei der Verwendung benannter regulärer Ausdrücke ist darauf zu achten, dass *flex* eine reine Textersetzung durchführt, ohne den einzusetzenden Text zusätzlich mit Klammern zu versehen. Folgendes Beispiel zeigt die böse Falle auf:

```
BUCHSTABE  [a-z]|[A-Z]
%%
{BUCHSTABE}+     puts("Wort gefunden");
```

„{BUCHSTABE}“ wird ersetzt durch „[a-z]|[A-Z]“.
„{BUCHSTABE}+“ wird ersetzt durch „[a-z]|[A-Z]+“, nicht aber, wie gewünscht durch „([a-z]|[A-Z])+“. Sicherheitshalber sollte man also komplexe Ausdrücke in Definitionen mit Klammern versehen !

Der generierte Scanner verhält sich wie folgt:

- Falls ein Anfangsstück der Eingabedatei eindeutig zu einem Ausdruck zugeordnet werden kann, wird die dem Ausdruck zugeordnete Aktion ausgeführt. Sofern diese die Scannerfunktion nicht beendet (*return* oder *exit* in der Aktion) wird mit der Resteingabe genauso verfahren.
- Falls ein Anfangsstück der Eingabe einem Ausdruck A entspricht, ein längeres Anfangsstück aber einem Ausdruck B, wird die Aktion zu B ausgeführt. Der Scanner nimmt also immer das längstmögliche Token.
- Falls dasselbe Anfangsstück der Eingabe zu zwei verschiedenen Ausdrücken passt, entscheidet die Reihenfolge: Die Aktion zum ersten Ausdruck wird ausgeführt.
- Falls ein Stück Eingabetext zu keinem der Ausdrücke passt, wird es per Default einfach auf die Standardausgabe kopiert. Will man das nicht, sollte man dem regulären Ausdruck „.“, der ja zu jedem Eingabezeichen passt eine andere Aktion (oder keine Aktion) zuordnen. Dieser Ausdruck muss als letzter angegeben werden.

Beispiel:

- Folgende Behandlung von Integer- und Float-Literalen sei programmiert:

```
-?[0-9]+                                {return INT_NUMBER;}
-?[0-9]+(\.[0-9]+)?([eE]-?[0-9]+)?  {return FLOAT_NUMBER;}
```

  Findet der Scanner eine Float-Konstante z.B. „-3.12“, wird korrekt die zweite Regel ausgeführt, da sich hierbei das längere Token (-3.12 statt -3) ergibt.

  Findet der Scanner eine Integer-Konstante, passen beide Ausdrücke. Beim zweiten ist nämlich sowohl der Nachkomma- als auch der Exponenten-Anteil optional !.

  Hier ist die Reihenfolge wesentlich: Die Integer-Aktion steht zuerst und wird korrekterweise ausgeführt.

- Für die Erkennung von Schlüsselwörtern und die Abgrenzung zu Bezeichnern ist auch die Reihenfolge maßgeblich:

```
"while"                 {return WHILE;}
"do"                    {return DO;}
     ...
[_a-zA-Z][_a-zA-Z0-9]*  {return IDENTIFIER;}
```

Obwohl hier die Bezeichner-Syntax auch auf jedes beliebige Schlüsselwort passt, werden die Schlüsselwörter korrekt erkannt, da sie zuerst spezifiziert werden !

Ein einfaches Beispiel ist ein Programm, das einen Text von der Standardeingabe kopiert und dabei alle Kleinbuchstaben in Großbuchstaben umwandelt. Der Generator erhält folgende Eingabedatei:

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%%
[a-z]    { putchar(toupper(*yytext)); }
%%
main(){
    yylex();
}
```

Ein weiteres Beispiel ist dem *flex*-Manual entnommen:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }
```

```
{DIGIT}+"."{DIGIT}*        {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

5 Syntaxanalyse

Die Syntaxanalyse prüft, ob Eingabetexte (z.B. Programm-Quelltexte) bezüglich einer zugrundeliegenden Grammatik syntaktisch korrekt sind. Ein Eingabetext ist genau dann syntaktisch korrekt, wenn ein Syntaxbaum, bzw. eine Ableitung gemäß den Regeln der Grammatik konstruierbar ist.

Während der Syntaxanalyse werden i.d.R. weitere Verarbeitungsschritte vorgenommen, z.B. semantische Prüfung, Aufbau einer internen Zwischen-Repräsentation für das Programm (Symboltabelle + Abstrakter Syntaxbaum) oder Zwischencode-Generierung. Diese Aspekte werden erst in den nachfolgenden Kapiteln behandelt. In diesem Kapitel geht es zunächst einmal nur um die Berechnung der Ableitung. Dann wird auf die Fehlerbehandlung eingegangen.

5.1 Strategien zur Berechnung des Ableitungsbaums

Ziel der Syntaxanalyse ist die (explizite oder implizite) Konstruktion des Ableitungsbaums zu einer Eingabe auf der Grundlage einer kontextfreien Grammatik.

Bei den **Top-Down-Verfahren** wird der Baum von der Wurzel aus konstruiert, d.h. die Ableitung entsprechend der zugrundeliegenden Grammatik wird vom Startsymbol ausgehend berechnet.

Bei den **Bottom-Up-Verfahren** wird der Baum von den Blättern aus konstruiert, die Ableitung wird rückwärts berechnet.

5.2 Mehrdeutige Grammatiken, Präzedenz und Assoziativität

Bevor konkrete Verfahren zur Bestimmung des Ableitungsbaums behandelt werden, soll das Problem der Mehrdeutigkeit behandelt werden.

Definition

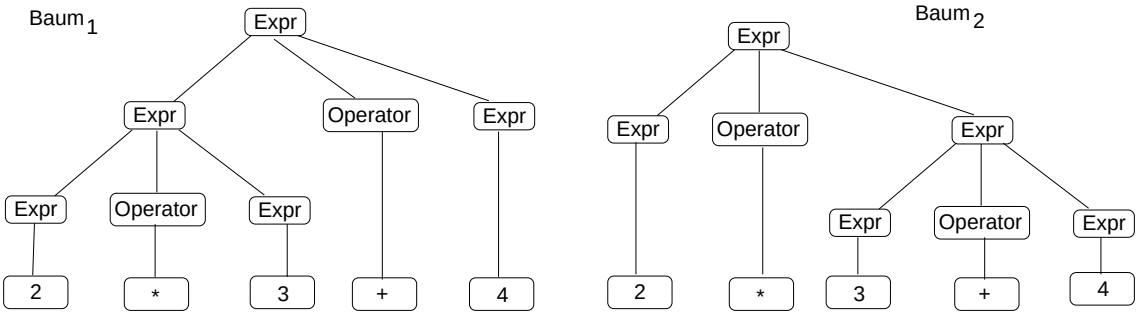
Eine Grammatik G heißt mehrdeutig, falls zu einem Wort w aus L(G) mehrere unterschiedliche Ableitungsbäume existieren.

Beispiel:

Expr →	Expr Operator Expr
	| '(' Expr ')'
	| KONSTANTE
	| BEZEICHNER
Operator →	'+' | '-' | '*' | '/'

Die Syntax von KONSTANTE und BEZEICHNER sei auf der lexikalischen Ebene definiert.

Mehrdeutigkeit zeigt sich etwa bei `2*3+4` :



5.2.1 Präzedenzen und Assoziativitätsregeln in Programmiersprachen

Bei ungeklammerten komplexen Ausdrücken ist die eindeutige Zuordnung der Operanden zu den Operatoren durch Präzedenz- und Assoziativitätsregeln definiert. (statt Präzedenz wird auch der Begriff Priorität gebraucht)

Präzedenz

Unter den Operatoren wird eine Präzedenzfolge festgelegt, ein Operator höherer Präzedenz bindet stärker.

Beispiel:

Der Pascal-Ausdruck

```
3+4*5
```

wird aufgrund der höheren Präzedenz von `*` gegenüber `+` als

```
3+(4*5)
```

interpretiert.

Assoziativität

Falls Gruppen von Operatoren gleicher Präzedenz existieren, wird innerhalb jeder Gruppe einheitlich linksassoziativ

$$a \bullet b \circ c = (a \bullet b) \circ c$$

bzw. rechtassoziativ

$$a \bullet b \circ c = a \bullet (b \circ c)$$

zugeordnet.

Wir bezeichnen einen Operator $\circ$ als *nicht-assoziativ*, wenn ein Ausdruck der Form $a \circ b \circ c$ nicht erlaubt ist.

Beispiel:

In Pascal wird der Ausdruck `a-b+c` wegen der Links-Assoziativität der Additions-Operatorengruppe als `(a-b)+c` interpretiert.

Pascal-Operatoren in absteigender Präzedenzfolge:

not, - (einstellig)	nicht-assoziativ
*, /, **div**, **mod**, **and**	linksassoziativ
+, −, **or**	linksassoziativ
=, <=, >=, <>, <, >	nicht-assoziativ

C-Operatoren in absteigender Präzedenzfolge:

Bezeichnung	**Operatoren**	**Assoziativität**
Primäroperatoren	`() [] . ->`	links
einstellige Operatoren	`++ -- - ! ~ & * sizeof`	rechts
Multiplikationsoperatoren	`* / %`	links
Additionsoperatoren	`+ -`	links
Shift-Operatoren	`<< >>`	links
relationale Operatoren	`< > <= >=`	links
Gleichheitstest	`== !=`	links
Bit-and	`&`	links
Bit-exor	`^`	links
Bit-or	`|`	links
and	`&&`	links
or	`||`	links
conditional	`?:`	rechts
Zuweisungsoperatoren	`= += -= *= /= %= <<= >>= &= |= ^=`	rechts
Komma-Operator	`,`	links

Man beachte die unterschiedliche Einordnung der logischen Operatoren. Während in C Vergleichs- oder Zuweisungsoperatoren hintereinander auftreten dürfen, ist dies in Pascal nicht erlaubt. Daher ist auch keine Assoziativitätsregelung für diese Pascal-Operatoren erforderlich.

5.2.2 Präzedenz und Assoziativität in der Grammatik

Die Mehrdeutigkeiten bei ungeklammerten Ausdrücken mit mehreren Operatoren lassen sich durch Berücksichtigung von Präzendenzen und Assoziativitäten in der Grammatik auflösen.

Man beachte, dass es bei Parser-Generatoren (z.B. bison) oft die komfortable Möglichkeit gibt, zu einer mehrdeutigen Grammatik die Präzedenzen und Assoziativitäten der Operatoren direkt anzugeben. Dies ist in aller Regel günstiger, da die Grammatik dadurch wesentlich kompakter gehalten werden kann.

Beispiele für einfache Ausdrucks-Grammatiken

1. mehrdeutig keine Assoziativität für + festgelegt

   Expr $\rightarrow$ Expr '+' Expr
   Expr $\rightarrow$ '(' Expr ')'
   Expr $\rightarrow$ KONSTANTE

2. eindeutig, + ist linksassoziativ

Expr → Expr '+' Term
Expr → Term
Term → '(' Expr ')'
Term → KONSTANTE

3. eindeutig, + ist rechtsassoziativ

   Expr → Term '+' Expr
   Expr → Term
   Term → '(' Expr ')'
   Term → KONSTANTE

4. eindeutig, + und $*$ linksassoziativ

   Expr → Expr '+' Term (*niedrigste Präzedenz)*
   Expr → Term
   Term → Term '*' Faktor (*mittlere Präzedenz)*
   Term → Faktor
   Faktor → '(' Expr ')' (*höchste Präzedenz)*
   Faktor → KONSTANTE

Schema zum Einbringen von Präzedenz und Assoziativität

Eine Ableitungsregel der Form $X \rightarrow Xw$ heißt *linksrekursiv*. Eine Ableitungsregel der Form $X \rightarrow wX$ heißt *rechtsrekursiv*.

Präzedenzgruppen und Assoziativitätsregeln für zweistellige Operatoren lassen sich in einer Grammatik wie folgt umsetzen:

- Jeder Präzedenzstufe i wird in der Grammatik ein eigenes Nonterminalsymbol $Expr_i$ zugeordnet.
- Ein linksassoziativer Operator Op_i der Stufe i wird durch die linksrekursive Regel
  $$Expr_i \rightarrow Expr_i Op_i Expr_{i+1}$$
  beschrieben. Dadurch ist sichergestellt, dass der rechte Operand weder den Operator selbst, noch Operatoren gleicher oder geringerer Präzedenz (z.B. Stufe i-1) in ungeklammerter Form enthält.
- Entsprechend wird ein rechtsassoziativer Operator durch die rechtsrekursive Regel
  $$Expr_i \rightarrow Expr_{i+1} Op_i Expr_i$$
  beschrieben.
- Für einen nichtassoziativen Operator *Op* der Stufe i lautet die Regel
  $$Expr_i \rightarrow Expr_{i+1} Op_i Expr_{i+1}$$

5.2.3 Grammatik-Transformationen für Top-Down-Analyse

Entfernung linksrekursiver Regeln

Transformation bei direkter Linksrekursion:

Ersetze linksrekursive Ableitung

$$X \to X\alpha \mid \beta$$

durch

$$\begin{aligned} X &\to \beta X' \\ X' &\to \varepsilon \mid \alpha X' \end{aligned}$$

Der Algorithmus zur Beseitigung indirekter Linksrekursionen steht im „Drachenbuch“.

Linksfaktorisierung

Falls zwei alternative rechte Regelseiten für ein Nonterminal *X* mit einem gemeinsamen Präfix *u* der Länge k beginnen, ist die eindeutige Regelauswahl mit einer Vorausschau der Länge k nicht möglich:

$$\begin{aligned} X &\to uv_1 \\ X &\to uv_2 \end{aligned}$$

Durch Linksfaktorisierung lässt sich das Problem lösen:

- Erzeuge eine Regel für den gemeinsamen Präfix der rechten Regelseiten
- Repräsentiere den Rest durch eine neue Variable
- Generiere die unterschiedlichen Suffix-Alternativen aus der neuen Variablen

$$\begin{aligned} X &\to uX' \\ X' &\to v_1 \\ X' &\to v_2 \end{aligned}$$

5.3 Top-Down-Verfahren

Bei Top-Down-Analyse wird für die Eingabe *w* eine Ableitung mit den Regeln der zugrunde liegenden Grammatik ausgehend vom Startsymbol berechnet. Der Ableitungsbaum wird dabei von der Wurzel ausgehend konstruiert.

5.3.1 Nichtdeterministische Spezifikation einer Top-Down-Analyse:

Eingabe: Grammatik $G = (N, T, S, P)$, Eingabewort $w \in T^*$

Ausgabe: Syntaxbaum zu *w*

Algorithmus:

Beginne die Konstruktion des Ableitungsbaums mit der Wurzel, die mit dem Startsymbol der Grammatik, S, markiert wird.

while (mit Nonterminalsymbol markierter Blattknoten vorhanden)
do

Bestimme einen mit einem Nonterminalsymbol *A* markierten Blattknoten zum neuen aktuellen Knoten *k*;

Wähle zu *A* eine Regel

$$A \to a_1 \dots a_n$$

Erzeuge für jedes Symbol der rechten Seite einen mit diesem Symbol markierten Baumknoten als Nachfolger von *k*

end while

if (Blattmarkierungen stimmen mit *w* überein) **then**

konstruierter Baum ist Syntaxbaum zu *w*;

die in der Schleife ausgewählten Regeln bilden eine Ableitung für *w*

end if

Probleme

1. **Welcher Knoten ist auszuwählen, falls mehrere Blätter mit Nonterminalen markiert sind ?**

   Für eindeutige Grammatiken ist die Auswahlreihenfolge zunächst einmal ohne Bedeutung, denn den Syntaxbaum ist eindeutig bestimmt.

   Bei einer Linksableitung wird jeweils das äußert links vorkommende Nonterminal gewählt. (Analog: Rechtsableitung)

2. **Bei der Auswahl ungeeigneter Regeln führt die Baumkonstruktion in eine Sackgasse.**

Lösungsmöglichkeit: „trial and error“
Revidiere bei Sackgassen jeweils die zuletzt getroffene Auswahlentscheidung und fahre mit nächsten passenden Regel fort. (Backtracking) Falls alle Alternativen in eine Sackgasse führen, ist das Wort nicht ableitbar.

Aber:

Analyse mit Backtracking ist i.a. zu ineffizient (n Tokens - $O(n^3)$ Ableitungsschritte — für ein Programm mit 5000 Quelltextzeilen a 10 Tokens ist $n^3 = 1,25 * 10^{14}$!)

Lösungsmöglichkeit:

Anstatt irgendeine Regel auszuwählen, versucht man anhand der Eingabe die nächste Regel eindeutig zu bestimmen (Vorausschau, „lookahead“)

Um den Abgleich mit der Eingabe durchzuführen, muss der Top-Down-Parser eine Linksableitung berechnen !

Falls eine Grammatik durch Vorausschau auf die nächsten k Tokens in jedem Fall eine eindeutige Regelauswahl ermöglicht, heißt sie LL(k)-Grammatik.

3. **Bei linksrekursiven Regeln sind Endlosschleifen möglich.**

   Lösungsmöglichkeit:

   Beseitigung der Linksrekursion durch Grammatik-Transformation.

5.3.2 Top-Down-Analyse durch rekursiven Abstieg

Voraussetzung: LL(1)-Grammatik $G = (T, N, P, S)$

Parser-Aufbau

- Zu jedem Nonterminalsymbol $A \in N$ wird ein Unterprogramm $PARSE_A$ konstruiert, das alle aus A ableitbaren Wörter analysiert
- $PARSE_A$ wählt anhand des lookahead-Symbols die wegen der LL(1)-Eigenschaft der Grammatik eindeutig bestimmte Ableitungsregel

$$A \to a_1 \dots a_n$$

- Die rechte Seite $a_1 \dots a_n$ wird von links nach rechts wie folgt verarbeitet:

$a_i \in T \Longrightarrow$ **if** (a_i=lookahead-Symbol) **then**
bestimme nächstes lookahead-Symbol
else Fehler
end if

$a_i \in N \Longrightarrow$ $PARSE_{a_i}$ aufrufen

Anmerkung

Falls die Grammatik keine eindeutige Regelauswahl erlaubt, kann das gleiche Schema verwendet werden. allerdings müssen bei mehreren Ableitungsalternativen mittels Backtracking alle Möglichkeiten durchprobiert werden. (Endlosrekursion vermeiden!)

Beispiel:

Wir konstruieren einen rudimentären Top-Down-Parser zu folgender Grammatik:

1. S $\rightarrow$ ABC
2. A $\rightarrow$ aa
3. A $\rightarrow$ b
4. B $\rightarrow$ bbB
5. B $\rightarrow$ ϵ
6. C $\rightarrow$ c

Zunächst definieren wir einen einfachen Scanner und einige Hilfsfunktionen:

