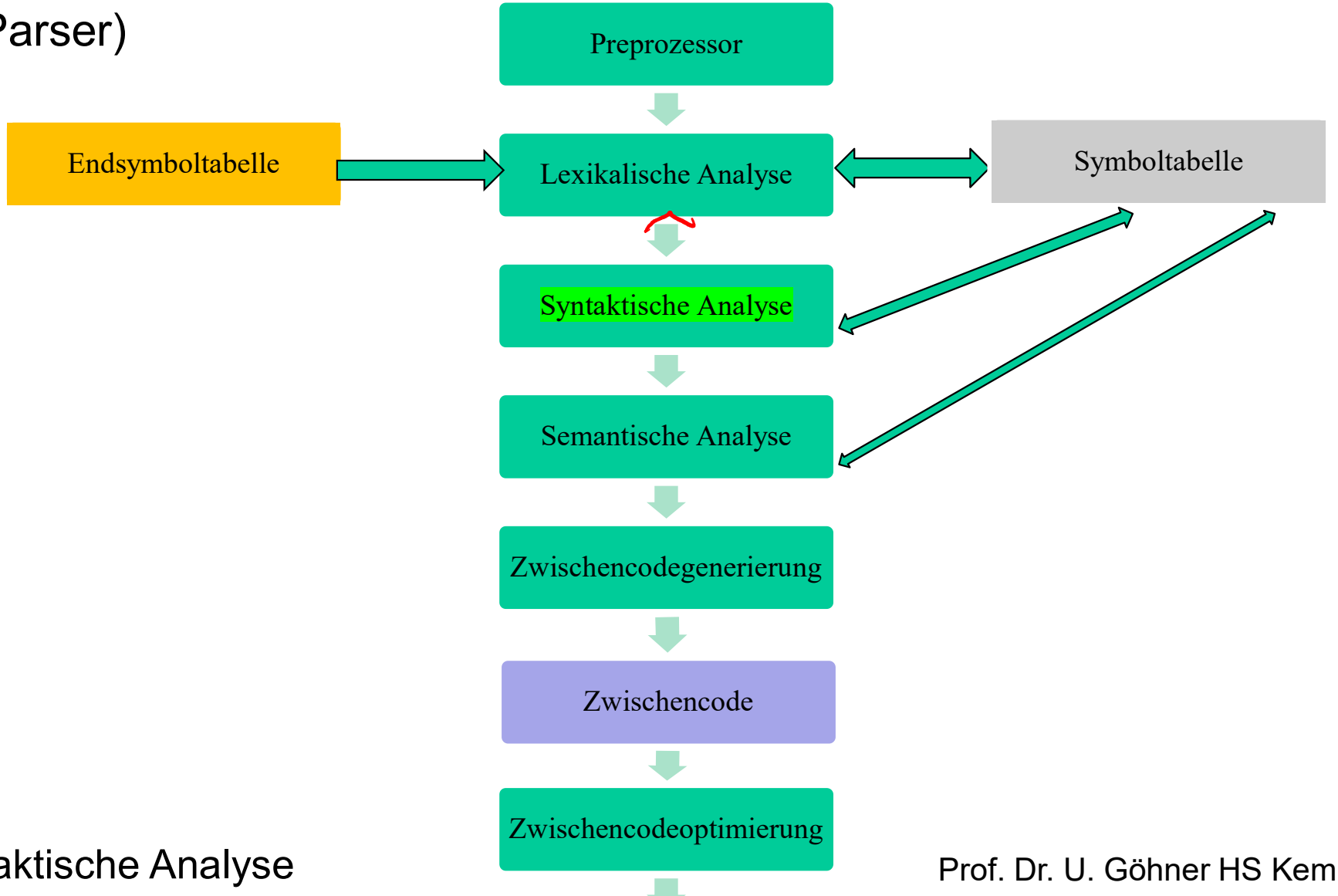
Syntaxanalyse

Syntaktische Analyse
(Parser)



