Vorlesungsmodul Compilerbau
- VorlMod Comp -

Matthias Ansorg

20. März 2003 bis 26. März 2005

Zusammenfassung

Studentische Mitschrift zur Vorlesung Compilerbau bei Prof. Dr. Michael Jäger (Wintersemester 2004/2005) im Studiengang Informatik an der Fachhochschule Gießen-Friedberg. Die mündliche Diplomprüfung kann häufig Grundlagenfragen wie »Was ist ein Compiler?« enthalten.

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum Download bereitPersönliche Homepage Matthias Ansorg http://matthias.ansorgs.de/InformatikDiplom/Modul.Comp.Jaeger/Comp.pdf.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der angegebenen Quellen zu beachten.
- **Korrekturen und Feedback:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg <mailto:matthias@ansorgs.de>.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu LaTeX) unter Linux geschrieben und mit pdfLaTeXals pdf-Datei erstellt. Grafiken wurden mit dem Programm xfig unter Linux erstellt und als pdf-Dateien exportiert.
- **Dozent:** Prof. Dr. Michael Jäger.
- **Verwendete Quellen:** <quelle> {<quelle>}.
- **Klausur:**
  - Die Klausur orientiert sich nicht mehr an den (eher leichten) Klausuren der vorigen Jahre.
  - Bisher durften in den Klasuren bei Prof. Dr. Jäger keine Unterlagen oder sonstigen Hilfsmittel verwendet werden.
  - Die Klausur wird nach den Semesterferien gechrieben.

Inhaltsverzeichnis

1 Organisation **2**

2 Einführung **3**

2.1 Lernziele 4
2.2 Inhalt 4
2.3 Anwendungen von Compilerbautechniken 5
2.4 Programmiersprachliche Grundbegriffe 5

3 Architektur eines Compilers **6**
3.1 Phasenmodell 6
3.2 Beispiel für die Phasen der Compilierung 6

4 Die Phase der lexikalischen Analyse **7**
4.1 Reguläre Ausdrücke 8
4.1.1 Definitionen 8
4.1.2 Reguläre Ausdrücke 8
4.1.3 Beispiele zu regulären Ausdrücken 9
4.1.4 Anwendung von regulären Ausdrücken in Programmiersprachen 9

5 Die Phase der Syntaxanalyse **10**

6 Die Phase der semantischen Analyse **11**

7 Die Phase Zwischencodeerzeugung **11**

8 Die Phase Maschinencodeerzeugung **11**

9 Aufgaben und Lösungen **11**
9.1 Frame-Layout für eine SPL-Prozedur 11
9.2 SPL in ECO32-Assembler übersetzen 11

10 Algorithmen für Klausuraufgaben **13**
10.1 Grammatik in LL(1)-Grammatik transformieren 13
10.2 LL(1)-Parsertabelle bestimmen 13
10.3 Zustände eines LALR(1)-Parsers (z.B. yacc/bison-generierte Parser) 14
10.4 Abstrakter Syntaxbaum eines SPL-Programms 14

11 Fragen und Anmerkungen **15**

1 Organisation

Im Wintersemester 2004/2005:

- Hausübungen dürfen wie die Bonuaufgaben zu zweit gelöst werden.
- Man kann sich nach Belieben eine Praktikumsgruppe aussuchen. Das Praktikum wird auf den Sun-Workstations durchgeführt.
- Zur Entwicklung des SPL-Compilers enötigt man etliche Beratung. Es gibt deshalb auch Tutoren.
- Im Praktikum werden C++, flex (lex-kompatibler Scannergenerator) und bison (yacc-kompatibler Compilergenerator) als Werkzeuge verwendet. yacc ist der Compilergenerator, die zum Industriestandard geworden ist. Unter Windows können die freien Versionen von `http://www.cygwin.com` verwendet werden, unter Linux sind flex und bison standardmäßig verfügbar.
- Voraussetzung für die Klausurteilnahme: 2 Hausübungen, die fristgerecht, formgerecht und korrekt abgeliefert werden müssen. Im Somersemester 2004 wurde eine semesterbegleitende Prüfung durchgeführt (SPL-Compiler als praktischer Prüfungsteil). Das hat sich nicht bewährt und wurde für das Wintersemester 2004/2005 wieder ausgesetzt.

- Die Hausübungen sind nicht vergleichbar mit dem, was in den letzten Semestern gefordert wurde. Es sind kleine Ausschnitte des SPL[1]-Compilers zu schreiben. Sie dienen nur dazu, die Studenten dazu zu bringen, am Praktikum dranzubleiben.
- Die Teilnahmevoraussetzungen zur Klausur sind also niedrig, die Klausur wird jedoch schwer. Aus diesem Grund sollte möglichst jeder auch Bonuspunkte sammeln.
- Die Bonusaufgaben (und gleichzeitig Aufgabenstellungen für die Übungen) bestehen darin, den kompletten SPL-Compiler zu schreiben. Sie sind zerlegt in möglichst unabhängige Teilaufgaben und können einzeln oder zu zweit gelöst werden (es gibt keine Ausnahmen).
  - Die Aufgabenstellung ist minimalistisch: SPL ist eine sehr einfache Sprache, der Maschinencode wird für eine virtuelle Maschine (ECO32) erzeugt. ECO32 ist sehr stark an die MIPS-Architektur angelegt, es ist für den Compilerbauer die einfachste vorstellbare Architektur, weil sie hauptsächlich mit Registern arbeitet.
  - Die Entwicklung des kompletten SPL-Compilers ist sehr zeitaufwändig
  - Es gibt Klausuraufgaben, etwa die manuelle Generierung von Maschinencode, die man beim ECO32 Compiler lernt, die man in der Klausur benötigt.
- Näheres wird bei Ausgabe der Hausübungen und Bonusaufgaben bekanntgegeben.
- Es wird wiederum das MOS-System zur Überprüfung der Ähnlichkeit von Code verwendet. Entwickelt am MIT, ein Plagiat-Erkennungssystem. Wer Code von anderen Gruppen kopiert, desse Lösung wird nicht gewertet. Durch das Programm MOS vom MIT können solche Strukturähnlichkeiten von Code erkannt werden. Es wurde auch schon von Prof. Geisse in der Veranstaltung Compilerbau im WS 2003/2004 eingesetzt.
- Nur im letzten Drittel der Lehrveranstaltung, wenn es um die Codegenerierung geht, wird der Simulator ECO32 benötigt.
- Alle Modalitäten zur Veranstaltung Compilerbau stehen auch auf der Internetseite von Prof. Jäger.
- SPL-Compiler: Dabei wird ein Großteil der Lösung als Codegerüst vorgegeben, das ergänzt werden muss.
- Auch in den Compilerbau-Veranstaltungen bei Prof. Geisse wurde ein SPL-Compiler programmiert. Die Unterschiede zur Veranstaltung von Prof. Jäger bestehen in der Menge des vorgegebenen Codes und der Bewertung von Teillösungen.
- Die Teilnahme an den Praktika ist keine Prüfungsvoraussetzung, nur die Abgabe der Hausübungen.