```
#include <iostream>
#include <string>
using namespace std;
char token;         // naechstes Token im Eingabestrom

// Fehlermeldung ausgeben
void fehler(string *s){ cerr << s << endl; }

// Ausgabe einer Regel
void regel(string s){ cout << s << endl; }

// token = naechstes Symbol aus dem Eingabestrom
void scanner(){ cin >> token; }

void check(char erwartetes_token){
  // prueft, ob naechstes Token = Argument
  // wenn ja, naechstes Token lesen, sonst Fehlerabbruch
  if ( erwartetes_token != token ){
     cerr << "Fehler: statt " <<
             erwartetes_token << " wurde gelesen: " << token << endl;
     exit(1);
     }
  else
     scanner();
}
```

Jetzt der eigentliche Parser:

```
void S(); void A(); void B(); void C();

void parser() {
  scanner(); // Vorausschau-Token lesen
  S();       // Eingabe analysieren
  // Pruefen, ob unzulaessige weitere Zeichen folgen
  if (!cin.eof())
    fehler("Eingabeende erwartet");
}

void S() {
  regel("S -> ABC");
  A(); B(); C();
}

void A() {
  switch(token) {
  case 'a':
    regel("A -> aa"); check('a'); check('a');
    break;
  case 'b':
    regel("A -> b"); check('b');
    break;
  default:
    fehler("a oder b erwartet");
  }
}

void B() {
  if (token=='b') {
    regel("B -> bbB"); check('b'); check('b'); B();
    }
  else
    regel("B -> epsilon");
}

void C() {
  regel("C -> c"); check('c');
}

int main() { parser(); }
```

Kleine Schönheitskorrekturen sind noch angebracht. Im Parser sind redundante

Token-Vergleiche enthalten, etwa bei B:

```
void B() {
  if (token=='b') {
    regel("B -> bbB"); check('b'); check('b'); B();
    }
  else
    regel("B -> epsilon");
}
```

Statt des ersten `check('b');`-Aufrufs genügt hier natürlich ein scanner()-Aufruf.

Statt eines rekursiven Aufrufs, lässt sich B auch mit einer Schleife programmieren:

```
void B() {
  while (token=='b') {
    regel("B -> bbB"); scanner(); check('b');
    }
  regel("B -> epsilon");
}
```

5.3.3 Top-Down-Analyse mit tabellengesteuertem LL(1)-Parser

Betrachten wir einen Top-Down-Parser, der die Ableitungsregel

$$S \rightarrow ABC$$

ausgewählt hat, während der Analyse von A. Unabhängig davon, wie A analysiert wird, muss sich der Parser merken, dass anschließend erst B und dann C analysiert werden müssen.

Beim rekursiven Abstieg merkt sich der Parser dies dadurch, dass in der Analysefunktion für S die Aufrufe für die Analysefunktionen der drei Bestandteile nacheinander erfolgen:

```
void S() {
   switch(token){
   case ... :
          // REGEL: S->ABC
          A(); B(); C(); break;
   // sonstige Regeln fuer S
   case ... :
   }
}
```

Alternativ kann man die Bestandteile der rechten Regelseite, die noch zu verarbeiten sind, auf einem Stack ablegen. Dabei ist die richtige Reihenfolge zu beachten: Der Parser bearbeitet die rechte Regelseite von links nach rechts: ABC. Im Stack muss also C, weil es zuletzt analysiert wird, ganz unten stehen, B darüber und A ganz oben:

```
switch(token){
case ... :
       // REGEL: S->ABC
       push('C');
       push('B');
       push('A');
// sonstige Regeln fuer S
case ... :
}
```

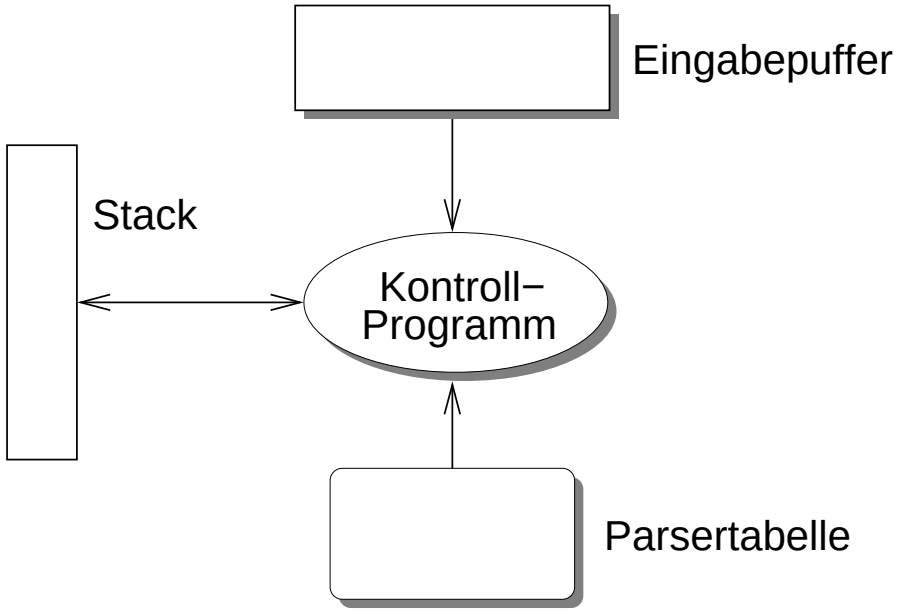
Der Parserstack beinhaltet also zu jedem Zeitpunkt der Analyse die Folge der noch zu bearbeitenden Eingabebestandteile.

Ein Bearbeitungsschritt betrifft immer den ganz oben auf dem Stack stehenden nächsten Bestandteil der Eingabe. Dies kann ein Terminal- oder ein Nonterminalsymbol sein:

- Bei einem Terminalzeichen *t* besteht die Bearbeitung nur darin, dass geprüft wird, ob das nächste Eingabesymbol mit *t* übereinstimmt. Wenn ja, ist *t* erfolgreich verarbeitet, so dass es vom Stack entfernt werden kann. Ein Scanner-Aufruf liefert das nachfolgende Token.
- Bei einem Nonterminalsymbol wird wieder die richtige Regel bestimmt und **anstelle** des Nonterminalsymbols die Symbole der rechten Regelseite auf den Stack geschoben.

Wenn man die Bestimmung der richtigen Ableitungsregel durch Nachschauen in einer Tabelle realisiert, hat man einen tabellengesteuerten Parser.

Modell eines tabellengesteuerten LL(k)-Parsers



Funktionsweise:

- Wir verwenden im folgenden $ als Endmarkierung der Eingabe
- Der Algorithmus gibt die Ableitung des Eingabeworts Regel für Regel aus

Der Algorithmus des Kontrollprogramms:

- Die Grammatik sei G=(T, N, P, S).
- In der Tabelle TAB steht zu dem gerade bearbeiteten Nonterminal $X \in N$ und dem aktuellen Eingabesymbol $a \in T$ die zu verfolgende Ableitungsregel aus P:

  $TAB[X, a] = X \to r$

  (oder, falls es diese nicht gibt, eine Fehlermarkierung *error*)
- X ist immer das oberste Stack-Element

```
push(S) /* Startsymbol S auf Stack */
a:=nexttoken /* Eingabesymbol := erstes Symbol */
repeat
      if Stack leer and a=$
      then stop /* erfolgreich beendet*/
      elseif X = a then /* ”Match”eines Terminalsymbols */
            pop
```