Syntaxanalyse

- Syntaxanalyse prüft, ob Eingabe bezüglich einer zugrundeliegenden Grammatik syntaktisch korrekt sind.
- Ein Eingabe ist syntaktisch korrekt, wenn ein Syntaxbaum gemäß den Regeln der Grammatik aufgebaut werden kann
- Während der Syntaxanalyse werden weitere Verarbeitungsschritte vorgenommen
  - semantische Prüfung
  - Aufbau Symboltabelle + Abstrakter Syntaxbaum
  - Berechnung der Attribute
  - Zwischencode-Generierung.

Syntaxanalyse

Strategien zur Berechnung des Ableitungsbaums

- Ziel der Syntaxanalyse ist die Konstruktion des Ableitungsbaums zu einer gegebenen 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.

Syntaxanalyse

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.

Syntaxanalyse

Präzedenzen und Assoziativitätsregeln in Programmiersprachen

- Bei ungeklammerten Ausdrücken ist die eindeutige Zuordnung der Operanden zu den Operatoren durch Präzedenz- und Assoziativitätsregeln definiert.
- Präzedenz
  - Unter den Operatoren wird eine Präzedenzfolge festgelegt
  - Operator höherer Präzedenz bindet stärker.
- Assoziativität
  - bei Gruppen von Operatoren gleicher Präzedenz wird innerhalb jeder Gruppe linksassoziativ: a* b* c = (a* b)* c
  - bzw. rechtassoziativ a* b* c = a* (b* c) zugeordnet.
- Operator * heißt nicht-assoziativ, wenn ein Ausdruck der Form a*b*c nicht erlaubt ist.

Syntaxanalyse

Präzedenz und Assoziativität in der Grammatik

- Mehrdeutigkeiten bei ungeklammerten Ausdrücken mit mehreren Operatoren lassen sich durch Berücksichtigung von Präzedenz- und Assoziativitätsregeln in der Grammatik auflösen.
- bei Parser-Generatoren (z.B. yacc) gibt es oft die Möglichkeit, zu einer mehrdeutigen Grammatik die Präzedenzregeln und Assoziativitäten der Operatoren direkt anzugeben. Die Grammatik kann dadurch kompakter gehalten werden:

Yacc/Bison: Links-/Rechtsassoziative Operatoren werden in der token-Bezeichnung mit %left bzw. %right gekennzeichnet

Syntaxanalyse

Top-Down-Verfahren:

Ableitung wird ausgehend vom Startsymbol berechnet, d.h. Ableitungsbaum wird von der Wurzel ausgehend konstruiert.

Nichtdeterministische Top-Down-Analyse:

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

Ausgabe: Syntaxbaum zu w

Syntaxanalyse

Top-down-Parser

Vorgehensweise:

Parsebaum entsteht von der Wurzel zu den Blättern

Auch "LL-Parser":

"L" => Tokenstrom wird links->rechts verarbeitet

”L" => Parser liefert Linksableitung der Eingabe

Syntaxanalyse

Top-down-Parser

Ablauf nichtdeterministischer LL-Parser

1. Parsebaum mit erstem Knoten starten -> Startsymbol
   (spätere Wurzel des Parsebaumes)
2. Wiederholen bis Parsebaum komplett:
   a. ersten linken, nichtterminalen Knoten auswählen
   b. Regel auswählen (nichtdeterministisch)
   c. Unterknoten (rechte Regelseite) anfügen
3. Parsebaum fertig

Problem: Hoher Aufwand, falls Regelauswahl nicht eindeutig, da alle Möglichkeiten durchprobiert werden müssen

Syntaxanalyse

Top-Down-Verfahren

Nichtdeterministische Top-Down-Analyse

Algorithmus:

Beginne die Konstruktion des Ableitungsbaums mit der Wurzel=S
while (mit Nonterminalsymbol markierter Blattknoten vorhanden)
wähle mit einem Nonterminalsymbol A markierten Blattknoten als aktuellen Knoten k;
Wähle zu A eine Regel A ->a_1 ... a_n
Erzeuge für jedes Symbol der rechten Seite einen Baumknoten als Nachfolger von k

if (Blattmarkierungen stimmen mit w überein) then
konstruierter Baum ist Syntaxbaum zu w;
die in der Schleife ausgewählten Regeln bilden Ableitung für w

Syntaxanalyse

Top-Down-Verfahren

deterministische Top-Down-Analyse

Eindeutige Auswahl der nächsten Regel durch **Vorausschau** der nächsten k Eingabezeichen (“lookahead”)

LL(k)-Grammatik:

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

Der Einfachkeit halber werden im weiteren Verlauf nur LL(1)-Grammatiken behandelt.

Syntaxanalyse

Top-Down-Analyse durch rekursiven Abstieg

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

Parser-Aufbau:

Zu jedem Nonterminalsymbol A $\in$ N wird eine Funktion PARSEA konstruiert, das alle aus A ableitbaren Wörter analysiert

PARSEA wählt anhand des lookahead-Symbols die wegen der LL(1)-Eigenschaft der Grammatik eindeutig bestimmte Ableitungsregel

A -> $a_1 ... a_n$

rechte Seite $a_1 ... a_n$ wird von links nach rechts verarbeitet:

$a_i \in T$ => if (a_i = lookahead-Symbol) then
bestimme nächstes lookahead-Symbol
else Syntaxfehler: „a_i statt lookahead erwartet“

$a_i \in N$ => PARSEa_i aufrufen

Syntaxanalyse

Beispiel für Top-Down-Analyse:

S->ABC

A->aa

A->b

B->bbB

B->ε

C->c

Syntaxanalyse

Beispiel für Top-Down-Analyse:

- Konstruktion Scanner und Hilfsfunktionen
- Parser für Top-Down-Analyse

Definition:

Falls die richtige Ableitungsregel durch Nachschauen in einer Tabelle realisiert ist, handelt es sich um einen **tabellengesteuerten Parser**.

Wie konstruiert man diese Tabelle?

Syntaxanalyse

Konstruktion der LL(1)-Parsertabelle

Zur Grammatik G=(T, N, P, S) wird die LL(1)-Parsertabelle bestimmt

Parser muss für Nonterminalsymbol X eine Ableitungsregel bestimmen.

Auswahlkriterium: Vorausschautoken= nächstes Terminalsymbol

- Falls Ableitungsregel X ->w nicht leeres Wort w erzeugt, ist Vorausschautoken das erste Symbol von w.
- Ableitungsregel passt für alle Terminalsymbole mit denen w beginnen kann (FIRST(w)).

Syntaxanalyse

Konstruktion der LL(1)-Parsertabelle

- Falls mit Ableitungsregel X -> w das leere Wort erzeugt wird, gehört das Vorausschausymbol zu einem auf X folgenden anderen Bestandteil der Eingabe.
- Folgesymbol von X ist ein Terminalsymbol, dass in der Eingabe direkt hinter X steht.

Ableitungsregel passt in diesem Fall für alle möglichen Folgesymbole (Follow(X)).

Wie bestimmt man First (w) und Follow(X)?

Syntaxanalyse

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.

1. Falls $X \in T$, FIRST(X)=X
2. Falls $(X \rightarrow \varepsilon) \in P$, übernehme ε in FIRST(X)
3. Für jede Regel $(X \rightarrow Y_1 \ldots Y_k) \in P$:

- Falls $\varepsilon \in FIRST(Y_1) \wedge \ldots \wedge \varepsilon \in FIRST(Y_{i-1})$ mit $1 < i <= k$, übernehme alle Terminalzeichen aus $FIRST(Y_i)$ in FIRST(X)
- Falls $\varepsilon \in FIRST(Y_1) \wedge \ldots \wedge \varepsilon \in FIRST(Y_k)$, übernehme ε in FIRST(X)