2 Einführung

Die Semantik von Programmen, d.i. die Wirkung von Programmen, kann formal beschrieben werden, z.B. mit denotationaler Semantik. Dazu werden Funktionen höherer Ordnung verwendet. Alternativ kann die Semantik formal operational beschrieben werden (als Verhalten einer einfachen virtuellen Maschine) oder algebraisch. In dieser Veranstaltung wird die Semantik informal beschrieben: als

[1] straightline programming language

Beschreibung des Verhaltens von Programmen. Damit lassen sich natürlich keine Interpreter automatisch generieren wie etwa bei denotationaler Semantik. Man beachte, dass hier stets die Semantik der Programmiersprache gemeint ist, nicht die Semantik eines speziellen zu übersetzenden Programms in dieser Programmiersprache. Über letztere hat der Compiler keinerlei Kenntnis, er verarbeitet zu compilierende Programme ja einfach mechanisch.

Es gibt keinen Bereich in der Informatik, der so gut erforscht und dokumentiert wäre wie der Compilerbau. Der Prozess der Compilerentwicklung ist dadurch quasi standardisiert. Es gibt viele und gute Algorithmen auf diesem Gebiet, die auch theoretisch gut fundiert sind durch Theorien der Automaten und formalen Sprachen. Außerdem gibt es viel gute Literatur.

2.1 Lernziele

1. Informatiker nutzen Compiler und sollten eine ungefähre Vorstellung von der Funktionsweise eines Compilers haben.
2. Informatiker entwickeln Compiler.
3. Compilerbau-Algorithmen werden auch überall dort eingesetzt, wo strukturierte Eingabetexte zu verarbeiten sind, etwa bei HTML und XML-Parsern.
4. Vertiefte Kenntnis der grundlegenden Konzepte hinter Programmiersprachen, um andere Sprachen in kurzer Zeit erlernen zu können.

2.2 Inhalt

1. Architektur eines Compilers
2. Programmiersprachen-Konzepte
3. Formale Beschreibungsmethoden von Programmiersprachen. Es werden zwar keine Methoden zur formalen Beschreibung der Semantik besprochen, sondern Methoden zur Strukturerkennung.
    - reguläre Ausdrücke (Muster zur Beschreibung von Wortmengen). Werden verwendet zur Erkennung von Tokens (Grundbausteine der Programme, Syntax). Tokens könnten auch mit Grammatik beschrieben werden, jedoch sind reguläre Ausdrücke nötig, um einen schnellen Compiler zu bauen.
    - Grammatik in EBNF. Werden verwendet zur Beschreibung, wie hierachische Konstrukte aus Tokens korrekt aufgebaut werden.
4. Implementierung eines Scanners, der mit regulären Ausdrücken verschiedene Tokens erkennt, d.h. die lexikalische Analyse durchführt.
5. Implementierung eines Parsers (der die Syntaxanalyse durchführt). Dazu gibt es zwei völlig unterschiedliche Klassen von Parser-Algorithmen:
    - Bottom-up Algorithmen. In dieser Veranstaltung wird der Quellcode eines Bottom-up Parsers mit bison aus einer Grammatik erzeugt.
    - Top-down Parser. Ein solcher Top-down Parser wird in dieser Veranstaltung implementiert, weil es einfacher und schneller geht als bei Bottom-up Parsern.
6. Kennenlernen von yacc bzw. bison (Parsergeneratoren)

7. Kennenlernen von lex und flex (Scannergeneratoren)

8. semantische Analyse: Typberechnungen und interne Repräsentation von Typen im Compiler. Typprüfungen sind recht komplex, weil es benutzerdefinierte hierachisch gegliederte Typen geben kann.

9. syntaxorientierte Übersetzung. Das bedeutet, die Syntaxspezifikation enthält neben der eiegntlichen Grammatik auch weitere Aktionen, die der Compiler durchzuführen hat (Codererzeugung, Typprüfungen usw.). Es gibt dazu zwei Ansätze, die miteinander kombiniert werden, nämlich attributierte Grammatiken und Übersetzungsschemata. Durch diese weiteren Aktionen ist es möglich, mit yacc nicht nur einen reinen Parser, sondern tatsächlich einen vollständigen Compiler zu erzeugen.

10. Einführung in die Codegenerierung. Die Erzeugung von Maschinencode geschieht am vereinfachten Modell einer MIPS-RISC-Architektur. Codeerzeugung für RISC-Maschinen ist wesentlich einfacher als für CISC-Maschinen.