```
        a:=nexttoken
    elseif X ∈ N and TAB[X, a] = (X → Y1Y2 ... Yk)
    then
        print(X → Y1Y2 ... Yk) /* Ausgabe des nächsten Ableitungsschritts*/
        pop
        push Yk, ..., push Y2, push Y1 /*Y1 wird als nächstes bearbeitet */
    else
        Fehlerbehandlung
    end if
end repeat
```

5.3.4 Konstruktion der LL(1)-Parsertabelle

Zur Grammatik G=(T, N, P, S) wird die LL(1)-Parsertabelle bestimmt, in der zu jedem Nonterminalsymbol *X* und zu jedem Vorausschautoken *a* die passende Ableitungsregel

$$X \rightarrow w$$

steht. Falls keine solche Regel existiert, steht in der Tabelle ein Fehlereintrag.

Zum besseren Verständnis beachte man folgendes:

- Der Parser muss für ein bestimmtes Nonterminalsymbol *X* eine Ableitungsregel bestimmen. Als Auswahlkriterium dient das Vorausschautoken, also das nächste in der Eingabe gefundene Terminalsymbol.
- Wenn mit der Ableitungsregel $X \rightarrow w$ ein **nicht leeres** Wort *w* erzeugt wird, dann ist das Vorausschautoken das erste Symbol von *w*. Die Ableitungsregel passt also für alle Terminalsymbole mit denen *w* beginnen kann (FIRST(w)).

  U $\rightarrow$ cV bei Vorausschau c
  U $\rightarrow$ de bei Vorausschau d
  V $\rightarrow$ aY bei Vorausschau a
  V $\rightarrow$ bZ bei Vorausschau b
  V $\rightarrow$ U bei Vorausschau c oder d

  Auszug aus der Parsertabelle:

Nonterminal-Symbol	Vorausschau				
	a	b	c	d	e
U	Fehler	Fehler	U$\rightarrow$cV	U$\rightarrow$de	Fehler
V	V$\rightarrow$aY	V$\rightarrow$bZ	V$\rightarrow$U	V$\rightarrow$U	Fehler

- Ganz anders sieht es aus, wenn mit der Ableitungsregel $X \rightarrow w$ das leere Wort erzeugt wird, egal ob direkt ($w = \epsilon$) oder indirekt.

  Da *X* in diesem Fall leer ist, gehört das Vorausschausymbol nicht zu *X*, sondern zu einem auf *X* folgenden anderen Bestandteil der Eingabe. Ein *Folgesymbol* von *X* ist ein Terminalsymbol, dass in der Eingabe direkt hinter *X* steht.

  Die Ableitungsregel passt also in diesem Fall für alle möglichen Folgesymbole (FOLLOW(X)).

1. Bestimmung der Anfangssymbolmengen

Bestimme zu jedem Grammatik-Symbol $X \in (T \cup N)$ die Menge FIRST(X) (Menge aller Terminalzeichen, mit denen aus X abgeleitete Wörter beginnen können; enthält ε, falls ε aus X ableitbar)

1. Falls X $\in$ T, FIRST(X)=X
2. Falls $(X \to \varepsilon) \in P$, übernehme ε in FIRST(X)
3. Für jede Regel $(X \to Y_1 \dots Y_k) \in P$ verfahre wie folgt:

   Falls $\varepsilon \in FIRST(Y_1) \wedge \dots \wedge \varepsilon \in FIRST(Y_{i-1})$ für ein i, $1 \leq i \leq k$, übernehme alle Terminalzeichen aus FIRST(Y_i) in FIRST(*X*)

   Falls $\varepsilon \in FIRST(Y_1) \wedge \dots \wedge \varepsilon \in FIRST(Y_k)$ übernehme ε in FIRST(*X*)

(Man beachte, dass (2) und (3) iteriert werden müssen, bis sich an den FIRST-Mengen nichts mehr ändert, eine geschickte Implementierung berücksichtigt die Abhängigkeiten der Nonterminale zur Berechnung einer optimalen Reihenfolge $\to$ topologische Sortierung)

Hat man zu jedem Symbol die FIRST-Menge berechnet, so lässt sich daraus zu jeder beliebigen Satzform $X_1 \dots X_k$ die Menge der Terminale berechnen, mit denen aus der Satzform abgeleitete Wörter beginnen können (FIRST($X_1 \dots X_k$)).

Für die Bestimmung der FIRST-Menge eines Wortes *w* lassen sich demnach drei Fälle unterscheiden:

a) $w = \epsilon$

   Dann ist $FIRST(w) = \epsilon$

b) *w* beginnt mit einem Terminalsymbol *x*:

   $w = xv, x \in T$ Dann ist $FIRST(w) = FIRST(x) = \{x\}$

c) *w* beginnt mit einem Nonterminalsymbol *X*: $w = Xv, X \in N$

   Für die FIRST-Mengen-Bestimmung in diesem Fall ist es entscheidend, ob *X* auf ϵ abgeleitet werden kann:

   - Wenn nicht, beginnt *Xv* mit einem Symbol aus FIRST(X) und *v* ist nicht weiter von Interesse.
   - Falls ϵ aus *X* ableitbar ist, lässt sich aus *w* über *Xv* jedoch auch *v* ableiten, so das FIRST(v) auch bestimmt werden muss.

ϵ ist aus *X* genau dann ableitbar, wenn $\epsilon \in FIRST(X)$.

Es gilt also:

$$FIRST(Xv) = \begin{cases} FIRST(X) & \text{falls } \epsilon \notin FIRST(X) \\ FIRST(X)\backslash\{\epsilon\} \cup FIRST(v) & \text{falls } \epsilon \in FIRST(X) \end{cases}$$

2. Bestimmung der Folgesymbolmengen

Bestimme zu jedem Nonterminal $X \in T$ die Menge FOLLOW(X)

FOLLOW(X) ist die Menge aller Terminalzeichen, die in einer Satzform direkt hinter X auftreten können; enthält $, falls X in einer Satzform ganz rechts auftreten kann, also letzter Bestandteil der Eingabe ist.

Um FOLLOW(X) zu bestimmen, muss man jedes Auftreten von X auf einer *rechten* Regelseite betrachten und sich fragen, welche Terminalsymbole gemäß der Regel dahinter stehen können.

Aus der Betrachtung der Regel

$$X \rightarrow aYbZc$$

ergibt sich beispielsweise:

- $b \in FOLLOW(Y)$
- $c \in FOLLOW(Z)$

Algorithmus

1. Übernehme das Eingabeende-Token $ in FOLLOW(*S*)

   Das Startsymbol der Grammatik ist in einer Hinsicht ein Sonderfall bei der Folgesymbolbestimmung: Da *S* die gesamte Eingabe repräsentiert, folgt auf *S* das Eingabeende, repräsentiert durch ein besonderes Token $.

2. Für jedes Auftreten eines Nonterminalsymbols *X* in einer Regel $(A \rightarrow uXw) \in P$ übernehme alle Terminalzeichen aus FIRST(*w*) in FOLLOW(*X*).

   Im einfachsten Fall steht in der Regel hinter *X* direkt ein Terminalsymbol *z*. Dann ist *z* Folgesymbol von *X*.

   Wenn auf *X* dagegen ein Nonterminal *Y* folgt, wird FIRST(*Y*) benötigt. Alle Anfangssymbole von *Y* sind dann Folgesymbole von *X*.

Wichtig ist die Prüfung, ob $\epsilon \in$ FIRST(*Y*). Wenn das der Fall ist, kann *Y* gelöscht werden und das hinter *Y* stehende Symbol wird für die Folgemenge von *X* wichtig.

Beispiel: $Z \rightarrow uXYZv$

Zunächst sind natürlich alle Terminalzeichen aus FIRST(*Y*) Folgesymbole von *X*. Wenn $\epsilon \in$ FIRST(*Y*), muss man auch die Terminalsymbole aus FIRST(*Z*) hinzunehmen, denn aus *uXYZv* kann *uXZv* abgeleitet werden. Falls dann auch noch gilt: $\epsilon \in$ FIRST(*Z*), ist auch *v* Folgesymbol von *X*.

3. Für jede Regel $(A \rightarrow uB) \in P$ sowie jede Regel $(A \rightarrow uBv) \in P$ mit $\varepsilon \in FIRST(v)$, übernehme alle Terminalzeichen aus FOLLOW(*A*) in FOLLOW(*B*).

   Die Folgezeichen von *A* sind Folgezeichen von *B*, weil *B* in diesem Fall der letzte Bestandteil von *A* ist.

   Beispiel: Nehmen wir an, *x* sei Folgesymbol von *A* und es existiere die Regel $A \rightarrow uB$. Aus *Ax* kann ich mit dieser Regel *uBx* ableiten, *x* folgt also auf *B*.

(auch hier sind (2) und (3) zu iterieren)

3. Konstruktion der Parsertabelle

Die Parsertabelle TAB enthält für jedes Nonterminalsymbol A eine Zeile und für jedes Terminalsymbol x eine Spalte. Ein Eintrag TAB(A,x) enthält die Regel(n), die für A angewendet werden, falls x das Vorausschau-Token ist.

Konstruiere die Parsertabelle durch Ausführung der folgenden Schritte (1) und (2) für jede Ableitungsregel $A \rightarrow \alpha$:

1. Für jedes Terminalsymbol $a \in FIRST(a)$ übernehme die Regel in $TAB[A, a]$
2. Falls $\epsilon \in FIRST(\alpha)$, übernehme die Regel in $TAB[A, b]$ für jedes $b \in FOLLOW(A)$

   Falls $\epsilon \in FIRST(\alpha)$ und \$ $\in FOLLOW(A)$, übernehme die Regel in TAB[A, \$].

Alle leeren Tabelleneinträge werden mit *error* markiert.

Eine Grammatik hat die LL(1)-Eigenschaft, g.d.w. kein Tabelleneintrag $TAB[X, a]$ mehrere Regeln enthält.

Beispiel für die Bestimmung der FIRST-Mengen

Zur nachfolgenden Grammatik sollen FIRST(A), FIRST(B) und FIRST(C) bestimmt werden, A sei das Startsymbol.

1. A $\rightarrow$ a
2. A $\rightarrow$ BBC
3. B $\rightarrow$ b
4. B $\rightarrow$ ϵ
5. C $\rightarrow$ cc

Bestimmung der Reihenfolge (topologische Sortierung):

- An Regel 2 sieht man, dass zur Bestimmung von FIRST(A) zumindest FIRST(B) benötigt wird. Falls $\epsilon \in FIRST(B)$ wird auch noch FIRST(C) gebraucht.
- FIRST(B) und FIRST(C) sind von anderen FIRST-Mengen unabhängig, mit jeder der beiden Mengen kann man die Berechnung beginnen.
- Wir wählen die Reihenfolge FIRST(B), FIRST(C), FIRST(A).

Bestimmung der FIRST-Mengen:

$$\begin{aligned} FIRST(B) &= FIRST(b) \cup \epsilon = \{b, \epsilon\} \\ FIRST(C) &= FIRST(cc) = \{c\} \\ FIRST(A) &= FIRST(a) \cup FIRST(BBC) \\ &= \{a\} \cup FIRST(B) \backslash \epsilon \cup FIRST(C) \\ &= \{a\} \cup \{b\} \cup \{c\} \\ &= \{a, b, c\} \end{aligned}$$

Beispiel für die Bestimmung der FOLLOW-Mengen

Wir bestimmen die FOLLOW-Mengen der Nonterminalsymbole der obigen Grammatik.

Bestimmung der Reihenfolge (topologische Sortierung):

- An Regel 2 sieht man, dass FOLLOW(A) zur Bestimmung von FOLLOW(C) benötigt wird. Alle Folgezeichen von *A* sind auch Folgezeichen von *C*, weil *C* gemäß rechter Regelseite der letzte Bestandteil von *A* ist.

- Eine Abhängigkeit zwischen FOLLOW(B) und FOLLOW(A) ergibt sich dagegen aus Regel 2 nicht, weil C nicht auf ϵ abgeleitet werden kann. Damit ist B nicht letzter Bestandteil von A.

  Hätte die Grammatik aber etwa die Regel

  C$\rightarrow\epsilon$

  wären die Folgezeichen von A auch Folgezeichen von B, denn aus *Ax* ließe sich ableiten *BBCx* und schließlich *BBx*.

- Weitere Abhängigkeiten sind nicht vorhanden, da andere Nonterminalsymbole nicht als letzte Bestandteile rechter Regelseiten auftauchen.

Bestimmung der FOLLOW-Mengen:

FOLLOW(A)=$\{\$\}$, A ist Startsymbol und kommt in keiner Regel auf der rechten Seite vor.

FOLLOW(B) vereinigt alle Terminalzeichen aus FIRST(B) (erstes Auftreten in Regel 2) und alle Terminalzeichen aus FIRST(C) (zweites Auftreten in Regel 2). Also: FOLLOW(B)=$\{b, c\}$

FOLLOW(C) = FOLLOW(A), denn C kommt nur in Regel 2 auf der rechten Seite vor und zwar als letzter Bestandteil von A.

Beispiel für die Bestimmung der Tabelle

Wir betrachten wieder die Grammatik von oben. Die Tabelle sieht wie folgt aus:

	a	b	c	$
A	$A \rightarrow a$	$A \rightarrow BBC$	$A \rightarrow BBC$	*error*
B	*error*	$B \rightarrow b$ $B \rightarrow \epsilon$	$B \rightarrow \epsilon$	*error*
C	*error*	error	$C \rightarrow cc$	*error*

Man betrachte als Beispiel den Eintrag TAB(B,b), der zwei Regeln enthält. Die erste steht dort, weil b Anfangssymbol der rechten Regelseite ist. Die zweite, weil ϵ in der FIRST-Menge der rechten Regelseite steht und b Folgesymbol von B ist.

Die Grammatik hat demnach nicht die LL(1) Eigenschaft. Bei genauer Betrachtung erkennt man, dass die Grammatik mehrdeutig ist. Man sieht dies an der Eingabe *bcc*. Aus A wird BBC abgeleitet. Jetzt lässt sich das b sowohl aus dem ersten als auch aus dem zweiten B erzeugen, das verbleibende B wird gelöscht!

5.3.5 Beispiel für die Arbeitsweise eines tabellengesteuerten LL(1)-Parsers

Zur folgender LL(1)-Grammatik

1. S $\rightarrow$ ABC
2. A $\rightarrow$ aaA
3. A $\rightarrow$ C
4. B $\rightarrow$ bBd
5. B $\rightarrow \epsilon$
6. C $\rightarrow$ c
7. C $\rightarrow$ d

gehört die LL(1)-Parsertabelle

	a	**b**	**c**	**d**	**$**
S	S→ABC	*error*	S→ABC	S→ABC	*error*
A	A→aaA	*error*	A→C	A→C	*error*
B	*error*	B→bBd	B→ϵ	B→ϵ	*error*
C	*error*	*error*	C→c	C→d	*error*

Die Eingabe sei: *aadbdc*

Die Analyse wird Schritt für Schritt durch folgende Parserkonfigurationen beschrieben:

Schritt	**Stackinhalt** (rechts=oben)	**Resteingabe** (Vorausschausymbol fett)	**Regel** (laut Tabelle)
1	S	**a**adbdc	S→ABC
2	CBA	**a**adbdc	A→aaA
3	CBAaa	**a**adbdc	–
4	CBAa	**a**dbdc	–
5	CBA	**d**bdc	A→C
6	CBC	**d**bdc	C→d
7	CBd	**d**bdc	–
8	CB	**b**dc	B→bBd
9	CdBb	**b**dc	–
10	CdB	**d**c	B→ϵ
11	Cd	**d**c	–
12	C	**c**	C→c
13	c	**c**	–
14			ERFOLG!

5.3.6 LL(k) - Verfahren bei mehrdeutigen Grammatiken

Mehrdeutigkeit führt bei der Syntaxanalyse zu Situationen, in denen die Auswahl einer Ableitungsregel nicht möglich ist. Da die Mehrdeutigkeit natürlich auch nicht

durch Vorausschau aufgelöst werden kann, hat eine mehrdeutige Grammatik niemals die LL(k)-Eigenschaft für irgendein k.

Trotz allem werden mehrdeutige Grammatiken in Verbindung mit LL(k)-Verfahren zur Syntaxanalyse eingesetzt, denn sie sind oft wesentlich einfacher als entsprechende eindeutige Grammatiken.

Vorgehensweise bei mehrdeutigen Grammatiken:

- man bestimme die LL(k)-Parsertabelle wie gewohnt.
- die Tabelle enthält an mindestens einer Stelle mehrere Ableitungsregeln
- man mache die Tabelle durch Entfernen der „unerwünschten“ Regeln eindeutig

5.4 Bottom-Up-Syntaxanalyse (LR-Verfahren)

Funktionsprinzip: Rückwärts-Konstruktion einer **Rechtsableitung**

Aktionen des Parsers:

shift	nächstes Token auf den Stack legen
reduce(i)	oben auf dem Stack wurde ein „handle“, d.h. die rechte Seite der nächsten Ableitungsregel (i) erkannt; ersetze das handle durch die linke Seite der Regel
accept	Eingabe gelesen, Startsymbol auf Stack $\Rightarrow$ akzeptiere
error	Syntaxfehler

Beispiel:

Expr	$\rightarrow$	Expr ’+’ Term
Expr	$\rightarrow$	Term
Term	$\rightarrow$	Term ’$*$’ Factor
Term	$\rightarrow$	Factor
Factor	$\rightarrow$	’(’ Expr ’)’
Factor	$\rightarrow$	KONSTANTE

Parser-Konfigurationen

Nr.	Stack (handles fett)	Aktion	Resteingabe vor Aktion
1		shift	3 * 5 + 6 * 7
2	**KONSTANTE**	reduce(6)	* 5 + 6 * 7
3	**Factor**	reduce(4)	* 5 + 6 * 7
4	Term	shift	* 5 + 6 * 7
5	Term *	shift	5 + 6 * 7
6	Term * **KONSTANTE**	reduce(6)	+ 6 * 7
7	**Term * Factor**	reduce(3)	+ 6 * 7
8	**Term**	reduce(2)	+ 6 * 7
9	Expr	shift	+ 6 * 7
10	Expr +	shift	6 * 7
11	Expr + **KONSTANTE**	reduce(6)	* 7
12	Expr + **Factor**	reduce(4)	* 7
13	Expr + Term	shift	* 7
14	Expr + Term *	shift	7
15	Expr + Term * **KONSTANTE**	reduce(6)	
16	Expr + **Term * Factor**	reduce(3)	
17	**Expr + Term**	reduce(1)	
18	Expr	accept	

Grobspezifikation der Arbeitsweise der LR-Parser:

while not Eingabeende **do**
  while not nächstes *handle* oben auf Stack **do**
    shift
  end while
  reduce
end while

Zwei Konfliktsituationen sind möglich:

1. reduce/reduce-Konflikt

   Der aktuelle Stack lässt unterschiedliche Reduktionen zu
   Welche Regel ist zu wählen ?

2. shift/reduce-Konflikt

   Der aktuelle Stack lässt (mindestens) eine Reduktion zu. Soll reduziert werden oder sollen weitere Eingabesymbole gelesen werden (shift), um ein anderes „handle“ zu erhalten ?

Konfliktsituationen im Beispiel:

Konfiguration 4: shift/reduce-Konflikt

Statt shift wäre auch reduce(2) ($Expr \rightarrow Term$) möglich gewesen.

Durch eine Vorausschau auf das nächste Symbol kann der Parser erkennen, dass der Stack-Inhalt kein „handle“, sondern nur ein Teil eines „handle“ der Form *Term * Factor* sein muss.

Konfiguration 7: reduce/reduce-Konflikt

Statt reduce(3) wäre *Factor* auch ein „handle“-Kandidat für reduce(4) ($Term \rightarrow Factor$) gewesen (führt in Sackgasse !).

Aufgrund der „Vorgeschichte“, genaugenommen aufgrund der Tatsache, dass $Term * Factor$ oben auf dem Stack steht, kann die richtige Reduktion bestimmt werden.

Die Lösung der Konfliktsituationen ist sowohl vom Stack-Inhalt als auch von der Vorausschau abhängig.

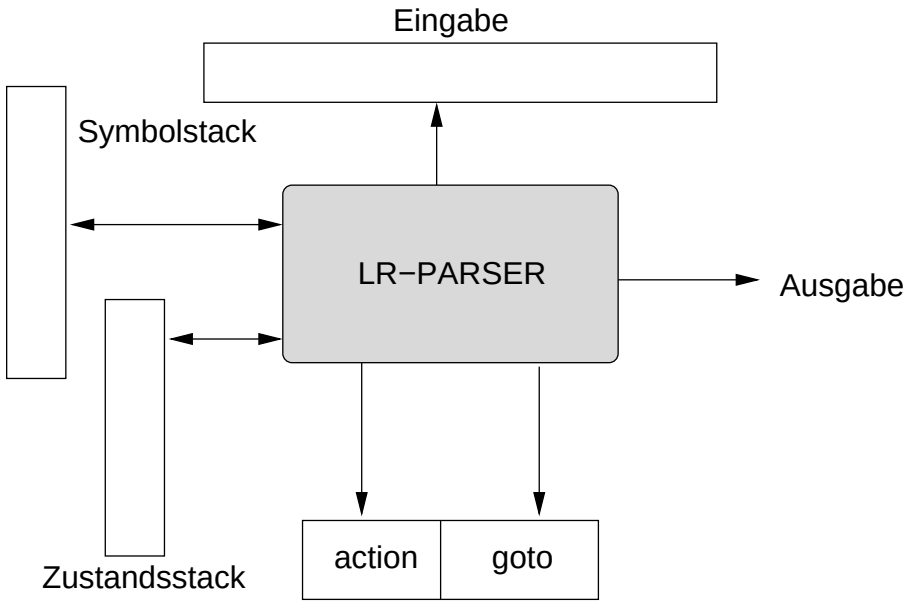
Ergebnis der LR-Parser-Theorie: Die Menge der möglichen Stack-Inhalte bildet eine reguläre Sprache

Konsequenz: Statt zur Konfliktlösung den Stack zu analysieren (ineffizient !), kann die notwendige Information in Form eines DEA-Zustands berechnet werden. Der aktuelle Zustand wird auf dem Stack abgelegt; er enthält die zur Konfliktlösung notwendige Information über den darunterliegenden Stack-Inhalt.

Die Bestimmung von Zustandsmenge und Parser-Tabelle, in der jedem Zustand abhängig von der Vorausschau ein Folgezustand und eine Aktion zugeordnet ist, erfolgt für die einzelnen Verfahren (LR, SLR, LALR) unterschiedlich.

5.4.1 LR-Parser-Schema

Das nachfolgende Schema verdeutlicht die Arbeitsweise eines LR-Parsers. SLR- und LALR-Parser arbeiten auf gleiche Weise, nur die verwendeten Tabellen sind unterschiedlich.



Algorithmus für Bottom-Up-Syntaxanalyse

Der Parser ist anfangs im Ausgangszustand *s0*. Dieser Zustand wird auf den Zustandsstack geschoben, der Symbolstack ist leer.

Der nächste Schritt des Parsers bestimmt sich

- aus dem lookahead-Symbol *l*
- dem aktuellen Zustand *s*

durch den Aktionstabelleneintrag **action**(*s*, *l*)

Fall 1: action(*s*, *l*)=*shift s1*

Parser schiebt *l* in einer SHIFT-Aktion auf den Symbolstack, geht über in Zustand *s1* und schiebt *s1* auf den Zustandsstack.

Fall 2: action(*s*, *l*)=*reduce A→B*

Sei n die Anzahl der Symbole auf der rechten Regelseite B, d.h. die Länge des auf dem Symbolstack liegenden „Handles“.

Sowohl vom Zustandsstack als auch vom Symbolstack werden n Elemente entfernt. A wird auf den Symbolstack geschoben. Der neue auf den Zustandsstack zu schiebende Zustand ergibt sich aus dem jetzt oben liegenden Zustand *s1* aus der *Sprung*-Tabelle: **Sprung**(*s1*, A)

Pseudocode:

```
push(S0);          // Anfangszustand
accept := false;
while (!accept) {
   sei S Zustand oben auf dem Stack;
   sei a das nächste Token;
   switch ( action[s][a] ) {
   case shift(n):  push(a,n);  // a auf Symbolstack, n auf Zustandsstack
                   scanner();
                   break;
   case reduce(r): sei A->ß die Regel r;
                   sei k die Länge von ß;
                   pop (k Elemente);  // k Symbole, k Zustände
                   sei t der Zustand oben auf dem Stack;
                   push(A, Sprung[t][A]);
                   break;
   case accept:    accept := true;
                   break;
   case error:     error();
                   break;
   }
}
```

Beispiel für Bottom-Up-Analyse mit Tabelle

Grammatik:

0. S $\rightarrow$ E $
1. E $\rightarrow$ T + E
2. E $\rightarrow$ T
3. T $\rightarrow$ a

Anmerkung: Zur Ausgangsgrammatik (Regeln 1-3) wird zunächst immer ein neues Startsymbol (hier *S*) und eine neue Regel 0 hinzugefügt, in der das Eingabeende-Token $ explizit auf der rechten Seite erscheint.

Der Parser akzeptiert bei Reduktion der Regel 0 und Vorausschau $.

Tabelle

	Aktion			**Sprung**	
Zustand	a	+	$	E	T
1	shift,5			2	3
2			accept		
3		shift,4	reduce(2)		
4	shift,5			6	3
5		reduce(3)	reduce(3)		
6			reduce(1)		

Beispiel für Parserkonfigurationen während der Analyse

Die Eingabe sei: `a + a`

Schritt	Symbolstack	Zustandsstack	Resteingabe	Aktion/Sprung
1		1	a+a$	shift,5
2	a	1 5	+a$	red(3), 3
3	T	1 3	+a$	shift,4
4	T+	1 3 4	a$	shift,5
5	T+a	1 3 4 5	$	red(3), 3
6	T+T	1 3 4 3	$	red(2), 6
7	T+E	1 3 4 6	$	red(1), 2
8	E	1 2	$	accept

5.4.2 Konstruktion von LR(0)-Parsertabellen

LR(0) ist ein einfaches Verfahren mit Konfliktlösung ohne Vorausschau.

Definition

Eine Produktion $A \to w$ mit einem Punkt irgendwo auf der rechten Seite heißt **LR(0)-Element** (kurz: Element).

Beispiel:

Zur Produktion $A \to aBc$ gehören 4 Elemente:

$A \to .aBc$, $A \to a.Bc$, $A \to aB.c$, $A \to aBc.$

Diese Elemente sind die Grundlage für die Berechnung der Zustände des Parsers. Die Position des Punkts spezifiziert dabei den Fortschritt beim Aufbau der rechten Regelseite: Alle Symbole links des Punkts befinden sich schon auf dem Parserstack.

Definition

Eine Menge von Elementen heißt **Kollektion**. Kollektionen dienen als *Zustände* des DEA.

Hilfsfunktionen zur Tabellenkonstruktion

Zwei Hilfsfunktionen werden zur Tabellenkonstruktion benötigt:

- Hülle (I) ist eine Erweiterung einer Kollektion I um neue Elemente. Falls in einem Element aus I ein Nonterminal hinter dem Punkt steht, enthält die Hülle Ausgangselemente zu den Regeln für das Nonterminal.
- Sprung (I,X) spezifiziert zu einer Kollektion I und einem Nonterminal X eine andere Kollektion J.

  Sprung (I,X) ist der Folgezustand von I nach Reduktion von X.

Sei *I* eine Kollektion und *X* ein Grammatiksymbol.

Algorithmus

```
HÜLLE (I) {
   do
     for (jedes Element A -> u.Xv aus I)
        for (jede Produktion  X->w)
            I := I U { X -> .w };
   while ( I hat sich in dieser Iteration geändert );
   return I;
 }
```

Algorithmus

```
SPRUNG (I,X) {
   J := {};
   for (jedes Element A -> u.Xw in I)
     J := J U { A -> uX.w }

   return HÜLLE(J);
 }
```

Definition: Der *Kern* einer Kollektion sind diejenigen Elemente, bei denen der Punkt auf der rechten Regelseite nicht am Anfang steht.

Eine Kollektion ist durch ihren Kern vollständig spezifiziert. Daher kann man auf die Angabe der anderen Elemente eigentlich verzichten. Diese lassen sich jederzeit durch Hüllenkonstruktion aus dem Kern berechnen.

Anmerkung: Bei der von *yacc/bison* erzeugten Beschreibung für den generierten Parser werden auch nur die Kernelemente aufgeführt.

Tabellenkonstruktion

Ergänze die Grammatik um die neue Startproduktion S' $\rightarrow$ S$. Sei T die Menge der Zustände des DEA und E die Menge seiner Übergänge.

Algorithmus

