5. Syntaxanalyse und der Parser-Generator yacc

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex (2 Termine)
5. Syntaxanalyse und der Parser-Generator yacc (3 T.)
6. Syntaxgesteuerte Übersetzung
7. Kontextanalyse
8. Übersetzungssteuerung mit make

5. Syntaxanalyse und der Parser-Generator yacc

5.1 Einleitung
5.2 Kontextfreie Grammatiken
5.3 Grundlagen von yacc
5.4 Absteigende Analyse
5.5 Aufsteigende Analyse

Lexikalische Analyse

Gruppierung der Eingabezeichen in Lexeme

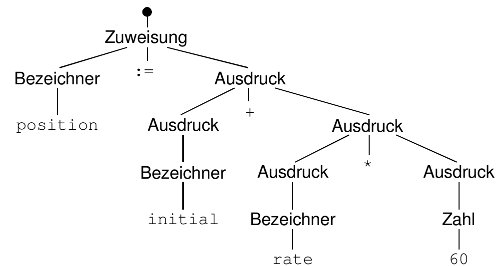
- Leerzeichen werden entfernt

Zuordnung Lexem Symbol

- Symbole (Token): Bezeichner, :=, +, *, Zahl, ...
- Symbol hat z.T. Wert als Attribut

Syntaxanalyse (Parsing)

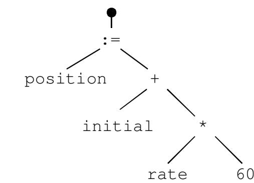
Parse-Baum:



hierarchische Gruppierung der Symbole

ñ mit Hilfe einer kontextfreien Grammatik

Syntaxbaum



ist komprimierte Darstellung des Parse-Baums

Ergebnis der Syntaxanalyse

1. Parse-Baum / Syntaxbaum
   ñ oder Fehlermeldungen
2. syntaxgesteuerte Übersetzung
   ñ Baum nicht explizit generiert

5. Syntaxanalyse und der Parser-Generator yacc

5.1 Einleitung

5.2 Kontextfreie Grammatiken

5.3 Grundlagen von yacc

5.4 Absteigende Analyse

5.5 Aufsteigende Analyse

Definition: Kontextfreie Grammatik

ein Tupel G = (V, N, R, *s*) mit

- Vokabular/Alphabet V
- Nichtterminale N ⊂ V
    - • geschrieben *a*, *b*, *c*, ...
  - induzieren Terminale T = V \ N
    - • geschrieben A, B, C, ...
- Regeln R ⊂ N x V*
  - geschrieben *a*   α, falls α ∈ V*
  - geschrieben *a*   w, falls α ∈ T*
- Startsymbol *s* ∈ N

Kontextfreie Grammatik einfacher Ausdrücke

- G = (V, N, R, *s*) mit
  - V = {*e*, ID, +, , (,)}
  - N = {*e*}   und T = {ID, +, , (,)}
  - R = {*e*   ID, *e*   *e* + *e*, *e*   *e*   *e*, *e*   (*e*)}
  - *s* = *e*

Kontextfreie Sprache

- kann mit einer kontextfreien Grammatik G beschrieben werden: L(G) ⊆ V*

Spezifikation kontextfreier Grammatiken mit BNF

- Backus-Naur-Form:
  Notation für kontextfreie Grammatiken
- Regeln geschrieben als
  a ::= ...
- Terminale in Anführungszeichen: '+'
- Verkettung durch Hintereinanderschreiben

Einfache Ausdrücke in BNF

e ::= 'ID'

e ::= *e* '+' *e*

e ::= *e* ' ' *e*

e ::= '(' *e* ')'

Erweiterung zu EBNF

- zusätzlich Abkürzungen:
  - Alternative: Regeln mit gleichem Nichtterminal links zusammengefaßt als
    a ::= ... | ... | ...
  - [...]: Option, null- oder einmal
  - {...}: Wiederholung, null- bis beliebig mal
  - (...): Klammerung