2.3 Anwendungen von Compilerbautechniken

- Compiler. Ein Compiler ist ein Programm, das einen Quelltext aus einer höheren Programmiersprache in ein Maschinenprogramm (oder in Assembler) übersetzt.

- Interpretierer. Ein Interpretierer interpretiert den Quelltext selbst statt ihn in Maschinencode zu übersetzten, der dann vom Prozessor ausgeführt würde. Java wird dagegen zuerst compiliert (in Maschinencode einer virtuellen Maschine, den Java-Bytecode) und der Java-Bytecode dann interpretiert durch die Java Virtual Machine.

- Struktur-Editor. Ein Editor für strukturierte Texte, der die Struktur der zu erstellenden Texte kennt und entsprechende Unterstützung bietet wie etwa Syntax-Highlighting.

- WWW-Browser. Er enthält einen Parser, der ganz entsprechend einem Parser für Compiler gebaut ist.

- Programme für Text-Markup-Sprachen: XML, SGML, LaTeX.

- SQL-Optimierer, der aus SQL-statements einen datenbankspezifischen Zugriffsplan erzeugt.

- Reverse Engineering. Viele Firmen betreiben riesige Mengen an COBOL-Programmen, die schlecht dokumentiert sind und deren Programmierer nicht mehr leben. Um diese Programmen zu warten und zu erweitern, bedient man sich Softwaretools, die den vorhandenen Code natürlich auch analysieren müssen.

- Seitenbeschreibungssprachen wie PDF, Postscript, HPGL.

- Berechnung von Funktions- und Pfadüberdeckungen für Regressionstest.

2.4 Programmiersprachliche Grundbegriffe

Dereferenzierung bedeutet: »Container zu wert«; »entferne den Container, gebe mir den Wert«, oder, wenn man sich beide als ineinander geschachtelt vorstellt: »entferne den Container«, übrig bleibt sein Wert. Werte von Containern können wieder Container sein - so bei Pointern und Pointern auf Pointern und Pointern dritten und höheren Grades. Auf der linken Seite einer Zuweisung muss stets ein Container stehen (sog. »L-Value«), auf der rechten Seite der Inhalt eines Containers (sog. »R-Value«,

ermittelt durch automatische Dereferenzierung). Das bedeutet nicht, dass auf der linken Seite einer Zuweisung überhaupt keine Dereferenzierung erlaubt ist; denn das Ergebnis einer Dereferenzierung kann ein Container sein, wie in »`*zeiger = *zeiger * 2;`«.

3 Architektur eines Compilers

3.1 Phasenmodell

Es gibt unterschiedlich grobe Modelle zur Beschreibung von Compilern. Eine sehr grobe Einteilung:

1. Compiler-Frontend: Der Teil eines Compilers, der sich am Quellcode orientiert.
2. Compiler-Backend: Der Teil eines Compilers, der sich mit der Codegenerierung befasst.

Ein feineres Modell ist die Unterteilung in Phasen. Die Zwischendarstellung ist gegenüber dem Quelltext eine redundanzfreie Darstellung, ohne jeden »syntactic sugar«. Sie ist etwa durch Verwendung einer Baumstruktur optimiert für die Verarbeitung in der Maschine. Dabei wird die Symboltabelle als besondere compilerspezifische Datenstruktur verwendet, außerdem weitere übliche Dinge wie Bäume, Listen, Hashtabellen. Symboltabellen sind notwendig, um die Bedeutung eines Bezeichners herauszufinden. Denn Bezeichner sind »universelle Repräsentanten für etwas anderes«. Bezeichner haben eine Definitionsstelle (an der der Bezeichner mit seiner Bedeutung in die Symboltabelle aufgenommen wird) und beliebig viele Verwendungsstellen, die die Informationen von der Definitionsstelle benötigen. Symboltabellen benutzen Datenstrukturen wie Hashtabellen oder B-Bäume zur effizienten Schlüssel-Wert-Zuordnung.

Es gibt Verwendungsstellen, Deklarationen und Definitionen von Variable; dies ist zu verstehen im Sinne folgender Spezialisierung: jede Deklaration ist auch eine Verwendungsstelle, jede Definition ist auch eine Deklaration.

3.2 Beispiel für die Phasen der Compilierung

Als Beispielsprache wird SPL verwendet. Sie weist große Ähnlichkeiten mit C und Pascal auf. Die Ergebnisse der einzelnen Phasen der Compilierung, in logischer Reihenfolge:

Quellcode

```
proc demo(n:int, ref k:int) {
  k:=n*3;
}
```

Tokens Hier nur der Tokenstrom, wie er von der lexikalischen Analyse für die Zuweisung erzeugt wird:

```
IDENT("k") ASGN IDENT("n") STAR INTLIT(3) SEMIC
```

Abstrakte Syntax Hier nur die Abstrakte Syntax, wie sie von der syntaktischen Analyse für die Zuweisung erzeigt wird:

Abbildung 1: Abstrakte Syntax einer Zuweisung

Es entsteht ein Syntaxbaum gemäß Abbildung 1.