```
T := { HÜLLE( { S' -> .S$ } ) };
E := {};
do {
   for (jeden Zustand I in T)
      for  (jedes Element  A -> u.Xv  in I) {
          J := SPRUNG(I,X);
          T := T U {J} ;
          E := E U { (I,X) -> J };
      }
while ( T oder E haben sich in dieser Iteration geändert );
```

Ausnahme: Für $ wird kein Sprung berechnet; jeder Zustand mit Element S'$\rightarrow$S.$ akzeptiert bei Vorausschau $.

Berechnung der Reduktionsaktionen

Algorithmus

```
R := {};
for (jeden Zustand I in T)
   for  (jedes Element  A -> w.  in I)
       R := R U { (I, A->w) };
```

Beispiel

Wir betrachten folgende Grammatik

1. E $\rightarrow$ E+T
2. E $\rightarrow$ T
3. T $\rightarrow$ T*F
4. T $\rightarrow$ F
5. F $\rightarrow$ (E)
6. F $\rightarrow$ **const**

Schritt 1: Die Grammatik wird mit einem neuen Startsymbol S und folgender Regel erweitert:

0. S $\rightarrow$ E

Schritt 2: Berechnung der LR(0)-Kollektionen, die als DEA-Zustände dienen:

I_0 (Startzustand):	S	$\rightarrow$	.E
	E	$\rightarrow$	.E+T
	E	$\rightarrow$	.T
	T	$\rightarrow$	.T*F
	T	$\rightarrow$	.F
	F	$\rightarrow$	.(E)
	F	$\rightarrow$	.**const**
I_1=Sprung(I_0,E):	S	$\rightarrow$	E.
	E	$\rightarrow$	E.+T
I_2=Sprung(I_0,T):	E	$\rightarrow$	T.
	T	$\rightarrow$	T.*F
I_3=Sprung(I_0,F):	T	$\rightarrow$	F.

I_4=Sprung(I_0,'('):	F	$\rightarrow$	(.E)
	E	$\rightarrow$	.T
	E	$\rightarrow$	.E+T
	T	$\rightarrow$	.F
	T	$\rightarrow$	.T*F
	F	$\rightarrow$	.(E)
	F	$\rightarrow$	.**const**
I_5=Sprung(I_0,**const**):	F	$\rightarrow$	**const** .
I_6=Sprung(I_1,+):	E	$\rightarrow$	E+.T
	T	$\rightarrow$	.F
	T	$\rightarrow$	.T*F
	F	$\rightarrow$	.(E)
	F	$\rightarrow$	.**const**
I_7=Sprung(I_2,*):	T	$\rightarrow$	T*.F
	F	$\rightarrow$	.(E)
	F	$\rightarrow$	.**const**
I_8=Sprung(I_4,E):	F	$\rightarrow$	(E.)
	E	$\rightarrow$	E.+T
I_9=Sprung(I_6,T):	E	$\rightarrow$	E+T.
	T	$\rightarrow$	T.*F
I_{10}=Sprung(I_7,F):	T	$\rightarrow$	T*F.
I_{11}=Sprung(I_8,')'):	F	$\rightarrow$	(E).

Weitere Kollektionen gibt es nicht. Alle Übergänge des DEA sind in folgender Tabelle aufgeführt :

	E	T	F	+	*	(	)	**const**
I_0	I_1	I_2	I_3			I_4		I_5
I_1				I_6				
I_2					I_7			
I_3								
I_4	I_8	I_2	I_3			I_4		I_5
I_5								
I_6		I_9	I_3			I_4		I_5
I_7			I_{10}			I_4		I_5
I_8				I_6			I_{11}	
I_9					I_7			
I_{10}								
I_{11}								

Schritt 3: LR(0)-Tabellenkonstruktion

Aus der oben angegeben Zustandsübergabgstabelle ergeben sich direkt die *Shift*-Aktionen und die Sprung-Tabelle:

Unvollständige LR(0)-Parsertabelle ohne Reduktionen

	Aktion						**Sprung**		
	+	*	(	)	**const**	$	E	T	F
0			s4		s5		1	2	3
1	s6								
2		s7							
3									
4			s4		s5		8	2	3
5									
6			s4		s5			9	3
7			s4		s5				10
8	s6			s11					
9		s7							
10									
11									

Zu ergänzen sind noch die Reduktionen für alle Zustände mit reduzierbaren Elementen:

LR(0)-Parsertabelle

	Aktion						**Sprung**		
	+	*	(	)	**const**	$	E	T	F
0			s4		s5		1	2	3
1	s6					akz.			
2	r2	s7,r2	r2	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4			s4		s5		8	2	3
5	r6	r6	r6	r6	r6	r6			
6			s4		s5			9	3
7			s4		s5				10
8	s6			s11					
9	r1	s7,r1	r1	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

5.4.3 SLR(1)-Tabellenkonstruktion

Man beachte, dass in der LR(0)-Parsertabelle des obigen Beispiels zwei Shift/Reduce-Konflikte vorhanden sind, da in jedem Zustand mit Reduktionsmöglichkeit die Reduktion als Aktion übernommen wird.

Das Analyseverfahren SLR(1) (Simple LR(1)) berücksichtigt zur Auflösung solcher Konflikte eine Vorausschau um 1 Token. Der Unterschied zum LR(0)-Verfahren besteht ausschließlich in folgender Einschränkung bei der Tabellenkonstruktion:

Eine Reduktion bezüglich einer Regel $X \to w$ wird nur für die Folgesymbole von X in die Aktionstabelle eingetragen.

Im Beispiel fallen dadurch einige Reduktionen und damit sämtliche Konflikte weg:

SLR(1)-Parsertabelle

	Aktion						**Sprung**		
	+	*	(	)	**const**	$	E	T	F
0			s4		s5		1	2	3
1	s6					akz.			
2	r2	s7		r2		r2			
3	r4	r4		r4		r4			
4			s4		s5		8	2	3
5	r6	r6		r6		r6			
6			s4		s5			9	3
7			s4		s5				10
8	s6			s11					
9	r1	s7		r1		r1			
10	r3	r3		r3		r3			
11	r5	r5		r5		r5			

5.4.4 Kanonische LR(1)-Tabellenkonstruktion

Die mächtigste LR Tabellenkonstruktionsmethode ist die sogenannte kanonische LR-Methode, bei Beschränkung auf 1 Token Vorausschau kurz LR(1) genannt.

Die Grundidee ist folgende:

Bei SLR(1) wird eine Reduktion bezüglich einer Regel

$$X \to w$$

in jedem Zustand, der das Element $X \to w.$ enthält, für alle Folgesymbole von X in die Aktionstabelle eingetragen. Diese Eintragung ist unabhängig vom Kontext, in dem X verwendet wird.

Bei LR(1) wird dagegen in einem DEA-Zustand das 1. Symbol des Rechtskontexts von X berücksichtigt, also das Terminalsymbol, das hinter X steht.

Definition: Ein LR(1)-Element ist ein Paar

$$[X \to u.v, t]$$

bestehend aus einem LR(0)-Element $X \to u.v$ und einem Vorausschausymbol t, das unmittelbar hinter X erwartet wird. Das Vorausschausymbol spielt für $v = \epsilon$ eine Rolle, denn nur bei Vorausschau t wird für

$$[X \to u., t]$$

eine Reduktion durchgeführt,

Zur Verwendung betrachten wir zunächst ein Beispiel zu folgender Grammatik:

S $\rightarrow$ aXa
S $\rightarrow$ bXb
X $\rightarrow$ a
X $\rightarrow$ ϵ

Betrachten wir einen SLR(1)-Parser bei der Analyse der Eingabe bab nach Einlesen von b.

Das Kernelement des DEA-Zustands ist in dieser Situation

S $\rightarrow$ b.Xb

die Hülle besteht aus folgenden LR(0)-Elementen:

S $\rightarrow$ b.Xb
X $\rightarrow$ .a
X $\rightarrow$.

Bei Vorausschau a entsteht ein Shift/Reduce-Konflikt: Da a Folgesymbol von X ist, kann die ϵ-Regel reduziert werden.

Ein LR(1)-Parser dagegen merkt sich, dass in dem spezifischen Kontext hinter X ein b kommen muss, so dass bei Vorausschau a nur eine Shift-Operation in Frage kommt. Die LR(1)-Elemente des entsprechenden DEA-Zustands sind folgende:

$[S \rightarrow b.Xb, \$]$
$[X \rightarrow .a, b]$
$[X \rightarrow ., b]$

5.4.5 Konstruktion der kanonischen LR(1)-Parsertabelle

Im Prinzip wird wie bei der Konstruktion der SLR(1)-Parsertabelle vorgegangen, nur dass die auf ein Nonterminal folgenden Terminalsymbole in den Elementen mitgeführt werden müssen.

Das erste Kernelement ist demnach

$[S' \rightarrow .S, \$]$

Angepasst werden muss die Hüllenkonstruktion:

Zu einem LR(1)-Element

$[X \rightarrow u.Yv, t]$

wird für jede Regel

$Y \rightarrow w$

und für jedes Terminalsymbol $q \in FIRST(vt)$ in die Hülle aufgenommen:

$[Y \to .w, q]$

Man beachte, dass hierbei t aus dem LR(1)-Element für X an das LR(1)-Element für Y „vererbt“ wird, falls Y der letzte Bestandteil von X ist, also

$X \to u.Yv$ und $v = \epsilon$

Algorithmus

```
HÜLLE (I) {
   do
     for (jedes Element [ A -> u.Xv, t ] aus I)
        for (jede Produktion  X->w)
            for (jedes Terminal y aus FIRST(vt))
                I := I U { [ X -> .w, y ] };
   while ( I hat sich in dieser Iteration geändert );
   return I;
 }
```

Algorithmus

```
SPRUNG (I,X) {
   J := {};
   for (jedes Element [ A -> u.Xw, t ] in I)
     J := J U { [ A -> uX.w, t ] }

   return HÜLLE(J);
 }
```

Beispiel

Wir betrachten folgende erweiterte Grammatik

1. S' → S
2. S → X X
3. X → x X
4. X → y

Schritt 1: Berechnung der LR(1)-Kollektionen, die als DEA-Zustände dienen. Dabei werden zur Schreibvereinfachung die eckigen Klammern weggelassen und mehrere LR(1)-Elemente, die sich nur im Vorausschautoken unterscheiden zusammengefasst. So steht beispielsweise $A \to u.Xw, a|b|c$ für die drei Elemente

$$[A \to u.Xw, a], [A \to u.Xw, b], [A \to u.Xw, c]$$

I_0 (Startzustand): $S' \to .S,\ \$$
$S \to .XX,\ \$$
$X \to .xX,\ x \mid y$
$X \to .y,\ x \mid y$

I_1=Sprung(I_0,S): $S' \to S.,\ \$$

I_2=Sprung(I_0,X): $S \to X.X,\ \$$
$X \to .xX,\ \$$
$X \to .y,\ \$$

I_3=Sprung(I_0,x): $X \to x.X,\ x \mid y$
$X \to .xX,\ x \mid y$
$X \to .y,\ x \mid y$

I_4=Sprung(I_0,y): $X \to y.,\ x \mid y$

I_5=Sprung(I_2,X): $S \to XX.,\ \$$

I_6=Sprung(I_2,x): $X \to x.X,\ \$$
$X \to .xX,\ \$$
$X \to .y,\ \$$

I_7=Sprung(I_2,y): $X \to y.,\ \$$

I_8=Sprung(I_3,X): $X \to xX.,\ x \mid y$

Sprung(I_3,x) = I_3

Sprung(I_3,y) = I_4

I_9=Sprung(I_6,X): $X \to xX.,\ \$$

Sprung(I_6,x) = I_6

Sprung(I_6,y) = I_7

Aus diesen Zuständen wird die Parsertabelle ohne die Reduktionen wie beim LR(0) oder SLR(1)-Verfahren konstruiert. Die Reduktionen werden nur für die in den LR(1)-Elementen enthaltenen Vorausschautokens eingetragen, z.B. wird im Zustand I_8 die Reduktion der Regel $X \to xX$ nur für x und y eingetragen, im Zustand I_9 dagegen für das Eingabeende-Token $\$$.

5.4.6 LALR(1)-Verfahren

Ein SLR(1)-Parser hat für eine Programmiersprache mit einfacher Syntax mehrere hundert Zustände, ein LR(1)-Parser dagegen mehrere tausend. Beim LALR(1)-Verfahren werden „ähnliche“ Zustände des LR(1)-Automaten vereinigt, um die Anzahl der Zustände auf eine ähnliche Größenordnung wie bei SLR(1) zu reduzieren. Durch die gröbere Zustandseinteilung geht zwar Information verloren, dies führt aber nur in seltenen Fällen zu Reduce-Reduce-Konflikten, die im LR(1)-Parser nicht vorhanden sind. LALR(1) ist also ähnlich komfortabel zu benutzen wie LR(1), wobei „komfortabel“ bedeutet, dass für herkömmliche Programmiersprachen wenig Konflikte auftreten und daher mühsame Grammatik-Transformationen vermieden werden können.

Andererseits sind LALR(1)-Parser ähnlich kompakt, wie die weniger mächtigen SLR(1)-Parser. Daher werden in der Praxis häufig LALR(1)-Verfahren verwendet. Die Konstruktion ist allerdings komplex: Zunächst werden die LR(1)-Zustände berechnet. Danach werden Mengen von „ähnlichen“ LR(1)-Zuständen zu jeweils einem einzigen LALR(1)-Zuständ zusammengefasst, wobei zwei Zustände „ähnlich“ sind, wenn sie den gleichen LR(0)-Kern haben, also bis auf die Vorausschau-Tokens identisch sind.

5.5 Fehlerbehandlung bei der Syntaxanalyse

Unproblematisch ist, bei erkanntem Fehler eine aussagefähige Fehlermeldung ausgeben.

Problematisch dagegen ist die Frage nach der Fortsetzung der Syntaxanalyse eines fehlerhaften Programms.

Heuristische Vorgehensweise beim Wiederaufsetzen

Jeder Fehlersituation ist eine **Hypothese über die Art des Fehlers** zuzuordnen, z.B.

- Terminalsymbol wurde vergessen
- komplette Sub-Struktur wurde vergessen
- Schreibfehler bei Operator oder Bezeichner (Scanner liefert spezielles Fehler-Token)
- Schreibfehler in einem Schlüsselwort (Scanner liefert undefinierten Bezeichner)

Je nach Hypothese werden **fehlende Terminalsymbole ergänzt** oder auch **fehlerhafte Strukturen ignoriert**.

Als **Wiederaufsetzpunkte** für die Analyse einer fehlerhaften Struktur *S* bieten sich die Symbole in *FOLLOW(S)* an. Insbesondere sind Schlüsselwörter geeignet, da diese selten vergessen oder falsch geschrieben werden.

Daraus ergibt sich für einen „recursive descent"-Parser die Forderung, dass jede Parser-Prozedur die am Ort Ihres Aufrufs gültige FOLLOW-Menge kennt (diese ist ggf. als zusätzlicher Parameter zu übergeben).

Zielloses Überspringen von Text ist allerdings zu vermeiden, denn gerade das als Wiederaufsetzpunkt gesuchte Folgesymbol könnte ja auch fehlen! Hier können eindeutige **Anfangssymbole wichtiger Satzformen** als „Notbremse" dienen.

Man beachte insbesondere **Erfahrungswerte**, wie etwa das regelmäßige Vergessen von Interpunktationszeichen (z.B. Semikolon am Zeilenende). Hier kann der Parser intern von einer Grammatik ausgehen, bei der das Interpunktationszeichen optional ist. Natürlich erfolgt beim Fehlen eine entsprechende Meldung, ansonsten wird die Analyse „normal" fortgesetzt.

Solche Grammatik-Erweiterungen durch **Fehler-Produktionen** sind insbesondere auch bei automatischer Parser-Generierung sinnvoll.

5.6 Wertung der Analyseverfahren

Der Forschungsboom vor 25-30 Jahren erbrachte umfangreiche Parsing-Theorien und -Verfahren, z.B.

Parser-Klasse	**Grammatik-Klasse**
Top-Down	
Rekursiver Abstieg	ohne Linksrekursion
LL(k)	ohne Linksrekursion, Regelauswahl durch Vorausschau auf k Tokens möglich
Bottom-Up	
Operator-Präzedenz-Verfahren	Konfliktlösung durch Präzedenztabelle, besonders für Ausdrücke
SLR-Verfahren	einfachste LR-Methode
LALR-Verfahren	mächtiger, aber komplexer als SLR
kanonisches LR(k)-Verfahren	mächtiger, aber komplexer als LALR — Konfliktlösung durch Vorausschau auf k Tokens möglich
Early-Verfahren	beliebige kontextfreie Grammatik
Cocke-Kasami-Younger-Verfahren	beliebige kontextfreie Grammatik

- **Kompromiss bei Auswahl des Verfahrens erforderlich:** Je mehr spezielle Grammatik-Eigenschaften vorausgesetzt werden, desto einfacher und effizienter sind die verwendbaren Verfahren
- Top-Down-Verfahren sind (ohne Compilerbau-Werkzeuge) einfacher zu implementieren als Bottom-Up-Verfahren
- Bottom-Up-Verfahren sind insofern allgemeiner als Top-Down-Verfahren, als für viele reale Programmiersprachen LR(k)-Grammatiken existieren, LL(k)-Grammatiken dagegen nicht.

  (LL(k)-Grammatiken können allerdings – mit entsprechendem Transformations-Aufwand – für die meisten Sprachen erstellt werden).
- Jede LL(k)-Sprache ist eine LR(k)-Sprache. Die Umkehrung gilt nicht. (L ist LL(k)- (LR(k)-) Sprache, falls eine LL(k)- (bzw. LR(k)-) Grammatik für L existiert)

5.7 Der Parser-Generator *bison*

Bison basiert auf *yacc yet another compiler-compiler*. Der Parser-Generator ist auf unterschiedlichen Systemplattformen verfügbar und weit verbreitet. Er ist unter den GNU-Linzenzbedingungen als Quelltext frei verfügbar.

Bison erhält als Eingabe eine Grammatik, die die korrekte syntaktische Struktur der vom Parser zu analysierenden Eingabetexte beschreibt. Der Generator

erzeugt daraus einen tabellengesteuerten Bottom-Up-Parser in Form eines C- (oder C++ -)Quelltexts. Der wesentliche Bestandteil des generierten Parsers ist die Definition der Analysefunktion