Einfache Ausdrücke in EBNF

e ::= 'ID' | *e* '+' *e* | *e* ' ' *e* | '(' *e* ')'

Ableitung: Definition

- Ableitung: Folge von Ableitungsschritten.
  - Beispiel:
    e ⇒ *e* + *e* ⇒ ID + *e* ⇒ ID + ID
- Ableitungsschritt ⇒ ': Ersetzung eines beliebigen Nichtterminals in einem Wort ∈ V*
  - wenn $n \rightarrow \gamma \in R$
  - und $= \alpha n \beta$
  - dann $' = \alpha\gamma\beta$

Wiederholtes Ableiten

- 0... mal: ⇒*
  - Beispiele:
    - e ⇒* ID + *e*
    - e ⇒* e
- 1... mal: ⇒⁺

Sprache einer kontextfreien Grammatik

- Sprache eines Nichtterminals
  L(*n*) = { w T* | *n* ⇒* w }
- Sprache der Grammatik
  L(G) = L(*s*)

Eigenschaften von Ableitungen

- Linksableitung
  - das jeweils am weitesten links stehende Nichtterminal wird ersetzt
- Rechtsableitung
  - (analog)
- Satz:
  - Für jedes Wort einer kontextfreien Grammatik gibt es auch eine Linksableitung (bzw. Rechtsableitung).

Eigenschaften von Ableitungen (2)

- vollständige Ableitung
  - = *s*
- terminale Ableitung
  - ' T*

Ableitungsbaum (Parse-Baum): Definition

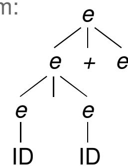
- ein Baum
- Knoten mit Elementen aus V ∪ {ε} markiert
- innere Knoten mit Elementen aus N markiert
- Nachfolger eines Knotens *n* markiert mit
  - $\alpha_1, ..., \alpha_n,$ falls *n* $\alpha_1 ... \alpha_n \in R$
  - ε, falls *n* $\varepsilon \in R$

Eigenschaften von Ableitungsbäumen

- vollständiger Ableitungsbaum
  - Wurzel mit *s* markiert
- terminaler Ableitungsbaum
  - alle Blätter mit Elementen aus T* ∪ {ε} markiert

Eigenschaften von Ableitungsbäumen (2)

- Beispiel
  - eine Ableitung:
    $e \Rightarrow e + e \Rightarrow e \quad e + e \Rightarrow ID \quad e + e \Rightarrow ID \quad ID + e$
  - ihr Ableitungsbaum:



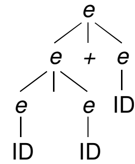
- Satz:
  - Für jeden vollständigen terminalen Ableitungsbaum gibt es *genau eine* vollständige terminale Linksableitung (bzw. Rechtsableitung)

Mehrdeutigkeit: Definition

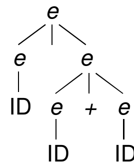
- Eine Grammatik ist mehrdeutig, wenn mindestens ein Wort ihrer Sprache mehrere Linksableitungen hat.
  - bzw. Rechtsableitungen bzw. Ableitungsbäume

Mehrdeutigkeit: Beispiel

- erste Linksableitung
  - ñ e ⇒ *e* + *e*
    ⇒ *e* *e* + *e*
    ⇒ ID *e* + *e*
    ⇒ ID ID + *e*
    ⇒ ID ID + ID



- zweite Linksableitung
  - ñ e ⇒ *e* *e*
    ⇒ ID *e*
    ⇒ ID *e* + *e*
    ⇒ ID ID + *e*
    ⇒ ID ID + ID



Dangling-Else-Problem

- Mehrdeutigkeit in Algol-60:
  - ñ *statement* ::= 'if' *expression* 'then' *statement*
    | 'if' *expression* 'then' *statement* 'else' *statement*
    | *other*
  - ñ `if` E_1 `then if` E_2 `then` S_1 `else` S_2
- Behebung:
  - ñ *statement* ::= *unbalanced_st* | *balanced_st*
  - ñ *unbalanced_st* ::= 'if' *expression* 'then' *statement*
    | 'if' *expression* 'then' *balanced_st* 'else' *unbalanced_st*
  - ñ balanced_st ::=
    'if' *expression* 'then' *balanced_st* 'else' *balanced_st*
    | *other*