Symboltabelle Hier der Zustand der Symboltabelle als Ergebnis der semantischen Analyse. Es werden Namen auf Eigenschaften abgebildet. Es gibt zwei Ebenen: Level 0 für lokale Variablen, Level 1 für eine Stufe globalere Variablen.

```
Level 0: n -> var: int
         k -> var: ref int
Level 1: demo -> proc: (int, ref int)
         int -> type: int
         main -> proc: ()
         //dazu etliche Bibliotheksprozeduren, weil sie auch im
         //Programm aufgerufen werden können
```

Variablenallokation Hier wird festgelegt, welche Variablen welche Speicherplätze bekommen. Dies wird anhand der gesamten Prozedur demo behandelt. Ergebnis:

```
param 'n': fp+0 //fp = frame pointer
param 'k': fp+4
local vars: 0
proc calls: none //werden andere Prozeduren aufgerufen?
```

Assemblercode Hier nur für die Zuweisung, weil alles andere noch zu kompliziert ist.

```
add    $8,$25,4    ;berechnet wird fp+4, die Adresse von k. fp in Register $25!
ldw    $8,$8,0     ;einfach indirekter Speicherzugriff: Inhalt von k := Referenz
add    $9$25,0     ;berechnet wird fp+0, die Adresse von n. fp in Register $25!
ldw    $9,$9,0     ;Umwandlung Adresse in Inhalt - Wert holen
add    $10,$0,3    ;Konstante 3 nach $10, Register $0 liefert nach
                   ;RISC-Konvention immer "0"
mul    $9,$9,$10   ;$9 <- $9 * $10
stw    $9,$8,0     ;mem[$8+0] <- $9
```

Maschinenprogramm Adressen von Daten und Funktionen werden im Assemblerprogramm noch symbolisch dargestellt, im Maschinenprogramm jedoch numerisch. Das ist der wesentliche Unterschied zwischen beiden Darstellungsformen; der Assembler hat die Aufgabe, die Transformation durchzuführen.

4 Die Phase der lexikalischen Analyse

Die Aufgabe: konvertiere den Strom von Eingabezeichen des Quellprogramms in einen Strom von Tokens. Was sind Tokens? Nur festgelegt als informale Definition: ein Token ist das kleinste bedeutungstragende Element. Das können Einzelzeichen (etwa »+«, »3« als Integerliteral) oder Zeichenketten (wie »n123« als Name) sein. Beispiele zu Token-Typen aus konkreten Programmen:

IDENT (Bezeichner) `k ab123 das_Letzte`

NUM (Zahl) `123 0 0xAFFE`

IF (Schlüsselwort für Bedingungen) `if`

COMMA (Komma) ,

LE (kleiner oder gleich) <=

RPAREN (schließende Klammer))

Wie kann man nun spezifizieren, was in der lexikalischen Analyse als Token erkannt werden soll? Als Hilfmittel dazu dienen die regulären Ausdrücke.

4.1 Reguläre Ausdrücke

4.1.1 Definitionen

Definition »Sprache« Eine Sprache ist eine Menge von Strings. Diese Menge ist typischerweise unendlich.

Definition »String« Ein String ist eine endliche Folge von Zeichen.

Definition »Alphabet« Die Menge aller zur Verfügung stehenden Zeichen.

4.1.2 Reguläre Ausdrücke

Beispiel: Sprache, String und Aphabet. Die Sprache bestehe aus beliebig vielen a gefolgt von genau einem b:

```
Alphabet = {a,b}
Sprache = {b,ab,aab,aaab,}
```

Diese informale Beschreibung ist ungeeignet, da die Regel zur Ergänzung der Auslassung nicht explizit angegeben wurde. Die Beschreibung einer (regulären) Sprache geschieht durch »reguläre Ausdrücke«. Jeder reguläre Ausdruck steht für eine Menge von Strings.

Reguläre Ausdrücke werden rekursiv (d.i. induktiv) definiert, weil sie unendliche Mengen beschreiben. Der Endpunkt der Rekursion ist das einzelne Zeichen:

Zeichen: a steht für L(a)={a}. Der reguläre Ausdruck a steht für eine Sprache, die nichts enthält außer dem String a. Es muss zwischen regulären Ausdrücken (in Fettduck) und Strings von Sprachen unterschieden werden.

Alternative: $\beta|\gamma$ steht für $\mathtt{L}(\beta|\gamma)=\mathtt{L}(\beta)\cup\mathtt{L}(\gamma)$

Beispiel: a|b beschreibt L(a|b))=L(a)∪L(b)={a}∪{b}={a,b}

Konkatenation[2]**:** $\beta\gamma$ steht für $L(\beta\gamma) = \{st \mid s \in L(\beta) \wedge t \in L(\gamma)\}$

Beispiel[3]: $a(b \mid c)$ steht für $L(a(b \mid c)) = \{st \mid s \in L(a) \wedge t \in L(b \mid c)\} = \{st \mid s \in \{a\} \wedge t \in \{b, c\}\} = ab, ac$

Epsilon: Der leere String. Es ist ein String, der keine Zeichen enthält, nicht die leere Menge. Man verwendet als Formelzeichen ε. ε steht für $L(\varepsilon) = \{\varepsilon\}$ (leerer String).

Kleene'scher Abschluss: $\beta*$ steht für $L(\beta*) = L(\varepsilon) \cup L(\beta) \cup L(\beta\beta) \cup \ldots$

Beispiel: $(ab)*$ steht für $L((ab)*) = L(\varepsilon) \cup L(ab) \cup L(abab) \cup \ldots = \{\varepsilon, ab, abab, ababab, \ldots\}$. Aufgrund von Vorrangregeln ist $ab* = a(b*) \neq (ab)*$

[3] Wie sonst auch in Ausdrücken können Klammern verwendet werden, um Vorrang auszudrücken.