```
int yyparse(void)
```

Diese Analysefunktion prüft zunächst nur die Eingaben auf syntaktische Korrektheit gemäß der Grammatik. Für praktische Anwendungen sind aber darüber hinaus gehende Verarbeitungsschritte, je nach der gewählten Phasenaufteilung des Compilers z.B. Typprüfungen, Erzeugung eines Syntaxbaums und einer Symboltabelle, oder Zwischencode-Erzeugung notwendig. Diese sollen schon während der Syntaxanalyse durchgeführt werden, müssen also in die Analysefunktion *yyparse* integriert werden. Dazu unterstützt der Generator die Möglichkeit, semantische Aktionen in Form von C-Anweisungen in die Regeln der Grammatik einzubetten.

Im nächsten Kapitel wird darauf näher eingegangen.

5.7.1 Das Eingabe-Dateiformat

Die Eingabedatei für den Generator hat (typischerweise) folgende Bestandteile:

```
%{
  C-Deklarationen
%}
  bison-Deklarationen
%%
  Grammatik + Aktionen
%%
  C-Definitionen
```

Beispiele für bison-Deklarationen:

```
/* Startsymbol */
%start Module

/* Operatoren, geringste Praezedenz zuerst */
%nonassoc '=' LEQ '<' GEQ '>' '#'
%left '+' '-' OR
%left DIV MOD '&' '*'
%right '~'
%right UMINUS
%right '.' '['

/* sonstige Tokens */
%token ASSIGN IDENT INTEGER MODULE TYPE CONST ...
```

Das Format für die Grammatik-Regeln entnehmen Sie bitte den folgenden Beispielen:

```
FieldList : IdentList ':' Type
          | /* EMPTY */
;

FieldDecls : FieldList ';' FieldDecls
           | FieldList
;

RecordType : RECORD FieldDecls END ;

Type : IDENT | ArrayType | RecordType ;

WhileStatement : WHILE Expression DO StatementSequence END ;

Statement : Assignment | ProcedureCall | IfStatement | WhileStatement ;
```

In die rechten Seiten der Regeln lassen sich, wie oben schon gesagt, *semantische Aktionen*, einstreuen, deren Stellung innerhalb der Regel den Zeitpunkt der Ausführung durch den generierten Parser bestimmt.

Beispiel:

```
X : X1 { Aktion_1 } X2 { Aktion_2 } X3 ;
```

Der generierte Parser wird die drei Bestandteile von X analysieren, nach der Analyse von X1 die erste und nach der Analyse von X2 die zweite Aktion ausführen.

Die C-Deklarationen und -Definitionen werden vom Generator in den generierten Parser-Quelltext kopiert. Benötigt werden in jedem Fall die Scanner-Fuktion *yylex* und eine Fehlermeldungsausgabe *yyerror*:

```
int yylex(void) /* Scanner */

int yyerror(char*mesg){
  /* Fehlerausgabe */
  fputs(mesg,stderr);
}

main(){ ... }
```

5.7.2 „Debuggen“ einer Grammatik

Rufen Sie den Generator *bison*) auf, um Ihre Grammatik auf **Konfliktfreiheit** zu testen:

```
bison -dtv parser.y
```

oder

```
bison -b parser -dtv parser.y
```

Aufrufoptionen:

-d	erzeugt die C-Definitionen für die Tokenklassen in einem separaten Header-File (bison: `parser.tab.h`,
-t	erzeugt Code zum debuggen des generierten Parsers
-v	erzeugt eine lesbare Spezifikation des generierten Parsers mit allen Konfliktmeldungen
-b *prefix*	weist *bison* an, als Präfix für die Ausgabedateien den String *prefix* zu verwenden. Ansonsten wird *y* genommen.

Ausgabedateien:

`parser.tab.h`	Scanner-Schnittstelle: Token-Definitionen u.a.
`parser.tab.c`	Parser-Quelltext: *yyparse* u.a.
`parser.output`	Parser-Beschreibungsdatei zum Debuggen der Grammatik

5.7.3 „Debuggen“ des generierten Parsers

Wird der Generator mit der Option `-t` aufgerufen, erzeugt er im generierten Programm Anweisungen zum Verfolgen der Analyse. Der Parser gibt seine Aktionen

dann auf *stderr* detailliert aus.

Dieser Code wird vom C-Compiler nur mitübersetzt, wenn `YYDEBUG` definiert ist. Im C-Deklarationsteil der bison-Eingabedatei sollte dann am Anfang stehen:

```
#define YYDEBUG 1
```

Der Code zum Verfolgen der Analyse kann dann zur Laufzeit des Parsers nach Belieben aktiviert oder deaktiviert werden:

```
extern int yydebug;

...

main(){
   yydebug=1; /* Debugging einschalten */
   yyparse();
   return 0;
}
```

Zur partiellen Verfolgung bietet es sich an, das Ein- und Ausschalten des Protokolls als semantische Aktion in die Regeln der Grammatik aufzunehmen.

Beispiel:

```
/* Verfolgen der Analyse der rechten Seite einer Wertzuweisung */
Assignment : IDENT Selector ASSIGN
             {yydebug=1;} Expression {yydebug=0;} ;
```

Eine bison-Datei zum Eingewöhnen

Machen Sie sich mit *bison* zunächst anhand der nachfolgenden Eingabedatei `grammar.y` vertraut.

Mit

```
make YACC=bison YFLAGS=-vdt grammar
```

sollten Generator, C-Compiler und Linker daraus einen lauffähigen Parser namens *grammar* erzeugen, den Sie über die Standardeingabedatei mit der Testeingabe `abbc` versorgen sollten.

```
/* grammar.y - bison Test-Eingabedatei */

%{
#include <stdio.h>
#include <ctype.h>
#define YYDEBUG 1
extern int yydebug;

%}

%%
S: A B C ;
A: 'a' | 'b' ;
B: 'b' B | /* empty */ ;
C: 'c' ;
%%
int yyerror(char*mesg){
  fputs(mesg,stderr);
}

int yylex(){
  int c;

  if (feof(stdin))
    return(-1);

  while(isspace(c=getchar()))
       ;

  return c;
}

main(){
  yydebug=1;
  yyparse();
  exit(0);
}
```

5.8 Fehlerbehandlung mit bison

Der Compilergenerator generiert aus der Eingabegrammatik einen Bottom-Up-Parser nach dem LALR(1)-Analyseverfahren. Ein Vorteil dieses Analyseverfah-

rens ist, dass es einen Fehler an der frühestmöglichen Stelle des zu analysierenden Wortes erkennt, was als Präfix-Eigenschaft bezeichnet wird.

Die Fehlerbehandlung wird mit Fehlerproduktionen realisiert. Der Benutzer fügt für wichtige komplexe Konstrukte *X* (z.B. Ausdrücke, Anweisungen, Funktionsdefinition) Produktionen der Form

$$X \rightarrow \alpha \; error \; \beta$$

in die Grammatik ein. Dabei ist *error* ein spezielles vordefiniertes Fehlertoken, das einen fehlerhaften Eingabeteil repräsentiert.

Die Fehlerproduktionen werden beim Generieren des Parsers wie normale Produktionen behandelt. Findet der generierte Parser in seiner Eingabe aber einen Fehler, dann behandelt er die Zustände, deren Elemente Fehlerproduktionen enthalten, in einer speziellen Weise.

Beispiel:

```
Anweisungen:  /* leer */
            | Anweisungen Anweisung ';'
            | Anweisungen error ';'
```

Die dritte Regel besagt, dass eine Anweisungsfolge gefolgt von einem fehlerhaften Eingabeteil gefolgt von einem Semikolon eine gültige Anweisungsfolge darstellt. Sie erlaubt dem Parser die Reduktion einer Anweisungsfolge (`Anweisungen`), auch wenn eine der Anweisungen in der Folge fehlerhaft ist.

Die normale Reduktion der Fehlerregel ist genaugenommen nur dann möglich, wenn oben auf dem Stack die 3 Symbole *Anweisungen*, *error* und ';' stehen. Wenn inmitten der Analyse einer Anweisung ein Fehler auftritt, wird auf dem Stack allerdings meist nicht nur die rechte Seite der Fehlerregel stehen, sondern zusätzliche Symbole, die bei Analyse der fehlerhaften Anweisung auf dem Stack abgelegt wurden. Außerdem wird das Semikolon meist nicht direkt hinter der Fehlerstelle kommen.

Um die Fehlerregel anwendbar zu machen, erfolgt die Sonderbehandlung: Der Parser nimmt bei Auftreten eines Fehlers zuerst soviele Symbole (bzw. auch Zustände) vom Stack bis eine Shift-Operation für das *error*-Token zu einem gültigen Präfix einer rechten Regelseite führt. Im Beispiel würde also zunächst alles oberhalb von *Anweisungen* vom Stack entfernt und erst dann das *error*-Token auf den Stack geschoben. Dadurch werden die schon analysierten Unterbestandteile der fehlerhaften Anweisungen ignoriert.

Anschließend ignoriert der Parser die nachfolgenden Tokens solange, bis er ein zu dem auf *error* in der Fehlerregel folgenden Symbol passendes Token findet, im Beispiel also bis zum nächsten Semikolon.

Beim Entdecken von Fehlern wird vom generierten Parser eine Standardfehlermeldung („Syntaxfehler“) ausgegeben, die dem Benutzer wenig Hilfestellung bei der Bestimmeung der Fehlerursache bietet.

Daher sollte in jede Fehlerregel noch eine semantische Aktion mit weiteren oder alternativen Meldungen bzw. zusätzlichen Fehlerbehandlungsoperationen eingefügt werden. Man denke zum Beispiel an eine fehlerhafte Deklaration. Falls der deklarierte Bezeichner erkannt wurde, kann man einen unvollständigen Symboltabelleneitrag vornehmen. Dann kann man beim Parsen der Verwendungsstellen zwischen undeklarierten und fehlerhaft deklarierten Bezeichnern unterscheiden und z.B. dort gezieltere Fehlermeldungen ausgeben.

Die Wahl der Fehlerregeln hängt von der verfolgten Fehlerbehandlungsstrategie ab. Eine einfache und oft nützliche Strategie ist das Überspringen einer fehlerhaften Deklaration oder Anweisung wie im Beispiel oben.

Auch das Parsen einer schließenden Klammer zu einer schon gefundenen öffnenden macht in vielen Fällen Sinn, z.B.

```
Faktor:   '(' Ausdruck ')'
        | '('  error   ')' | ... ;

...

DoWhileAnweisung :   DO Anweisungsliste WHILE Ausdruck ';'
                   | DO   error         WHILE Ausdruck ';'
;
```

Problematisch ist immer die Bestimmung der syntaktischen Ebene, auf der man die Fehlerbehandlung vornimmt. Werden die Fehlerbehandlungsregeln für Symbole weit oben im Syntaxbaum spezifiziert, ist die Fehlerbehandlung recht grobkörnig. Dadurch lässt sich einerseits die Vollständigkeit der Fehlerbehandlung leichter sicherstellen und der Fehlerbehandlungsaufwand ist gering. Andererseits werden dann im Fehlerfall auch große Teile der Eingabe ignoriert und die Fehlermeldungen sind sehr ungenau (z.B. „fehlerhafte Klassendeklaration“, „fehlerhafte Funktionsdefinition“).

yyerrok-Makro

Falls eine Annahme über die Fehlerursache nicht zutrifft, kann ein Syntaxfehler oft zu mehreren Fehlermeldungen führen. Falls beispielsweise inmitten einer ansonsten korrekten Anweisung versehentlich ein Semikolon eigefügt wurde, wird die im Beispiel oben verwendete Fehlerregel zur Meldung zweier fehlerhafter Anweisungen führen.

Um übermäßige Ausgabe von Fehlermeldungen zur selben Fehlerstelle zu vermeiden, werden nach einem Fehler frühestens nach drei erfolgreichen SHIFT-Operationen wieder weitere Fehlermeldungen erzeugt.

Soll schon früher ein neuer Fehler gemeldet werden, so muss dieses Standardverhalten durch das Makro *yyerrok* abgeschaltet werden. Der nächste Fehler wird dann auch ge- meldet, wenn noch nicht drei Eingabesymbole auf den Stack geschoben wurden. (Der Parser schaltet sofort vom Fehlerbehandlungsmodus in den normalen Modus zurück.)

Das *yyerrok*-Makro sollte keinesfalls direkt hinter einem error-Symbol als Aktion eingesetzt werden, da der Parser dann bei einem falschen Symbol den Fehlerbehandlungsmodus verlassen würde und als nächstes wieder das fehlerhafte aktuelle Eingabesymbol verarbeitet, also erneut in den Fehlerbehandlungsmodus eintritt. Der Parser gerät in diesem Fall in eine Endlosschleife, in der er ständig die gleiche Fehlermeldung ausgibt.

yyerrok sollte nur an den Stellen eingesetzt werden, an denen bei einem Fehler in jedem Fall wieder richtig aufgesetzt werden kann. Während sich der Parser im Fehlerbehandlungsmodus befindet und nach einem passenden Eingabesymbol zur Synchronisation sucht, werden nicht akzeptierbare Eingabesymbole verworfen. Will man dieses Verhalten explizit angeben, das heißt, das fehlerhafte Eingabesymbol aus der Eingabe entfernen, dann muss man das yyclearin-Makro in der Aktion aufrufen.

Dieses Makro wird überwiegend in interaktiven Parsern verwendet, um ein fehlerhaftes Vorausschausymbol zu entfernen und anschließend die Eingabe eines neuen Befehls vom Benutzer zu verlangen.

Beispiel:

```
stmtlist:     stmt l stmtlist stmt ;
stmt:         error   { reset_input(); yyclearin; }
```

Bei einem Fehler wird durch reset_input() die Eingabe zurückgesetzt und dann mit yyclearin das falsche Vorausschausymbol entfernt, um das Lesen eines neuen Befehls vorzubereiten.

Eine andere Variante der Anwendung von yyclearin: Wenn ein Programm in der nach error angegebenen Aktion eine „intelligente“ Fehlerbehandlungsoperation aufruft, die alle Token bis zu einem neuen Aufsetzpunkt liest und verarbeitet, dann handelt es sich beim nächsten von der lexikalischen Analyse gelieferten Token um das erste Token der nächsten Syntax-Konstruktion. In diesem Fall muss das fehlerhafte Token ignoriert und der Parser in den normalen Modus geschaltet werden. Eine Regel mit einer solchen Routine könnte folgendes Aussehen haben:

```
Isymbol:    error   { intelligent(); yyerrok; yyclearin; }
```

Eine Heuristik für die Fehlerbehandlung in Listen ist in folgender Aufstellung zusammengefasst:

1. Listen mit Y-Elementen (leere Liste erlaubt):

```
X:   /* leer */
     | X Y { yyerrok; }
     | X error ;
```

2. Listen mit Y-Elementen (leere Liste nicht erlaubt):

```
X:   Y
     | X Y { yyerrok; }
     | X error
     | error ;
```

3. Listen mit Y-Elementen und Trennzeichen T (leere Liste nicht erlaubt):