5. Syntaxanalyse und der Parser-Generator yacc

5.1 Einleitung
5.2 Kontextfreie Grammatiken
5.3 Grundlagen von yacc
5.4 Absteigende Analyse
5.5 Aufsteigende Analyse

Aufbau einer yacc-Datei

Definitionen
`%%`
Regeln
`%%`
Unterprogramme } optional

Demo

Der kadonische Leuchtturm

Die Einwohner der Insel Kadonien lieben die Abwechslung, und sie streichen daher ihren Leuchtturm alle zwei Wochen neu an. Dies ist natürlich sehr verwirrend für die Seefahrer. Aber wir können den Seefahrern helfen: Jedes Mal geht eine zufällig sich ergebende Gruppe von kadonischen Frauen und Männern los, um den Turn neu zu streichen. Jeder Helfer streicht dabei drei Meter zusammenhängende Turmhöhe, und zwar in drei Ringen. Ein Mann streicht stets rot-weiß-rot, eine Frau stets weiß-rot-weiß. Der Turm ist zwölf Meter hoch. Die Leuchttürme auf den Nachbarinseln sind zwar auch immer in ein Meter breiten Ringen bemalt, und sie sind zum Teil auch zwölf Meter hoch. Aber die Nachbarn mögen die kadonische Leuchtturmmode nicht und haben streng darauf geachtet, niemals das kadonische Schema zu verwenden. Wir schreiben daher ein Programm, das die Insel Kadonien sicher anhand seines Leuchtturms erkennt. Das Programm soll eine Zeile vom benutzenden Kapitän lesen und dann ausgeben, ob er vor Kadonien liegt oder nicht.

EBNF für den kadonischen Leuchtturm

kadonischerturm ::= *helfer helfer helfer helfer*
helfer ::= *mann*
| *frau*
mann ::= ROT WEISS ROT
frau ::= WEISS ROT WEISS

ROT ::= `'rot'`
WEISS ::= `'weiss'`

Der kadonische Leuchtturm: Lösung

leuchtturm.y:

```
%{
#include <stdio.h>
void yyerror(char *);
%}
%token ROT WEISS SCHREIBFEHLER
%%
kadonischerturm:  helfer helfer helfer helfer
                ;
helfer:           mann
                | frau
                ;
mann:             ROT WEISS ROT
                ;
frau:             WEISS ROT WEISS
                ;
%%
void yyerror(char *msg) {
}
int main() {
    printf("Wie sieht der Leuchtturm aus, Sir? ");
    if(yyparse() == 0)
        printf("Wir liegen vor der Insel Kadonien, Sir!\n");
    else
        printf("Wir liegen *nicht* vor der Insel Kadonien, Sir!\n");
    return 0;
}
```

Der kadonische Leuchtturm: Lösung (2)

Anmerkungen:

- `%token`: welche Symbole der Lexer liefern kann
- Startregel: erste Regel
- wir müssen ein `main()` schreiben
  - `yyparse()` ruft den Parser auf
    - Rückgabe == 0: Eingabe paßt zur Grammatik
- wir müssen eine Fehlermeldefunktion `yyerror()` schreiben
  - diese tut nichts, weil wir den Fehler in `main()` melden

Der kadonische Leuchtturm: Lösung (3)

leuchtturm.l:

```
%{
#include "leuchtturm.tab.h"
%}
%option noyywrap
%option nodefault
%%
"rot"           { return(ROT); }
"weiss"         { return(WEISS); }
\n              { return 0; /* Nur eine Zeile lesen. */ }
[ \t]           /* Ignoriere sonstigen Whitespace. */
.               { return(SCHREIBFEHLER); }
```