4.1.3 Beispiele zu regulären Ausdrücken

1. Binärzahlen, die Vielfache von 2 sind: $(0 \mid 1) * 0$
2. Strings aus a und b, die keine zwei a hintereinander enthalten: $b * (abb*) * (a \mid \varepsilon)$
3. Strings aus a und b, die mindestens einmal zwei a hintereinander enthalten: $(a \mid b) * aa\,(a \mid b) *$

4.1.4 Anwendung von regulären Ausdrücken in Programmiersprachen

Die bisher eingeführten Elemente der regulären Ausdrücke reichen aus, um jede reguläre Sprache zu beschreiben. Es ist jedoch mühsam. Deshalb führt man Abkürzungen ein[4]:

- $[abcd] := (a \mid b \mid c \mid d)$
- $[b - e] := (b \mid c \mid d \mid e)$
- $\beta? := (\beta \mid \varepsilon)$ Die Notation für »optional«.
- $\beta+ := (\beta\beta*)$
- $. := (\text{irgendein Zeichen außer Zeilenumbruch})$

Beispiel: die formale Spezifikation eines Scanners zur lexikalischen Analyse einer Programmiersprache. Sie besteht aus Zuordnungen von regulären Ausdrücken zu Tokens:

```
if                      -> IF
[a-z][a-z0-9]*          -> IDENT
[0-9]+                  -> NUM
0x[0-9a-fA-F]+          -> NUM
\)                      -> RPAREN
```

Problem: Eine Spezifikation wie die obige ist mehrdeutig:

- Was ist `if8`? IDENT oder IF NUM?
- Was ist `if`? IDENT oder IF?

Diese Mehrdetigkeiten werden nach den folgenden zwei Regeln (in dieser Reihenfolge) aufgelöst:

längste Übereinstimmung Danach ist `if8` ein IDENT. Die Mehrdeutigkeit `if` kann nicht entschieden werden.

erste Regel Die zuerst genannte Regel hat Vorrang. Im obigen Beispiel steht die Regel für IF vor der für IDENT, also ist `if` der Token IF.

[4] Was genau hier möglich ist, steht in der Manualpage von lex und flex.

5 Die Phase der Syntaxanalyse

Die verschiedenen Parser, vom einfachsten zum mächtigsten:

LL(1)-Parser Ein Parser, der auf LL(1)-Grammatiken arbeitet. Auch Top-Down-Parser oder prädikativer Parser (»voraussagender Parser«) genannt. Die Abkürzung bedeutet:

L (left) Die Eingabe wird von links nach rechts gelesen.

L (left) Der Syntaxbaum wird beim Parsen durch eine Linksableitung aufgebaut, d.h. es wird stets das am weitesten links stehende Nichtterminalsymbol als nächstes bearbeitet.

(1) (1) Ein Symbol Lookahead reicht aus, um eindeutig die nächste Regel zu bestimmen.

Eine Grammatik, bei der mit einem Symbol Lookahead die richtige Regel zur Ableitung eines Nichtterminals eindeutig bestimmt ist, heißt LL(1)-Grammatik. Bedingungen: (sei $\{A \to \beta_i\} = \{A \to \beta_i \mid (A \to \beta_i) \in P\}$)

- $\forall_{i \neq j} : FIRST(\beta_i) \cap FIRST(\beta_j) = \{\}$ Das verbietet u.a. Linksrekursionen und gemeinsame Präfixe in den $A \to \beta_i$.
- wenn $\epsilon \in FIRST(A)$: $FIRST(A) \cap FOLLOW(A) = \{\}$

LR(0)-Parser Auch Bottom-Up-Parser oder Shift-Reduce-Parser genannt. Idee: der Parser akzeptiert es im Gegensatz zum LL(1)-Parser, wenn während dem Einlesen des Tokenstroms mehrere Produktionen existieren, die diese Tokenfolge erzeugen können; erst wenn eine rechte Regelseite vollständig eingelesen ist, muss die Regel eindeutig bestimmt sein. Die Entscheidung »shift oder reduce« wird bei LR(0) ohne Vorausschau getroffen. Eine Grammatik ohne Shift/Reduce-Konflikte heißt »LR(0)-Grammatik«. Die Abkürzung bedeutet:

L (left) Die Eingabe wird von links nach rechts gelesen.

R (right) Der Syntaxbaum wird beim Parsen durch eine Rechtsableitung aufgebaut, d.h. es wird stets das am weitesten rechts (auf dem Symbolstack) stehende Nichtterminalsymbol als nächstes bearbeitet.

(0) Kein Symbol Lookahead.

SLR(1)-Parser Die Abkürzung bedeutet: S (simple) LR-Parser mit einem Symbol Lookahead. Shift/Reduce-Konflikte werden durch Vorausschau gelöst: ein A-Handle wird nur zu A reduziert, wenn das Lookahead-Token $t \in FOLLOW(A)$ ist.

LALR(1)-Parser Die Abkürzung bedeutet: LA (lookahead) LR(1)-Parser. Es ist eine kompaktere, funktionell fast ebenso mächtige Variante des LR(1)-Parsers. Dabei werden Zustände, die sich nur im FOLLOW-Token unterscheiden, zu einem zusammengefasst. Die Zustandsmenge wird dann nur ein Zehntel so groß.

LR(1)-Parser Sog. kanonischer LR(1)-Parser. Ein A-Handle wird nur zu A reduziert, wenn das Lookahead-Token $t \in \{FOLLOW(A)\}$ ist und At in der aktuellen Satzform möglich ist. Es wird also der Kontext der A-Handles berücksichtigt.

LR(k)-Parser Ein Parser, der auf LR(k)-Grammatiken arbeitet. Auch dieses mächtigste hier genannte Verfahren kann nicht beliebige kontextfreie Grammatiken verarbeiten: LR(k)-Grammatiken sind eine Untermenge davon. Early-Parser und Cocke-Kasami-Younger-Parser können beliebige kontextfreie Grammatiken verarbeiten.