```
X:   Y
     | X T Y { yyerrok; }
     | X error
     | error
     | X error Y  { yyerrok; }
     | X T error ;
```

6 Semantische Analyse

6.1 Syntaxorientierte Spezifikation – Syntaxorientierte Übersetzung

Ein Übersetzungsprozess erfordert unterschiedliche Berechnungen z.B.

- Überprüfung von Typkompatibilitäten
- Erzeugung interner Programmrepräsentation
- Erzeugung Zielmaschinen-Code

syntaxorientierte Übersetzung heißt:

Algorithmen für Berechnungen werden eindeutig den Syntaxregeln der Sprache zugeordnet

Vorteile:

- flexibler Spezifikationsmechanismus
- automatische Generierung eines Übersetzers (Compiler-Compiler) möglich

Varianten syntaxorientierter Übersetzung:

- Übersetzungsschemata

  beliebige Algorithmen (semantische Aktionen) werden in die Syntaxregeln eingefügt
- Attributierte Grammatiken
  - jedem Grammatik-Symbol können nach Bedarf Attribute als Informationsträger zugeordnet werden
  - jeder Syntaxregel werden die Algorithmen für die Attribut-Berechnung zugeordnet

6.2 Abstrakter Syntaxbaum

Ein abstrakter Syntaxbaum dient zur internen Repräsentation des Programms während der Übersetzung. Der Baum wird während der Syntaxanalyse aufgebaut. Der Wurzelknoten repräsentiert das gesamte Programm, die Teilbäume jeweils syntaktische Substrukturen. Der Unterschied zwischen konkreter und abstrakter Syntax besteht darin, dass syntaktische Detailinformationen ("syntactic sugar"), die für die Erkennung aber nicht für die Weiterverarbeitung nötig sind, im abstrakten Baum weggelassen werden. Dazu gehören beispielsweise Klammern, Kommentare und Schlüsselwörter.

Die Verarbeitung eines Baumknotens hängt natürlich davon ab, welches Sprachkonstrukt dieser Knoten repräsentiert, beispielsweise erfordert eine Funktionsdeklaration eine völlig andere Behandlung als eine bedingte Anweisung. Dazu werden verschiedene Knotentypen unterschieden. Alle Knoten eines Typs werden einheitlich verarbeitet. Die einzelnen Knoten des Baums werden mit allen Attributen versehen, die für die Verarbeitung notwendig sind. Welche Informationen dies sind, hängt natürlich vom Knotentyp ab. Beispiel: Der Baum-Designer entscheidet sich, den Knotentyp „BinaryOperation“ für alle binären Operationen zu verwenden. Wesentliche syntaktische Informationen sind:

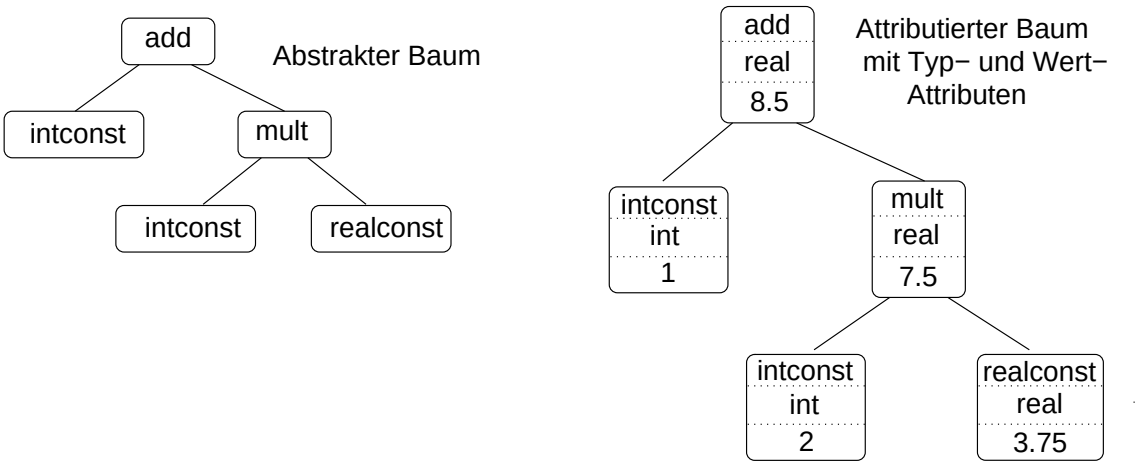
- Operator
- linker Operand
- rechter Operand

Diese Informationen müssen schon einmal für jeden „BinaryOperation“-Knoten vorhanden sein. Der Operator ist hier ein einfaches Attribut. Die Operanden sind selbst wieder Ausdrücke und müssen daher wieder durch Bäume repräsentiert werden. Die in der konkreten Syntax ggf. vorhandenen Klammern dienen nur der Syntaxanalyse und tauchen im abstrakten Baum nicht auf.

Man beachte, dass der Entwurf des abstrakten Baums viele Freiheitsgrade bietet. So könnte man anstelle des oben genannten Knotentyps „BinaryOperation“, für jede Operation einen eigenen Knotentyp („add“, „subtract“, . . .) verwenden.

Im Verlauf der Weiterverarbeitung werden ggf. weitere Attribute berechnet, der Baum also mit immer mehr Informationen angereichert. Beispiel: In statisch typisierten Sprachen muss zu jedem Ausdruck dessen Typ bestimmt werden, also auch für den Knotentyp „BinaryOperation“. Dies führt zu einer Erweiterung des Knotens um ein neues Attribut „Typ“. Man beachte dabei, dass ein Typ normalerweise hierarchisch aufgebaut ist, so dass Typen selbst wieder durch abstrakte Bäume repräsentiert werden. Ein Syntaxbaum mit zusätzlich berechneten Attributwerten heißt *attributierter* oder *dekorierter Baum*.

Beispiel: Pascal-Ausdruck `1 + 2 * 3.75`



6.3 Attributierte Grammatiken

Ein Attribut ist eine Information, die an einen Knotentyp im (abstrakten) Syntaxbaum geknüpft ist. Alle für die Übersetzung benötigten Informationen können als Attribute der Baumknoten definiert werden.

6.3.1 Vererbte und synthetische Attribute

Ein Attributwert, der einem Baumknoten vom Vaterknoten übergeben wird, heißt **vererbt** (top-down-Berechnung).

Ein Attributwert, der an den Vaterknoten übergeben wird, heißt **synthetisch** (bottom-up-Berechnung).

Eine **attributierte Grammatik** ist eine Grammatik $G = (T, N, S, P)$ mit folgenden Erweiterungen:

- jedem Symbol $X \in T \cup N$ ist eine endliche Menge von Attributen $\{X.A_1, \ldots, X.A_k\}$ zugeordnet.
- Zu jeder Ableitungsregel $X \to w$ wird angegeben, wie die synthetischen Attribute von *X* zu berechnen sind.
- Zu jeder Ableitungsregel $X \to w$ wird angegeben, wie die vererbten Attribute aller in *w* vorkommenden Nonterminalzeichen zu berechnen sind.

Notation: Durch die spezielle Schreibweise $X.\uparrow A_i$ wird ein Attribut A_i von *X* als synthetisches Attribut kenntlich gemacht, während $X.\downarrow A_i$ ein vererbtes Attribut ist.

Natürlich muss noch festgelegt werden, in welcher Form die Algorithmen zur Attributwertberechnung angegeben werden sollen. Wenn man Tools verwendet, die Attributierte Grammatiken verarbeiten, ist für jedes System eine spezielle Sprache dafür definiert. Für unsere Beispiele verwenden wir Pseudo-Code, der an C oder Pascal angelehnt ist oder die unten beschriebene *bison*-Notation.

Falls ein Nonterminalsymbol X in einer Regel mehrfach vorkommt, so verwenden wir die Indizes 1, 2, . . . von links nach rechts zur eindeutigen Zuordnung der Attribute, z.B.

$$Expr \rightarrow Expr \ + \ Expr \ \{Expr_1.\uparrow Wert := Expr_2.\uparrow Wert + Expr_3.\uparrow Wert\}$$

6.3.2 bison-Notation für Attribute

Der Parser-Generator *bison* unterstützt synthetische Attribute.

Jedem Grammatik-Symbol kann man nur **ein** Attribut zuweisen. Alle Attribute haben den gleichen Typ YYSTYPE.

Dies sieht auf den ersten Blick wie eine Einschränkung aus, denn man möchte schließlich zu jedem Grammatik-Symbol einen eigenen Satz von spezifischen Attributen verwenden.

Da jedoch YYSTYPE ein in C beliebig definierbarer Typ ist, kann man dem dadurch Rechnung tragen, dass man YYSTYPE als *union*-Typ deklariert und für jedes Grammatik-Symbol eine spezifische Variante definiert.

Auch die Einschränkung auf ein einziges Attribut pro Grammatik-Symbol ist keine echte Einschränkung, denn die zugehörige Typ-Variante kann ein strukturierter Typ mit beliebigen Komponenten sein.

Die Attribute der Terminalsymbole werden vom Scanner an den Parser in der globalen Variablen

```
YYSTYPE yylval
```

weitergegeben.

Die Attributwertberechnung für Nonterminalsymbole wird durch *semantische Aktionen* spezifiziert. Eine Aktion ist eine C-Anweisung, die in der rechten Seite einer Ableitungsregel steht. In der Aktion kann auf die Attribute mit $$ (Attribut der linken Regelseite), $1, $2 usw. (Attribute des 1., 2., . . . Symbols auf der rechten Regelseite) zugegriffen werden. (Bei der Zählung sind die Aktionen selbst mitzuzählen.)

Die Deklaration von YYSTYPE erfolgt mit einer speziellen `%union`-Anweisung, die Zuordnung einer Variante von YYSTYPE zu einem Symbol der Grammatik durch die speziellen Deklarationen

`%token` < Variante > *Token* ...
`%left` < Variante > *Operator* ...
`%right` < Variante > *Operator* ...
`%nonassoc` < Variante > *Operator* ...
`%type` < Variante > *Noterminal* ...

6.3.3 Beispiel für eine attributierte Grammatik: Roboter

Ein Roboter kann sich auf einen der Befehle N, O, S, W schrittweise in eine der vier Himmelsrichtungen bewegen.

Ausgehend von den Koordinaten $x = 0$ und $y = 0$ soll zu jeder Befehlsfolge die erreichte Endposition berechnet werden.

z.B. NNNOO x=2, y=3 oder SSW x=-1, y=-2

Grammatik:

Folge	$\rightarrow$	Folge Schritt
Folge	$\rightarrow$	ε
Schritt	$\rightarrow$	N
Schritt	$\rightarrow$	O
Schritt	$\rightarrow$	S
Schritt	$\rightarrow$	W

Attribute:

Folge.↑ex	x-Koordinate der Endposition
Folge.↑ey	y-Koordinate der Endposition
Schritt.↑deltax	relative Veränderung der x-Koordinate
Schritt.↑deltay	relative Veränderung der y-Koordinate

attributierte Grammatik:

Syntaxregeln	**Attributberechnungsregeln**
Folge → Folge Schritt	Folge$_1$.↑ex:=Folge$_2$.↑ex+Schritt.↑deltax; Folge$_1$.↑ey:=Folge$_2$.↑ey+Schritt.↑deltay;
Folge → ε	Folge.↑ex:=0; Folge.↑ey:=0;
Schritt → N	Schritt.↑deltax:= 0; Schritt.↑deltay:= 1;
Schritt → O	Schritt.↑deltax:= 1; Schritt.↑deltay:= 0;
Schritt → S	Schritt.↑deltax:= 0; Schritt.↑deltay:= -1;
Schritt → W	Schritt.↑deltax:= -1; Schritt.↑deltay:= 0;

6.4 Semantische Aktionen und Übersetzungschemata

Ein **Übersetzungsschema** ist eine Grammatik, in deren Regeln auf der rechten Seite zwischen den Grammatik-Symbolen auch **semantische Aktionen** eingefügt sind, die beliebige Berechnungen spezifizieren.

Die Reihenfolge der Berechnungen ist durch die Stellung der semantischen Aktionen in der Regel eindeutig bestimmt. Dadurch ist eine automatische Generierung eines Parsers, der die Berechnungen durchführt, ohne weiteres möglich.

6.4.1 Ausführung der Aktionen durch Top-Down-Parser

Ein Top-Down-Parser selektiert zunächst eine Ableitungsregel

$$A \to a_1 \dots a_k$$

und bearbeitet dann die Bestandteile der rechten Regelseite $a_1 \dots a_k$ von links nach rechts.

Wenn wir nun neben Terminal- und Nonterminalzeichen Aktionen als reguläre Bestandteile der rechten Regelseite zulassen, heißt *Bearbeitung* einfach: Ausführen der Aktion.

Beim rekursiven Abstieg ist nach der Selektion der Regel die rechte Seite $a_1 \ldots a_n$ von links nach rechts wie folgt zu verarbeiten:

$a_i \in T \Longrightarrow$ **if** (a_i=lookahead-Symbol) **then**
bestimme nächstes lookahead-Symbol
else Fehler
end if

$a_i \in N \Longrightarrow$ $PARSE_{a_i}$ aufrufen

$a_i\ Aktion \Longrightarrow$ a_i ausführen

Beim Tabellen-gesteuerten LL(1)-Parser sind die Aktionen nach der Regelauswahl zunächst wie Terminal- und Nonterminalzeichen auf dem Stack abzulegen. Die Ausführung der Aktion erfolgt, wenn die Aktion als oberstes Stack-Symbol bearbeitet werden muss.

6.4.2 Ein Übersetzer für Ausdrücke vom Infix- in das Postfixformat

Als Beispiel für die Verwendung semantischer Aktionen, soll mit *bison* ein Ausdrucksübersetzer erzeugt werden. Die zu übersetzenden Ausdrücke sind Integer-Konstanten oder mit zweistelligen Operatoren (+,-,*,/) und Klammern gebildete komplexe Ausdrücke.

Folgende Regeln spezifizieren den Übersetzer:

Postfix(`INTCONST`)	=	`INTCONST`
Postfix(`A+B`)	=	Postfix(`A`) Postfix(`B`) `'+'`
Postfix(`A-B`)	=	Postfix(`A`) Postfix(`B`) `'-'`
Postfix(`A*B`)	=	Postfix(`A`) Postfix(B) `'*'`
Postfix(`A/B`)	=	Postfix(`A`) Postfix(`B`) `'/'`
Postfix(`(A)`)	=	Postfix(A)

Beispiel:

```
   Postfix( (3-4) * 5 )
= Postfix( (3-4) ) Postfix(5) *
= Postfix(  3-4  )  5    *
= Postfix(3) Postfix(4) - 5 *
= 3 4 - 5 *
```

Die bison-Spezifikation für den Übersetzer zeigt Listing 6.4.2, S. 140. Man beachte, dass bei den Operatoren-Regeln nur jeweils die Ausgabe des Operators selbst, nicht aber der Argumente angegeben ist, z.B.

```
expr: expr '+' expr { printf("%s", " + "); }
```

Für die Übersetzung der Argumente wird nämlich während deren Analyse schon gesorgt. Die Ausgabe des Operators erfolgt erst nach der Analyse beider Argumente und damit auch nach der Postfix-Code-Generierung für diese Argumente gemäss obiger Übersetzungsregel.

6.4.3 Ausführung der Aktionen durch Bottom-Up-Parser

Beim Bottom-Up-Parser unterscheiden wir zwischen Regelende-Aktionen („end-rule action“) und Regelmitte-Aktionen („mid-rule action“). Eine Regelende-Aktion ist letzter Bestandteil der rechten Regelseite einer Grammatik-Regel, Regelmitte-Aktionen stehen an anderer Stelle.

Behandlung von Regelende-Aktionen

Eine Regelende-Aktion wird bei der Reduktion der rechten Regelseite ausgeführt.

Behandlung von Regelmitte-Aktionen

Wenn auf der rechten Regelseite eine Aktion *A* als Regelmitte-Aktion auftritt,

$$X \to u\{A\}v$$

will man damit ausdrücken, dass die Aktion nach dem Lesen von *u* und noch vor der Bearbeitung von *v* ausgeführt werden soll.

Allerdings berechnet der Bottom-Up-Parser erst unmittelbar vor einem Reduktionsschritt die korrekte Ableitungsregel, dann ist der *v* zugehörige Teil der Eingabe aber schon gelesen.

Eine Regelmitte-Aktion wird daher in eine Regelende-Aktion transformiert: Ein neues Nonterminalsymbol N_A repräsentiert die Aktion:

$$X \to uN_Av$$

für N_A wird eine neue Regel, nun aber mit Regelende-Aktion, eingeführt:

$$N_A \to \varepsilon\{A\}$$

6.5 Syntax-orientierte Spezifikationstechniken im Vergleich

Attributierte Grammatiken

- nichtdeterministisch, da Zeitpunkt der Attribut-Auswertung nicht angegeben wird

Listing 6.1 Infix-Postfix-Übersetzer

```
%{
#include <stdio.h>
%}

%left '+' '-'
%left '*' '/'

%%
lines: line | line lines ;
line: expr '\n' { putchar('\n'); } ;
expr: intconst        { putchar(' '); }
      | expr '+' expr { printf("%s", " + "); }
      | expr '-' expr { printf("%s", " - "); }
      | expr '*' expr { printf("%s", " * "); }
      | expr '/' expr { printf("%s", " / "); }
      | '(' expr ')'
;

intconst: digit | digit intconst;
digit:  '0' { putchar('0'); }
      | '1' { putchar('1'); }
      | '2' { putchar('2'); }
      | '3' { putchar('3'); }
      | '4' { putchar('4'); }
      | '5' { putchar('5'); }
      | '6' { putchar('6'); }
      | '7' { putchar('7'); }
      | '8' { putchar('8'); }
      | '9' { putchar('9'); }
;

%%
yyerror(char *message) {
 fprintf(stderr, "parser error: %s", message);
 }
yylex(){int c=getchar(); if(!feof(stdin)) return(c); else exit(0); }
main(){ yyparse(); }
```

- Auflösung des Nichtdeterminismus, d.h. Bestimmung eines Attribut-Auswertungs-Algorithmus durch:
  1. Analyse der Abhängigkeiten und manuelle Umsetzung in semantische Aktionen und/oder Programm
  2. automatische Abhängigkeitsananalyse, automatische Umsetzung in Programm zur Auswertung (keine zyklischen Abhängigkeiten erlaubt)
  3. automatische Umsetzung in Programm ohne Abhängigkeitsanalyse, Auswertungsschema fest durch Parser-Strategie bestimmt $\Longrightarrow$ Einschränkungen bei der Attributierung (häufig nur synthetische Attribute erlaubt, „S-Attributierung“, oder Auswertung bottom-up von links nach rechts, „L-Attributierung“)

Attributauswertung durch bison

- nach Methode 3
- Unterstützt eigentlich nur S-Attributierung. Durch freie Zugriffsmöglichkeit auf den Stack sind in sehr beschränktem Umfang auch L-attributierte Grammatiken verarbeitbar.
  - $$ ist der neue Top-of-Stack-Wert nach reduce-Aktion
  - $i ist der Wert des Stack-Eintrags zur i-ten *handle*-Komponente vor *reduce*-Aktion
  - $0, $-1, $-2, . . . sind die Attribute der darunterliegenden Stackelemente

Beispiel für die Verwendung von L-attributierten Grammatiken:

Folgende Regeln beschreiben die Funktionsaufruf-Syntax einer imaginären Programmiersprache *S*:

```
FunctionCall :  FunctionIdentifier '(' Parameters ')' ;
Parameters :    Expression
                |  Parameters ',' Expression
                ;
```

Greifen wir uns die Regel

```
Parameters :   Parameters ',' Expression ;
```

heraus und betrachten die Typprüfung: Der Typ des aktuellen Parameters (*Expression*), der sich aus `$3` ergibt, muss mit dem korrespondierenden formalen Parametertyp kompatibel sein. Wie erhält man Informationen zum formalen Parametertyp ?.

Falls *Parameters* ausschliesslich innerhalb von Funktionsaufrufen auftreten, kann man sich behelfen: Bei der Analyse von *Expression* ist der Parserstack nämlich immer wie folgt aufgebaut:

...	FunctionIdentifier	'('	Parameters	','	Expression
...	$-1	$0	$1	$2	$3

6.5.1 Implementierung des Roboter-Beispiels mit bison

Bei dem oben diskutierten Roboter-Beispiel ist die Umsetzung mittels bison einfach, da nur synthetische Attribute verwendet werden.

Die bison-Eingabedatei `roboter.y`:

```
%{
#include <stdio.h>
#include <unistd.h>

#define YYDEBUG 1
%}

%union {
  struct { int ex, ey; } endpunkt;
  struct { int dx, dy; } delta;
}

%type <endpunkt> kommandos;
%type <delta>    schritt;