Der kadonische Leuchtturm: Lösung (4)

Anmerkungen:

- die Datei `leuchtturm.tab.h` mit den Rückgabe-Konstanten wird von bison aus der `%token`-Zeile generiert

Ein allgemeines Makefile für bison mit flex

Makefile:

```
%.c: %.y # loesche alte implizite Regel
%.c: %.l # loesche alte implizite Regel
%.tab.c %.tab.h: %.y
        bison --defines $<
%.c: %.l %.tab.h
        flex -t $< > $@
%: %.tab.o %.o
        cc -o $@ $^
        rm -f $@.tab.h
```

Aufruf z.B.:
`make leuchtturm`

EBNF-Erweiterungen und yacc

erlaubt:

- Alternative: Regeln mit gleichem Nichtterminal links zusammengefaßt als
  a ::= ... | ... | ...

nicht erlaubt:

- [...]: Option, null- oder einmal
- {...}: Wiederholung, null- bis beliebig mal
- (...): Klammerung

Ausweg: normale BNF-Darstellung

Übung: Klammerausdrücke

ka ::= ε
    | *ka ka_einfach*
ka_einfach ::= BUCHSTABE
    | '(' *ka* ')'
    | '[' *ka* ']'
    | '{' *ka* '}'
    | '<' *ka* '>'

schreibe einen Parser (mit Scanner), der korrekte Klammerausdrücke von stdin liest und erkennt

- Definition von BUCHSTABE: Scanner
- Tipp: Alle 255 ASCII-Zeichen > 0 sind implizit bereits als Token definiert. Lex darf daher die Klammern direkt als Token zurückgeben.

Aufruf- und Datei-Optionen von bison

Datei	Aufruf	Aufruf lang	Bedeutung
`%defines`	`-d`	`--defines`	Schreibe die `.tab.h`-Datei für lex.
`%verbose`	`-v`	`--verbose`	Schreibe eine `.output`-Datei mit allen Parser-Zuständen und allen Regel-Konflikten.
`%debug`	`-t`	`--debug`	Definiere das Makro YYDEBUG, so daß Debugging möglich wird.
`%name-prefix="XYZ"`	`-p XYZ`	`--name-prefix=XYZ`	Ersetze in allen Variablen- und Funktionsnamen das Präfix `yy` durch `XYZ`.
	`-h`	`--help`	Gib die möglichen Aufrufoptionen aus.
...	...	...	...

5. Syntaxanalyse und der Parser-Generator yacc

5.1 Einleitung
5.2 Kontextfreie Grammatiken
5.3 Grundlagen von yacc
5.4 Absteigende Analyse
5.5 Aufsteigende Analyse

Absteigende Analyse versus aufsteigende Analyse

Absteigende Analyse (top-down-Analyse)

- Knoten des Ableitungsbaums werden von der Wurzel her konstruiert
- leichter von Hand zu programmieren

  „Praktische Informatik 2“

Aufsteigende Analyse (bottom-up-Analyse)

- Knoten des Ableitungsbaums werden von den Blättern her konstruiert
- größere Klasse von Grammatiken

  seltener Grammatiktransformation von Hand nötig
- yacc

Absteigende Analyse

Prinzip:

- rekursiver Abstieg
  - jedes Nichtterminal eine Funktion

Backtracking vermeiden:

- Transformation in SLL(1)-Grammatik
  - „strong leftmost left to right with 1 symbol lookahead“
  - SLL(1)-Bedingung:
    wenn für ein Nichtterminal mehrere Regeln:
    jede muß mit einem anderen Terminal beginnen
  - für „prädiktive“ Parser

Demo

Der kadonische Leuchtturm von Hand in C programmiert

leuchtturm-manuell.c

5. Syntaxanalyse und der Parser-Generator yacc

5.1 Einleitung
5.2 Kontextfreie Grammatiken
5.3 Grundlagen von yacc
5.4 Absteigende Analyse
5.5 Aufsteigende Analyse