6 Die Phase der semantischen Analyse

7 Die Phase Zwischencodeerzeugung

8 Die Phase Maschinencodeerzeugung

9 Aufgaben und Lösungen

9.1 Frame-Layout für eine SPL-Prozedur

Aufgabenstellung Bestimmen Sie das Frame-Layout (Bestandteile in der richtigen Reihenfolge mit Größe in Bytes) für den Aktivierungsrahmen der nachfolgenden SPL-Prozedur

```
x. proc x (ref i:int) {
  var k: int;
  k:=i+1;
  i:=k;
  printi(i);
}
```

Quelle: [2, 2004-WS, Bl.8, Aufg.6].

Lösung

Adresse	Element	Größe	Beschreibung
FP	`i`	4	1. Parameter von `x` (gehört zum vorigen Frame)
$FP-4$	`k`	4	lokale Variable
$FP-8$	OldFramePtr	4	gerettetes Register $25 (Framepointer)
$FP-12$	OldRetAdr	4	gerettetes Register $31 (Return-Adresse)
$FP-16$	`Arg1`	4	1. Parameter für Prozeduraufrufe in `x`

Lösungsverfahren Siehe [4, S.35]. Ein SPL-Stackframe besteht enthält (in Reihenfolge):

- lokale Variablen der Prozedur
- gerettetes Register $25 (Framepointer)
- wenn die Prozedur weitere Prozeduraufrufe enthält:
  - gerettetes Register $31 (Return-Adresse)
  - Parameter für Prozeduraufrufe in `x`

9.2 SPL in ECO32-Assembler übersetzen

Aufgabenstellung Bestimmen Sie den ECO32-Assemblercode zur nachfolgenden SPL-Prozedur `x`. Die Prozedur `printi` erwartet einen Wertparameter vom Typ `int`. (SP=$29, FP=$25, RET=$31, verfügbare Register: $8-$15).

```
proc x (ref i:int) {
  var k: int;
  k:=i+1;
```

```
  i:=k;
  printi(i);
}
```

Quelle: [2, 2004-WS, Bl.8, Aufg.6].

Lösung

```
.export x
x:
  sub $29,$29,16          ; allocate frame
  stw $25,$29,8           ; save old frame pointer
  add $25,$29,16          ; setup new frame pointer
  stw $31,$25,-12         ; save return register
  add $8,$25,-4           ; $8 <- Adresse(k)
  add $9,$25,0            ; $9 <- Adresse(i)
  ldw $9,$9,0             ; $9 <- Wert(i) (=Adresse der ¨ubergebenen Var.)
  ldw $9,$9,0             ; $9 <- Wert(Wert(i)) (=Wert der ¨ubergebenen Var.)
  add $10,$0,1            ; $10 <- 1
  add $9,$9,$10           ; $9 <- i+1
  stw $9,$8,0             ; Wert(k) <- $9
  add $8,$25,0            ; $8 <- Adresse(i)
  ldw $8,$8,0             ; $8 <- Wert(i) (=Adresse der ¨ubergebenen Var.)
  add $9,$25,-4           ; $9 <- Adresse(k)
  ldw $9,$9,0             ; ...
  stw $9,$8,0
  add $8,$25,0
  ldw $8,$8,0
  ldw $8,$8,0
  stw $8,$29,0            ; store arg #0
  jal printi
  ldw $31,$25,-12         ; restore return register
  ldw $25,$29,8           ; restore old frame pointer
  add $29,$29,16          ; release frame
  jr $31                  ; return
```

Lösungsverfahren

1. Stackpointer auf untere Grenze des neuen Frames setzen (Frame allokieren)
2. Framepointer sichern
3. neuen Wert des Framepointers setzen (auf Stackpointer plus Framesize)
4. Return-Register sichern
5. [Befehle für den Prozedur-Rumpf]
6. Return-Register wiederherstellen
7. Framepointer wiederherstellen

8. Stackpointer auf untere Grenze des vorigen Frames setzen (Frame freigeben)
9. Rücksprung

10 Algorithmen für Klausuraufgaben

10.1 Grammatik in LL(1)-Grammatik transformieren

10.2 LL(1)-Parsertabelle bestimmen

Am Beispiel folgender Grammatik:

$$\begin{aligned} S &\rightarrow X \mid Yc \\ X &\rightarrow aX \mid bY \\ Y &\rightarrow dY \mid \epsilon \end{aligned}$$

1. Schreibe alle Regeln einzeln, d.h. ohne Alternativen:

$$\begin{aligned} S &\rightarrow X \\ S &\rightarrow Yc \\ X &\rightarrow aX \\ X &\rightarrow bY \\ Y &\rightarrow dY \\ Y &\rightarrow \epsilon \end{aligned}$$

2. Fülle folgende Tabelle aus:

   - Bestimme $FIRST\,()$ von allen vorkommenden Satzformen ($FIRST\,(e)\,, e \in T$ sind dazu Zwischenergebnisse).
   - Bestimme $FOLLOW\,()$ von allen Nichtterminalsymbolen.

   Es macht dabei Sinn, die Nichtterminalsymbole auf der linken Seite der Produktionen in umgekehrter Reihenfolge ans Ende der Tabelle zu setzen.

Symbol	$FIRST\,()$	$FOLLOW\,()$
X	$\{a, b\}$	$\{\$\}$
Y	$\{d, \epsilon\}$	$\{c, \$\}$
S	$\{a, b, c, d\}$	$\{\$\}$
Yc	$\{d, c\}$	
aX	$\{a\}$	
bY	$\{b\}$	
dY	$\{d\}$	

3. Gehe alle Produktionen $P \rightarrow Satzform$ der Reihe nach durch

   - trage $P \rightarrow Satzform$ ein für alle Elemente $e \in FIRST\,(Satzform)$ in die Zelle $[P, e]$
   - falls $\epsilon \in FIRST\,(Satzform)$: trage $P \rightarrow Satzform$ ein für alle Elemente $e \in FOLLOW\,(P)$ in die Zelle $[P, e]$

10.3 Zustände eines LALR(1)-Parsers (z.B. yacc/bison-generierte Parser)

Mögliche Optimierung: statt bei Eingabe von x (besser: $\$default$) in einen anderen Zustand zu verzweigen, der nur die Aufgabe hat $A \rightarrow \epsilon$ zu reduzieren, kann man dies auch direkt vornehmen. Denn die A erhält man in diesem Sonderfall durch reduce ohne vorangehendes shift, d.h. weil nur ein Schritt notwendig ist muss man nicht in einen zweiten Zustand verzweigen.