%%
start:     kommandos { printf("X=%d, Y=%d\n", $1.ex, $1.ey); }
kommandos: kommandos schritt  { $$.ex=$1.ex+$2.dx;
                                $$.ey=$1.ey+$2.dy; }
         | /* empty */        { $$.ex=$$.ey=0; }
;

schritt:   'o'    { $$.dx= 1; $$.dy= 0; }
         | 's'    { $$.dx= 0; $$.dy=-1; }
         | 'w'    { $$.dx=-1; $$.dy= 0; }
         | 'n'    { $$.dx= 0; $$.dy= 1; }
```

```
;

%%
int yyerror(char *msg){fputs(msg,stderr);}
int yylex(){
  int c=getchar();
  if(c=='\n')
    return EOF;

  return c;
}

int main(){
  yydebug=1;
  yyparse();
}
```

Die von *bison* erzeugte Spezifikation des Parsers in `roboter.output` (*bison*-Format etwas anders):

```
Grammar
rule 1    start -> kommandos
rule 2    kommandos -> kommandos schritt
rule 3    kommandos ->          /* empty */
rule 4    schritt -> 'o'
rule 5    schritt -> 's'
rule 6    schritt -> 'w'
rule 7    schritt -> 'n'

Terminals, with rules where they appear

$ (-1)
'n' (110) 7
'o' (111) 4
's' (115) 5
'w' (119) 6
error (256)

Nonterminals, with rules where they appear

start (7)
    on left: 1
kommandos (8)
    on left: 2 3, on right: 1 2
```

```
schritt (9)
    on left: 4 5 6 7, on right: 2


state 0
    $default    reduce using rule 3 (kommandos)
    start       go to state 7
    kommandos   go to state 1
state 1
    start  ->  kommandos .   (rule 1)
    kommandos  ->  kommandos . schritt   (rule 2)

    'o'         shift, and go to state 2
    's'         shift, and go to state 3
    'w'         shift, and go to state 4
    'n'         shift, and go to state 5
    $default    reduce using rule 1 (start)

    schritt     go to state 6
state 2
    schritt  ->  'o' .   (rule 4)

    $default    reduce using rule 4 (schritt)
state 3
    schritt  ->  's' .   (rule 5)

    $default    reduce using rule 5 (schritt)
state 4
    schritt  ->  'w' .   (rule 6)

    $default    reduce using rule 6 (schritt)
state 5
    schritt  ->  'n' .   (rule 7)

    $default    reduce using rule 7 (schritt)
state 6
    kommandos  ->  kommandos schritt .   (rule 2)

    $default    reduce using rule 2 (kommandos)
state 7
    $           go to state 8
state 8
    $           go to state 9
state 9
    $default    accept
```

Die Interpretation der Zustandsbeschreibungen ergibt sich aus dem in 5.4.1, S. 107 beschriebenen Aufbau eines LR-Parsers. Wir betrachten dazu Zustand 1 als Beispiel. Aus den beiden ersten Zeilen

```
start  ->  kommandos .    (rule 1)
kommandos  ->  kommandos . schritt    (rule 2)
```

kann man die „Handles“ entnehmen, die der Parser auf dem Symbolstack aufzubauen versucht, entweder die rechte Regelseite von Regel 1 oder die rechte Seite von Regel 2. Die Symbole links des Punktes (ggf. auch "_") sind schon auf dem Stack, hier also das Nonterminal *kommandos*.

Die Entscheidung für eine der beiden Regeln ist hier durch die Vorausschau auf das nächste Token eindeutig zu treffen:

- Kommt noch ein Schritt hinzu (Token 'o','s','w','n') ist das zweite Handle korrekt
- Ist die Schrittfolge beendet, kann auf *start* reduziert werden.

Die nächsten 5 Zeilen der Zustandsbeschreibung geben die durchzuführende Aktion an, `$default` steht dabei für alle Vorausschau-Symbole:

```
'o'        shift, and go to state 2
's'        shift, and go to state 3
'w'        shift, and go to state 4
'n'        shift, and go to state 5

$default   reduce using rule 1 (start)
```

Die letzte Angabe in der Zustandsbeschreibung enthält keine shift- oder reduce-Aktion, sondern spezifiziert einen vom Symbolstack abhängigen Zustandübergang. In der schematischen Darstellung des LR-Parsers sind diese Übergange als *Sprung-Tabelle* bezeichnet.

Diese Tabelle wird nach jeder Reduktion benötigt: Bei einer Reduktion einer rechten Regelseite der Länge *n* werden *n* Symbole vom Symbolstack und *n* Zustände vom Zustandsstack heruntergenommen. Oben auf dem Zustandsstack erscheint dadurch ein alter Zustand *s*.

Nehmen wir an, *s* repräsentiert *x.y*. Falls inzwischen *y* durch Reduktion auf dem Stack steht, wird der in der Sprung-Tabelle angegebene Folgezustand *xy.* repräsentieren. In unserem Beispiel heißt das für Zustand 1:

```
schritt    go to state 6
```

Falls nach einer Reduktion Zustand 1 oben auf dem Zustandsstack steht, und gleichzeitig das Symbol *schritt* oben auf dem Symbolstack, geht der Parser über in Zustand 6 (`kommandos -> kommandos schritt .`).

Eine Trace-Ausgabe des generierten Parsers zeigt das Listing 6.5.1, S. 147.

6.6 Symboltabelle

Eine Symboltabelle enthält zu jedem Bezeichner einen Eintrag mit allen für die Übersetzung relevanten Attributen. Dieser Eintrag wird bei der Analyse der Definitionsstelle des Bezeichners erzeugt. Bei jeder Verwendungsstelle eines Bezeichners wird der Compiler den Symboltabelleneintrag lesen, um die korrekte Verwendung überprüfen oder den richtigen Code erzeugen zu können.

6.6.1 Symboltabellen-Einträge

Anhand der *Bezeichnerklasse* stellen wir fest, ob Bezeichner Variablen, Typen, Prozeduren oder sonstige benannte Objekte repräsentieren. Ein Bezeichnereintrag muss in jedem Fall den Namen und die Klasse enthalten. Welche weiteren Attribute für die Übersetzung benötigt werden, ist von der Klasse abhängig.

Beispiel:

Klasse	**Attribute**
Variable	Typ, Speicherdresse
Funktion	Resultatstyp, Speicheradresse, für jeden Parameter: Typ + ggf. Übergabeverfahren
Konstante	Typ, Wert

Welche Bezeichnerklassen existieren, hängt im wesentlichen von der Quellsprache ab. Die jeweils benötigten Attribute ergeben sich aus den vom Compiler für Typprüfung und Codeerzeugung benötigten Informationen. Man beachte, dass Adressen erst bei der Codeerzeugung vergeben werden, während die anderen Attribute aus der Definition entnommen werden.

6.6.2 Operationen

Der Datentyp Symboltabelle könnte etwa folgende Operationen besitzen:

- neue Tabelle erzeugen: S = Erzeuge_Symtab ()
- Tabelle löschen: Entferne_Symtab (S)

Listing 6.2 Ausgabe des generierten Parsers für die Eingabe ”oon”

```
roboter: make
bison -dtv roboter.y
cc -o roboter -g roboter.tab.c
roboter: roboter
Starting parse
Entering state 0
Reducing via rule 3 (line 20),  -> kommandos
state stack now 0
Entering state 1
Reading a token: oon
Next token is 111 ('o')
Shifting token 111 ('o'), Entering state 2
Reducing via rule 4 (line 23), 'o'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Next token is 111 ('o')
Shifting token 111 ('o'), Entering state 2
Reducing via rule 4 (line 23), 'o'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Next token is 110 ('n')
Shifting token 110 ('n'), Entering state 5
Reducing via rule 7 (line 26), 'n'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Now at end of input.
Reducing via rule 1 (line 17), kommandos  -> start
X=2, Y=1
state stack now 0
Entering state 7
Now at end of input.
Shifting token 0 ($), Entering state 8
Now at end of input.
```

- neuen Eintrag erzeugen:
  Eintrag = Erzeuge_Eintrag(S, IDENT, Klasse)

  Man beachte dass beim Eintragen eines neuen Bezeichners dessen Klasse bekannt ist.

- Eintrag für Bezeichner IDENT in einer Tabelle S suchen:
  Eintrag = Lookup(IDENT, S)

- Für jedes benötigte Bezeichner-Attribut *X*:
  Setze_*Attribut_X* (Eintrag)

- Für jedes benötigte Bezeichner-Attribut *X*:
  Attribut = *Attribut_X* (Eintrag)

Nach der Compilierung wird die Symboltabelle i.d.R. nicht mehr benötigt. Ausnahme: Falls eine symbolischer Debugger eingesetzt werden soll, müssen die Bezeichnerinformationen zur Laufzeit weiter verfügbar sein.

6.6.3 Symboltabellen und Gültigkeitsbereiche

In höheren Programmiersprachen sind unterschiedliche Gültigkeitsbereichs-Konzepte verwirklicht. Typischerweise lassen sich globale und Unterprogramm-lokale Definitionen unterscheiden. Der Sprachdefinierer entscheidet, ob eine beliebige Hierarchie von Gültigkeitsbereichen möglich ist (z.B. verschachtelte Unterprogramm-Definitionen in Pascal oder ADA), oder nur eine flache Struktur (global – lokal).

In jedem Fall muss die Bindung einer Definition an einen Gültigkeitsbereich in der Symboltabelle zum Ausdruck kommen. Konzeptuell ist für jeden Gültigkeitsbereich eine separate Symboltabelle vorzusehen.

Beispiel C:

- pro Modul eine globale Symboltabelle mit den Einträgen für globale Variablen, Typbezeichner und Funktionsbezeichner

- pro Funktionsdefinition ein Symboltabelle mit den lokalen Bezeichnern

Ob die einzelnen Symboltabellen dann auch durch separate Datenstrukturen implementiert werden, oder ob eine einzige Datenstruktur für alle Gültigkeitsbereiche verwendet wird, ist eine Implementierungsentscheidung.

6.6.4 Implementierung

Eine Symboltabelle lässt sich als verkettete Liste von Symboltabellen-Einträgen implementieren. Die Suche eines Bezeichnereintrags ist eine sehr häufige Operation, so dass sich die Implementierung eines effizienten Suchverfahrens anbietet (z.B. Hashverfahren oder sortierter Baum).

7 Laufzeitsysteme

Der Maschinen-Code eines Programms ist oft nicht ohne Unterstützung durch ein Laufzeitsystem möglich, das etwa die dynamische Speicherverwaltung (z.B. C, Pascal) oder Scheduling-Aufgaben in Sprachen mit Parallelisierungskonzepten (z.B. Multitasking in ADA) übernimmt.

Exemplarisch betrachten wir die Speicherverwaltung für Pascal.

Die Semantik einer Variablen in Pascal ist mathematisch gesehen durch eine zweistufige Abbildung gekennzeichnet:

- Die erste Abbildung, das „Environment“ (Umgebung), ordnet dem Bezeichner einen Speicherplatz zu
- Die zweite Abbildung, der Zustand („state“) , ordnet dem Speicherplatz einen Wert zu

$$Variablenbezeichner \overset{Environment}{\longrightarrow} Speicherplatz \overset{State}{\longrightarrow} Wert$$

Eine Wertzuweisung `x:=5` verändert den Zustand, hat jedoch keinen Einfluss auf das Environment. Allerdings ist in Pascal wie bei vielen anderen Programmiersprachen auch das Environment eines Bezeichners dynamischen Änderungen unterworfen.

Das Environment in Pascal

Die Zuordnung eines Speicherplatzes zu einem Bezeichner in Pascal-Programmen hat sowohl einen statischen als auch einen dynamischen Aspekt.

- Für Pascal gilt das Prinzip der **statischen Bindung**:

  Ein Bezeichner wird einer Deklaration aufgrund der statischen Gültigkeitsbereichsregeln zugeordnet. Hier ist also die Verschachtelung der Unterprogramm-Deklarationen im Programmtext maßgebend.

- Der dynamische Aspekt des Environment zeigt sich bei Unterprogramm-Aufrufen:

  Jeder Unterprogramm-Aufruf definiert ein neues Environment für die lokalen Variablen des Unterprogramms.

Die Variablendeklaration `i: integer` innerhalb eines rekursiven Unterprogramms führt zu einer neuen Speicherplatzzuordnung für jede Rekursionsebene.

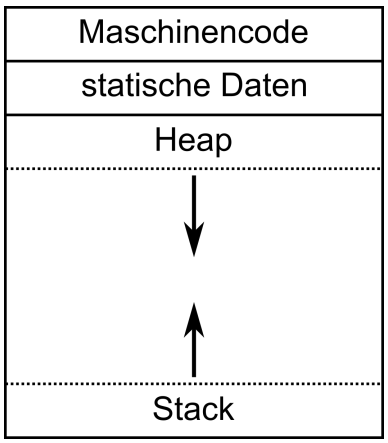
Somit kann die Bestimmung des Werts einer lokalen Variablen X nach folgendem Schema erfolgen:

1. Bestimmung des Unterprogramms U, das die Deklaration von X enthält (statisch, durch Compiler)
2. Bestimmung des Speicherplatzes („Bindung“ der Variablen) gemäß der dynamischen Aufruffolge (aktuelle „Inkarnation“ von U bestimmen, durch Laufzeitsystem)
3. Bestimmung des Werts

7.1 Speicherverwaltung

7.1.1 Speicheraufteilungsschema

Typisch für Pascal und C ist folgendes Aufteilungsschema:



Der Stack enthält für jeden noch nicht abgeschlossenen Unterprogramm-Aufruf einen Eintrag, den „activation record“ (auch Aktivierungsrahmen oder „frame“) des Unterprogramm-Aufrufs.

Dieser enthält alle für den Aufruf benötigten Daten bzw. Speicherplätze z.B.

- Rückgabewert

- aktuelle Parameter
- Kontroll-Verweis (Verweis auf vorangehenden Rahmen)
- Zugriffs-Verweis (Verweis auf obersten Rahmen des statisch umfassenden Unterprogramms)
- „save area“ zwischengespeicherter Status der Maschine bei Aufruf
- Speicher für lokale Variablen
- Speicher für Zwischenergebnisse

7.1.2 Speicher-Layout für lokale Daten

Aus dem bekannten Speicherbedarf für die Werte der Basistypen kann der Übersetzer den Speicherbedarf für strukturierte Variablen (z.B. Felder, Records, Varianten) ermitteln.

Bei Bearbeitung eines Deklarationsteils wird jeder Variablen eine relative Adresse (Distanz, „offset“), bezogen auf den Beginn des Speicherbereichs für die lokalen Daten dieses Unterprogramms zugeordnet.

Dabei sind maschinenspezifische Ausrichtungs-Anforderungen („alignment“) zu beachten (z.B. Hauptspeichereinteilung in 4-Byte-Maschinenworte, Wort-Ausrichtung für *real* und *integer*-Werte).

Die Adresse einer lokalen Variablen zur Laufzeit kann dann aus Rahmen-Adresse und Distanz berechnet werden.

Für Strukturen deren Länge erst nach erfolgtem Sprung in das Unterprogramm ermittelt werden kann (z.B. arrays in ALGOL68), ist eine Sonderbehandlung erforderlich:

Anstatt im Rahmen Platz für die Strukturen vorzusehen, wird Platz für einen Verweis auf deren Anfangsadresse reserviert. Die Strukturen werden vom aufgerufenen Unterprogramm, am Ende des Rahmens abgelegt (beliebig viel Platz), der Zugriff erfolgt indirekt über den zugehörigen Verweis.

7.1.3 Aktionen beim Unterprogramm-Aufruf

P_1 ruft P_2 auf.

Aufruf:

- P_1 wertet die aktuellen Parameter aus und speichert sie im Rahmen von P_2 ab

- P_1 trägt die Rücksprung-Adresse in den Rahmen von P_2 ein
- P_1 trägt den alten *top of stack*-Wert in den Rahmen von P_2 ein
- P_1 erhöht den *top of stack*-Wert so, dass er in den Rahmen von P_2 hinter die Rücksprungadresse verweist
- P_2 sichert die Registerwerte in seinen Rahmen
- P_2 initialisiert lokale Daten und beginnt mit der Ausführung des Unterprogramm-Rumpfs

Rücksprung:

- P_2 speichert sein Resultat direkt am Anfang seines Rahmens ab
- P_2 setzt die Register auf die alten Werte zurück
- P_2 setzt den *top of stack*-Zeiger zurück
- P_2 springt zur übergebenen Rücksprung-Adresse
- P_1 findet das Resultat direkt hinter dem eigenen Rahmen

Man beachte, dass dieses Verfahren den Aufruf von Unterprogrammen mit einer variablen Parameter-Anzahl zulässt.

7.2 Zugriff auf globale Adressen

Bei statischer Bindung ist ein besonderer Mechanismus für den Zugriff auf globale Adressen im Stack notwendig.

Sicherlich muss beim Zugriff auf eine solche globale Variable ein Unterprogramm-Rahmen existieren, in dem der Wert gespeichert ist. Bei rekursiven Unterprogrammen können beliebig viele solcher Rahmen auf dem Stack stehen, der letzte ist der richtige.

Beispiel:

```
program demo;                          Ebene 0
  procedure p1;                        Ebene 1
    var X: real;
    procedure p11;                     Ebene 2
      begin X:=5.0 end;
    procedure p12;                     Ebene 2
      var X: real;
      begin p11 end;
    begin
    ... if ... then p1 else p12; ...
    end;
  begin
    ... p1; ...
  end.
```

8 Code-Erzeugung

8.1 Einführung in die Maschinencode-Generierung und Optimierung

8.1.1 Abstrakte Maschinen und Zwischencode-Generierung

Ein Ein-Phasen-Compiler wird den Code direkt während der Analyse erzeugen. Bei einem Frontend-Backend-Compiler erzeugt das Frontend zunächst einen Syntaxbaum. Der Baum ist dann die Basis für die Code-Erzeugung durch das Backend.

Unabhängig von der Phaseneinteilung wird bei der Code-Erzeugung in vielen Fällen nicht gleich Maschinencode sondern zunächst ein Zwischencode erzeugt. Dieser basiert in der Regel auf den Datenstrukturen und dem Befehlssatz einer abstrakten virtuellen Maschine

Die Vorteile:

- Optimierungen auf hoher Abstraktionsebene möglich
- bei Portierung auf andere Maschine ist nur das maschinenanhängige „back end“ neu zu schreiben
- bei Implementierung ähnlicher Sprachen kann evtl. das gleiche „back end“ verwendet werden (z.B. für C und Pascal)
- bei einer Prototyp-Implementierung einer neuen Sprache kann man den Zwischencode zunächst interpretieren (bei logischen und funktionalen Sprachen wird oft auf Maschinencode-Generierung ganz verzichtet)

Bekannte Zwischenrepräsentationen sind:

- 3-Adress-Code (prozedurale Sprachen)
- SECD-Maschine (funktionale Sprachen)
- „Warren“-Maschine (logische Sprachen)

8.1.2 Code-Erzeugung für Stack-Maschinen

Als Vorüberlegung betrachten wir das Prinzip der Code-Erzeugung für Stack-Maschinen anhand einiger Beispiele. Eine Stack-Maschine benutzt für die Speicherung von Zwischenergebnissen einen Stack (anstelle von Registern). Wir gehen der Einfachheit halber von einem Wort-adressierten Hauptspeicher aus.

Wir verwenden für unsere Beispiele folgende Befehle:

Befehl	**Wirkung**	**Seiteneffekt**
PUSH w	Wert w wird auf dem Stack oben angefügt	`TOP++`
LOAD a	Lade Inhalt des Speicherworts a auf den Stack	`TOP++`
STORE a	Lege den oben auf dem Stack stehenden Wert in Speicherwort a ab	`TOP--`
ADD	Ersetze die beiden oben stehenden Werte durch ihre Summe	`TOP--`
SUB	Ersetze die beiden oben stehenden Werte durch ihre Differenz	`TOP--`
MUL	Ersetze die beiden oben stehenden Werte durch ihre Produkt	`TOP--`

Betrachten wir nun einige Code-Beispiele dazu:

Quelltext	**Stackmaschinencode**
`5`	`PUSH 5`

`5+3`	`PUSH 5`
	`PUSH 3`
	`ADD`

`j-2*i`	`LOAD Adr(j)`
	`PUSH 2`
	`LOAD Adr(i)`
	`MULT`
	`SUB`

`k:=2*i`	`PUSH 2`
	`LOAD Adr(i)`
	`MULT`
	`STORE Adr(k)`

Wir können die Codeerzeugung sehr einfach syntaxorientiert formulieren. Wir verwenden dabei ein Semikolon für das *Aneinanderhängen* von Maschinencodese-

quenzen. Die dabei verwendeten Attribute sollten selbsterklärend sein.

Beispiele:

CODE(`IntConst`)	=	`PUSH` *IntConst.Wert*
CODE(`VarIdent`)	=	`LOAD` *VarIdent.Adresse*
CODE(`Expr1+Expr2`)	=	CODE(`Expr1`) ; CODE(`Expr2`) ; `ADD`
CODE(`Expr1-Expr2`)	=	CODE(`Expr1`) ; CODE(`Expr2`) ; `SUB`
CODE(`VarIdent:=Expr`)	=	CODE(`Expr`) ; `STORE` *VarIdent.Adresse*

Die im nächsten Abschnitt beschriebene Rechnerarchitektur ist keine Stackmaschine, sondern ein virtueller RISC-Rechner mit 32 Mehrzweckregistern.

Die Codererzeugung benutzt dabei das oben erläuterte Prinzip. Dazu wird einfach der vorhandene Registersatz als Stack verwendet, der Stackpointer TOP verweist auf das letzte belegte Register.

Für den Fall, dass die Register nicht ausreichen, werden wir später eine geeignete Lösung suchen. Im Prinzip müssen bei Platzmangel die Registerinhalte vorübergehend in den Hauptspeicher kopiert werden.

8.1.3 Zielrechner-Architektur

Als Zielrechner betrachten wir die in [Wirth] definierte virtuelle RISC-Maschine. Dieser Rechner hat 32 Mehrzweck-Register (R0,...,R31) von 32 Bit Breite. R0 ist immer gleich 0. R31 wird von Unterprogramm-Einsprungbefehlen implizit mit der Rücksprungadresse geladen (Link-Register). Alle anderen Register sind frei verfügbar.

Das Instruktionsregister IR enthält den jeweils aktuellen Maschinenbefehl. Der Programmzähler PC enthält die Adresse des nächsten auszuführenden Befehls.

Der Hauptspeicher besteht aus 4-Byte-Wörtern und wird Byte-adressiert (Speicheradressen sind also immer Vielfache von 4).

Die Maschine hat die folgenden 3 Befehlsformate:

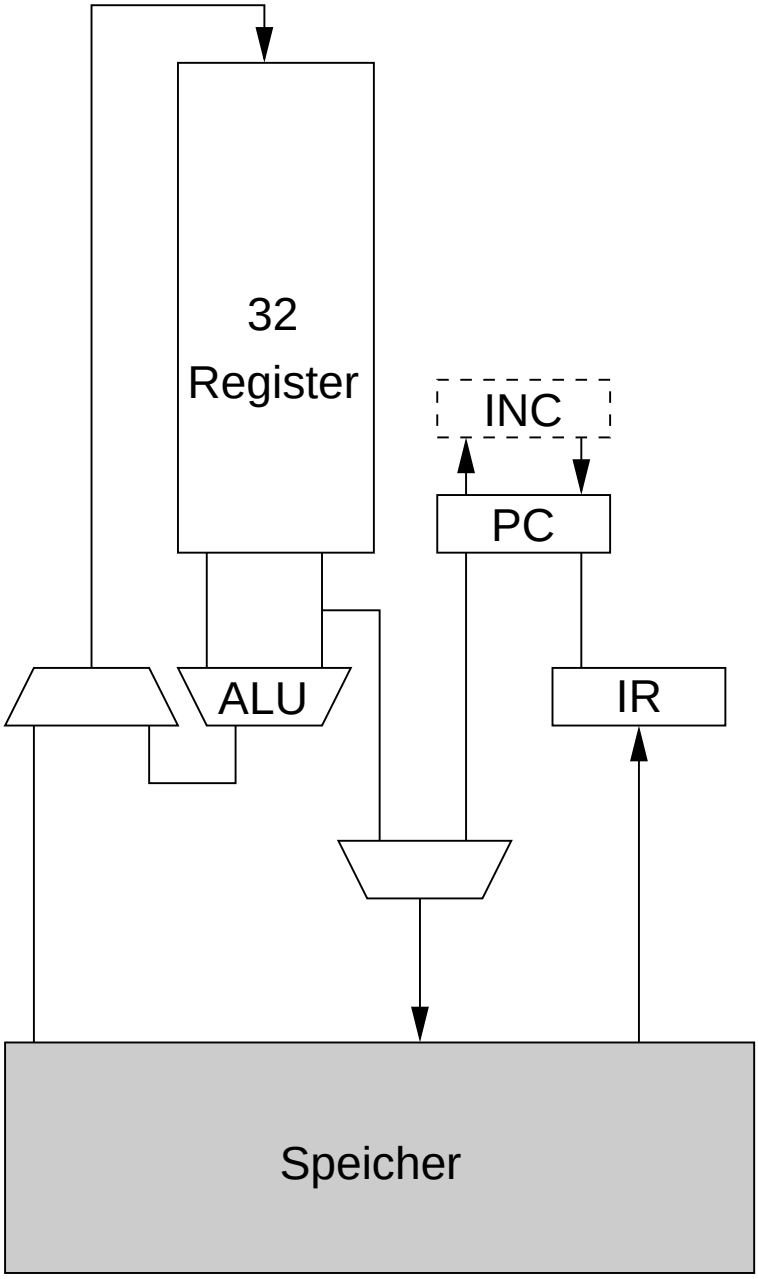
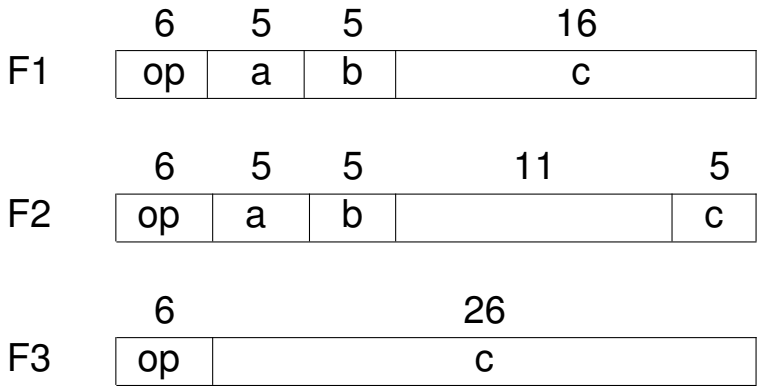


Abbildung 1: Architekturschema der virtuellen Maschine



Registerbefehle

Register + Register (Format F2)			**Register + Direktoperand** (Format F1)		
ADD	a,b,c	R.a:=R.b+R.c	ADDI	a,b,d	R.a:=R.b+c
SUB	a,b,c	R.a:=R.b-R.c	SUBI	a,b,d	R.a:=R.b-c
MUL	a,b,c	R.a:=R.b*R.c	MULI	a,b,d	R.a:=R.b*c
DIV	a,b,c	R.a:=R.b DIV R.c	DIVI	a,b,d	R.a:=R.b DIV c
MOD	a,b,c	R.a:=R.b MOD R.c	MODI	a,b,d	R.a:=R.b MOD c
CMP	a,b,c	R.a:=R.b-R.c	CMPI	a,b,d	R.a:=R.b-c
CHK	a,c	$0 \leq R.a < R.c$	CHKI	a,d	$0 \leq R.a < c$
AND	a,b,c	R.a:=R.b & R.c	ANDI	a,b,d	R.a:=R.b & c
BIC	a,b,c	R.a:=R.b & ~R.c	BICI	a,b,d	R.a:=R.b & ~c
OR	a,b,c	R.a:=R.b OR R.c	ORI	a,b,d	R.a:=R.b OR c
XOR	a,b,c	R.a:=R.b XOR R.c	XORI	a,b,d	R.a:=R.b XOR c
LSH	a,b,c	R.a:=LSH(R.b,R.c)	LSHI	a,b,d	R.a:=LSH(R.b,c)
ASH	a,b,c	R.a:=ASH(R.b,R.c)	ASHI	a,b,d	R.a:=ASH(R.b,c)

Anmerkung

- Der 16-Bit Direktoperand (Format F1) ist mit *d* bezeichnet, in der Semantikspezifikation steht stattdessen *c*. *c* ist die auf 32 Bit erweiterte Darstellung von *d*.
- CMP und SUB verhalten sich bei Überlauf unterschiedlich: SUB zeigt Überlauf an, CMP nicht.
- LSH steht für *Logical Shift*, ASH für *Arithmetic Shift*. Positive Werte für R.c (bei ASHI und LSHI *c*) bedeuten eine Verschiebung nach links, negative Werte eine Verschiebung nach rechts.

Speicherbefehle (F1-Format)

LDW	a,b,c	R.a := MEM[R.b+c]	load word
LDB	a,b,c	R.a := MEM[R.b+c]	load byte
STW	a,b,c	MEM[R.b+c] := R.a	store word
STB	a,b,c	MEM[R.b+c] := R.a	store byte
PSH	a,b,c	R.b := R.b-c; MEM[R.b]:=R.a	push stack
POP	a,b,c	R.a := MEM[R.b]; R.b:=R.b+c	pop stack

Sprungbefehle (F1-Format, Adresse PC-relativ)

BEQ	a,c	Branch to c, if R.a = 0
BNE	a,c	Branch to c, if R.a != 0
BLT	a,c	Branch to c, if R.a $<$ 0
BGE	a,c	Branch to c, if R.a $\geq$ 0
BGT	a,c	Branch to c, if R.a $>$ 0
BLE	a,c	Branch to c, if R.a $\leq$ 0

BSR	c	Save PC in R.31, branch to c (F1, Adresse PC-relativ)
JSR	c	Save PC in R.31, jump to c (F3, Adresse absolut)
RET	c	Jump to address in R.c (F2, Adresse absolut)

8.1.4 Grundlagen der Codeerzeugung für die virtuelle RISC-Maschine

Wie oben erläutert, wird das Stackmaschinen-Prinzip verwendet, wobei die Register R1 bis R30 den Stack bilden. Der Compiler muss sich nur das letzte belegte Register *r* merken.

Statt des Stackmaschinenbefehls `ADD`, wird jetzt der RISC-Maschinenbefehl

```
ADD r-1,r-1,r
```

verwendet:

- der zweite Operand der Addition steht ganz oben auf dem Stack. d.h. im Register *r*
- der erste Operand der Addition steht auf dem Stack unmittelbar darunter, also im Register *r-1*
- beide Operanden werden vom Stack entfernt – Register *r* und *r-1* werden frei – die Summe auf dem Stack gespeichert, im Register *r-1*, dem ersten freien Register

Gegenüberstellung

Quelltext	Stackmaschinencode	RISC-Code	Seiteneffekt
			(nach Befehlsausgabe)
5	PUSH 5	ADDI r+1,0,5	r++
5+3	PUSH 5	ADDI r+1,0,5	r++
	PUSH 3	ADDI r+1,0,3	r++
	ADD	ADD r-1,r-1,r	r--
j-2*i	LOAD Adr(j)	LDW r+1,0, Adr(j)	r++
	PUSH 2	ADDI r+1,0,2	r++
	LOAD Adr(i)	LDW r+1,0, Adr(i)	r++
	MULT	MUL r-1,r-1,r	r--
	SUB	SUB r-1,r-1,r	r--
k:=2*i	PUSH 2	ADDI r+1,0,2	r++
	LOAD Adr(i)	LDW r+1,0, Adr(i)	r++
	MULT	MUL r-1,r-1,r	r--
	STORE Adr(k)	STW r,Adr(k)	r--

Falls R1 das erste freie Register ist, ergibt sich also zur Wertzuweisung

```
k := j-2*i
```

folgender Code:

Befehl	Wirkung	Stack	
LDW	R1,R0,Adr(j)	R1 := j	j
ADDI	R2,R0,2	R2 := 2	j,2
LDW	R3,R0,Adr(i)	R3 := i	j,2,i
MUL	R2,R2,R3	R2 := R2 * R3	j, 2*i
SUB	R1,R1,R2	R1 := R1 - R2	j - 2*i
STW	R1,R0,Adr(k)	k := R1	--

8.1.5 Erste Optimierungen – Verzögerte Code-Erzeugung

Das oben verwendete Schema berücksichtigt bei den Operatoren nicht die Adressierung der Operanden. Schon deshalb ist der Code nicht sonderlich effizient. Der Ausdruck

```
2*i
```

wird übersetzt in

```
ADDI R2,0,2
LDW  R3,0,Adr(i)
MUL  R2,R2,R3
```

Effizienter wäre dagegen

```
LDW  R2,0,Adr(i)
MULI R2,R2,2
```

Dazu ist eine Fallunterscheidung notwendig:

- Operand ist Konstante
- Operand ist Variable
- Operand ist Zwischenergebnis

Im ersten Fall kann dann für die Operation ein *Immediate*-Befehl (Befehl mit Direktoperand) erzeugt werden. Falls beide Operanden Konstanten sind, kann der Compiler natürlich den Wert sofort bestimmen.

Wie wird die für die Fallunterscheidung notwendige Information berechnet ?

Wir benötigen für jeden Ausdruck ein Attribut *ort*, das bei der Code-Erzeugung Aufschluss über die Speicherung des Wert gibt:

- *ort.modus* gibt die Adressierungsart an:
  - direkt (Konstante)
  - Hauptspeicher (Variable)
  - Register (Zwischenwert)
- *ort.wert* enthält den Wert, falls Modus=direkt
- *ort.adresse* enthält die Speicheradresse, falls Modus=Hauptspeicher
- *ort.register* enthält die Registernummer, falls Modus=Register

Mit dieser Information kann man die Code-Erzeugung für Konstantenwerte verzögern bzw. ganz unterdrücken, wenn bessere Lösungen existieren. Die Verzögerung besteht hier darin, dass nicht sofort beim Auftreten einer Konstante *k* die Maschinenanweisung

```
ADDI r,0,k
```

erzeugt wird, sondern erst bei der Anwendung eines Operators, abhängig von den *ort*-Attributen der Operanden über den zu erzeugenden Code entschieden wird.

Beispiele:

Wir betrachten die Subtraktion A - B. Jede Kombination A.*ort.modus*, B.*ort.modus* wird gesondert betrachtet und möglichst effizient behandelt. Einige Kombinationen:

A.*ort*	B.*ort*	**Behandlung**	**Resultat**
direkt *w1*	direkt *w2*	keine Code-Erzeugung, Compiler berechnet Wert	direkt *w1+w2*
register X	register Y	SUB-Befehl erzeugen (SUB X,X,Y), Y freigeben	Register X
register X	direkt *w*	SUBI-Befehl erzeugen (SUBI(X,X,*w*))	Register X
register X	Speicher *a*	Register Y für B reservieren, B laden (LDW Y,0,*a*), SUB-Befehl (SUB X,X,Y), Y freigeben	Register X
Speicher *a*	register X	Register Y für A reservieren, A laden (LDW Y,0,*a*), SUB-Befehl (SUB Y,Y,X), X freigeben	Register Y
direkt *w*	register X	Register Y für A reservieren, A laden (ADDI Y,Y,w), SUB-Befehl (SUB Y,Y,X), X freigeben	Register Y

Man beachte insbesondere, dass die Buchführung über den Ort, an dem der Wert gespeichert ist, eine andere Registerverwaltung ermöglicht:

Statt die Register strikt nach dem Stackprinzip zu verwalten, kann bei Bedarf ein beliebiges freies Register reserviert werden.

9 Literatur

- Das Standardwerk zum Compilerbau („Drachenbuch“):

  Aho, Sethi, Ullmann:
  Compilers
  Principles, Techniques, and Tools
  Addison-Wesley, 1986

  (Deutsche Ausgabe in 2 Bänden: Compilerbau)

- A.W. Appel:
  Modern Compiler Implementation in Java.
  2. Ausgabe. Cambridge University Press, 2002.

- D. Grune, H.E. Bal, J.H. Jacobs, K.G. Langendoen:
  Modern Compiler Design.
  John Wiley & Sons, 2001.

- Niklaus Wirth:
  The Design of a RISC Architecture and its Implementation with an FPGA.
  https://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/RISC.pdf, rev. 2018.