10.4 Abstrakter Syntaxbaum eines SPL-Programms

Hier die Syntax zur Darstellung entsprechend den Ausgabekonventionen des Referenzcompilers (übernommen aus den Funktionen `show*()` in der Datei `absyn.c` des Referenzcompilers). Argumente der Nodekonstruktoren, die auf »N« enden zeigen dass hier ein weiterer Nodekonstruktor-Aufruf stehen muss.

1. `--empty--`
2. `List(headN<5-12>,tailN<2|5-12>)` oder `List(headN<5-12>)`
3. `Nametype(name)` (beliebiger, durch einen einfachen Namen identifizierbarer Basistyp)
4. `Arraytype(size,typeN<3>)`
5. `Typedecl(name,typeN<4>)`
6. `Vardecl(name,typeN<3-4>)`
7. `Procdecl(name,paramsN<2<8>>,declsN<2<6>>,bodyN<2<9-12>>)`
8. `Param(name,typeN<3-4>,isref)`
9. `Assign(varN<16-17>,exprN<13-15>)`
10. `If(testN<13>,thenN<2<9-12>>,elseN<2<9-12>>)`
11. `While(testN<13>,bodyN<2<9-12>>)`
12. `Call(name,argsN<2<13-15>>)`
13. `Op(name,leftexprN<13-15>,rightexprN<13-15>)`
14. `Var(varN<16-17>)` (bestimmt den Wert einer Variablen)
15. `Int(value)`
16. `Simplevar(name)`
17. `Arrayvar(varN<16-17>,index)`

Hinweise:

- `List(headN,[tailN])` erhält als erstes Argument einen beliebigen Node-Konstruktor außer `List()`, als zweites (optionales) Argument den Node-Konstruktor `List()`.
- `Arraytype()` erhält als zweites Argument entweder `Nametype()` (für Basistypen) oder `Arraytype()`.
- `Simplevar()` und `Arrayvar()` geben den L-Value einer Variablen, notwendig auf der linken Seite einer Zuweisung. `Var(Simplevar())` und `Var(Arrayvar())` geben den R-Value einer Variablen.
- Ein SPL-Programm hat stets `List()` als äußersten Node-Konstruktor: es ist eine Liste von Prozedur-Deklarationen.

11 Fragen und Anmerkungen

Skript S.38 »(Die Ableitbarkeitsrelation ist demzufolge der transitive und reflexive Abschluss der direkten Ableitbarkeit.)«

Skript S.39 »$\{a^n b^n c^n \mid n \in N\}$« Was bedeutet Potenzierung mit einem Nichtterminalsymbol?

Skript S.40 Rechtsableitung und Linksableitung sind eigentlich unnötige Konzepte. Die Ableitung ist in beliebiger Reihenfolge durchführbar, sofern man die Ergebnisse der Ableitung in der ursprünglichen Reihenfolge der Nichtterminalsymbole in w und nicht in der Reihenfolge der Ableitung konkateniert.

Skript S.50 Programmiersprachen sind keine regulären Sprachen, das heißt mit regulären Ausdrücen wird eine Programmiersprache nicht vollständig beschrieben. Eine kontextfreie Grammatik ist Obermenge der regulären Grammatik, d.h. eine kontextfreie Sprache enthält Bestandteile, die mit regulären Grammatiken beschrieben werden können. Oder anders gesagt: ein Wort in einer kontextfreien Sprache ist vollständig zerlegbar in Teilwörter in regulären Sprachen. Diese Teilwörter sind die Tokens. Nur die hierarchische Struktur, in der Tokens angeordnet sein dürfen, ist mit regulären Grammatiken nicht beschreibbar.

Skript S.51 Was ist Operator-Assoziativität? Sie kann sein:

- linksassoziativ (genauer: von links nach rechts)
- rechtsassoziativ (genauer: von rechts nach links)
- nicht-assoziativ; diese Operatoren können nicht in der Form $x \circ y \circ z$ auftreten, denn ohne Assoziativität ist diese Form nicht interpretierbar

Die Assoziativität eines Operators gibt die »Ausführungsrichtung« an, in der Operationen derselben Priorität ausgeführt werden. Beispiel: Operationen mit linksassoziativen Operatoren gleicher Priorität werden »von links nach rechts« ausgeführt, so dass $w = x+y+z$ dasselbe ist wie $w = ((x + y) + z)$. Das Wort »assoziativ« bedeutet dabei »verbindend«; es ist wohl so zu verstehen dass ein linksassoziativer Operator den links von ihm stehenden Teil des Ausdrucks als »verbundenen« (zusammengesetzten, komplexen, geklammerten) Ausdruck interpretiert. Beispiel: in $x + y + z$ interpretiert der $+$-Operator vor z den links von ihm stehenden Teil als $(x + y)$, den rechts von ihm stehenden Teil als z.

Assoziativität hat nichts mit dem Assoziativitätsgesetz zu tun. Nach diesem muss die Assoziativität eines Operators, für den das Assoziativitätsgesetz gilt, gar nicht festgelegt werden, denn das Ergebnis ist unabhängig von der Auswertereihenfolge: $(a + b) + c = a + (b + c)$. Trotzdem wird in Grammatiken die Assoziativität festgelegt, einfach um einem einheitlichen System zu folgen. Dürfte der Compiler die Auswertereihenfolge bei Mehrdeutigkeiten selbst wählen statt eine Fehlermeldung auszugeben, würden manche Fehler bei der Grammatikentwicklung schwerer erkennbar.

Assoziativität funktioniert nur, wenn Operatoren gleicher Präzedenzstufe die gleiche Assoziativität zugewiesen wird!

Skript S.61 Was bedeutet »(Die Teilmengen, die Endzustände repräsentieren, müssen einelementig sein, sonst ist die Token-Syntax mehrdeutig)«.

Skript S.71 »Ein Wort $w \in \Sigma^*$unterscheidet zwei Zustände, falls A mit Eingabe w aus genau einem dieser Zustände einen Endzustand erreicht.« Was bedeutet das? Wir variieren A, indem

wir einen anderen Startzustand wählen und geben ihm dann das Eingabewort w. Aus Sicht von w gibt es zwei Äquivalenzklassen unserer Automatenvariationen: solche die w akzeptieren (nach w in einem Endzustand sind) und solche die w nicht akzeptieren. Aus Sicht von w sind damit nur Zustände unterscheidbar, die zu einem unterschiedlichen Ergebnis führen wenn man sie als Startzustand festlegt: einer zu einem Automaten, der w akzeptiert und einer zu einem Automaten, der w nicht akzeptiert.

Skript S.85 Warum sollte es für Top-Down-Analyse durch rekursiven Abstieg einen Fehler bedeuten, wenn man ein Terminalsymbol findet, das nicht gleichzeitig das erste Symbol einer Regel (d.h. das lookahead-Symbol) ist?

Skript S.110-111 »Zu ergänzen sind noch die Reduktionen für alle Zustände mit reduzierbaren Elementen:« Das Verfahren wird zwar beschrieben auf S. 107-108 (»Berechnung der Reduktionsaktionen«), jedoch ist nicht ersichtlich wie es zum hier dargestellten Ergebnis führt.

Skript S.112 Was bedeutet eigentlich »LR«?

Skript S.113 »Die LR(1)-Elemente des entsprechenden DEA-Zustands sind folgende:«

- $[S \rightarrow b.Xb,\ \$]$ Das verlangt folgende Voraussetzungen, um zu S reduzieren zu können:
  - weiteren Eingaben derart dass schließlich der Zustand $bXb.$ entsteht
  - Folgesymbol von S ist \$, äquivalent: Vorschausymbol im Zustand $bXb.$ ist \$; dies kann also noch nicht jetzt überprüft werden, sondern erst wenn $bXb.$ erreicht ist
- $[X \rightarrow .a,\ b]$ Das verlangt folgende Voraussetzungen, um nach $X \rightarrow a$ reduzieren zu können:
  - weitere Eingaben derart dass schließlich der Zustand $a.$ entsteht
  - Folgesymbol von X ist b, äquivalent: Vorschausymbol im Zustand $a.$ ist b
- $[X \rightarrow .,\ b]$ Das verlangt folgende Voraussetzungen, um nach $X \rightarrow \epsilon$ reduzieren zu können:
  - keine weiteren Eingaben, denn der Zustand $\epsilon.$ ist bereits erreicht
  - Folgesymbol von X ist b, äquivalent: Vorschausymbol um Zustand $\epsilon.$ ist b; dies kann jetzt überprüft werden, denn der Zustand $\epsilon.$ ist bereits erreicht

Literatur

[1] Prof. Dr. Michael Jäger: »Compilerbau«. Das offizielle Skript zu dieser Veranstaltung. Quelle: Homepage von Prof. Jäger http://homepages.fh-giessen.de/~hg52/, alternativ http://m-jaeger.de/. Das Skript enthält alles, was für die Klausur notwendig ist; dazu alle für das Praktikum relevante Dokumentation von flex und bison; dazu Literaturempfehlungen; jedoch nicht SPL-spezifischen Stoff für das Praktikum, vgl. dazu [4].

[2] Prof. Dr. Michael Jäger: Übungsaufgaben, Hausübungen und Bonusaufgaben zur Veranstaltung Compilerbau. Quelle: http://homepages.fh-giessen.de/~hg52, alternativ: http://m-jaeger.de.

[3] Prof. Dr. Michael Jäger: Klausuren zu Compilerbau. Ein Archiv alter Klausuren zu dieser Veranstaltung. Quelle: http://homepages.fh-giessen.de/~hg52.

[4] Boris Budweg: Mitschrift zu Compilerbau bei Prof. Dr. Hellwig Geisse; Wintersemester 2003/2004; FH Gießen-Friedberg. Veröffentlicht auch auf http://homepages.fh-giessen.de/~hg52.

[5] Webelsiep: Quellcodes zu Compilerbau. http://www.webelsiep.de/downloads/quellcode/compilerbau.zip, referenziert unter http://www.webelsiep.de/default.php?main=studium, Bereich Quellcodes.

[6] Tim Pommerening: Lernkartei zu Compilerbau. http://homepages.fh-giessen.de/~hg12061/studium/COBA_Lernkartei.rar, referenziert auf http://homepages.fh-giessen.de/~hg12061/, Bereich Studium.

[7] Aho, Setho, Ullman: »Compilerbau«. In zwei Bänden. Das »Drachenbuch«. Das Standard-Lehrbuch, das alle Methoden und Verfahren des Compilerbaus erklärt und diskutiert.

[8] Wirth: Implementierung einer Teilmenge von Oberon1. Es gibt eine Unmenge von solchen Büchern, die den Bau eines Compilers an einem Beispiel beschreiben statt verschiedenste Verfahren gegenüberzustellen. Diese Bücher haben wesentlich weniger Umfang als Lehrbücher wie [7].