Syntaxanalyse

Für die Bestimmung der FIRST-Menge eines Wortes w gibt es drei Fälle:

1. w = ε, dann ist FIRST(w) = ε
2. w beginnt mit einem Terminalsymbol x: w = xv, x $\in$ T.
   Dann ist FIRST(w) = FIRST(x) = $\{x\}$
3. 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:
   a) Wenn nicht, beginnt Xv mit einem Symbol aus FIRST(X) und v wird nicht weiter verfolgt. Alle Terminalsymbole aus FIRST(X)\{ε} werden in FIRST(w) übernommen.
   b) Falls ε aus X ableitbar ist, lässt sich aus w über Xv auch v ableiten, also muss auch FIRST(v) bestimmt werden und alle Terminalsymbole aus FIRST(X)\{ε} werden in FIRST(w) übernommen. Falls w=Xv komplett nach ε abgebildet werden kann, übernehme ε in First(w)

Syntaxanalyse

Bestimmung der Folgesymbolmengen

- Bestimme zu jedem Nonterminal X $\in$ N 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 jedes Auftreten von X auf einer rechten Regelseite betrachtet werden um herauszufinden welche Terminalsymbole gemäss der Regel dahinter stehen können.

Syntaxanalyse

Bestimmung der Folgesymbolmengen

- Bestimme zu jedem Nonterminal X $\in$ N 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 jedes Auftreten von X auf einer rechten Regelseite betrachtet werden um herauszufinden welche Terminalsymbole gemäss der Regel dahinter stehen können.

Syntaxanalyse

Algorithmus zur Bestimmung der Follow-Mengen

1. Übernehme das Eingabeende-Token $ in FOLLOW(S)
2. Für jedes Auftreten eines Nonterminalsymbols X in einer Regel (A -> uXw) $\in$ P übernehme alle Terminalzeichen aus FIRST(w)\{ε} in FOLLOW(X).
3. Für jede Regel (A -> uB) $\in$ P sowie jede Regel (A -> uBv) $\in$ P mit $\varepsilon \in$ FIRST(v), übernehme alle Terminalzeichen aus FOLLOW(A) in FOLLOW(B).

Syntaxanalyse

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.

Algorithmus zur Konstruktion der Parsertabelle

Für jede Ableitungsregel A -> α :

1. Für jedes Terminalsymbol a $\in$ FIRST(α) übernehme
   die Regel A -> α in TAB[A, a]
2. Falls $\varepsilon \in$ FIRST(α), übernehme
   Regel A -> α in TAB[A, b] für jedes b $\in$ FOLLOW(A)
   Falls $\varepsilon \in$ FIRST(α) und \$ $\in$ FOLLOW(A), übernehme
   Regel A -> α in TAB[A, \$].

Syntaxanalyse

Definition:

Eine kontextfreie Grammatik G heißt **LL(1)-Grammatik**, wenn gilt:

a) Gibt es zwei Produktionen N $\rightarrow$s1 und N $\rightarrow$ s2, dann ist first(s1)$\cap$ first(s2) = $\varnothing$.

b) Gibt es ein Nichtterminal N, mit N $\rightarrow$ ε , dann gilt first(N) $\cap$ follow(N) = $\varnothing$.

L L (1)

Linksableitung, Eingabe von Links gelesen, 1 Zeichen vorrauslesen um zu entscheiden, welche Produktion zu verwenden ist

Syntaxanalyse

Bestimmung der Predict Mengen

Definition:

Ist $N \rightarrow s$, $s \in A^*$ eine Produktion, dann heisst die Entscheidungsmenge:

predict($N \rightarrow s$) = first (s follow(N))

predict($N \rightarrow s$) ist die Menge der Terminalsymbole, mit denen eine Satzform bei Verwenden der Produktion $N \rightarrow s$ beginnen kann.

Syntaxanalyse

Bestimmung der Predict Mengen

Satz:

Eine kontextfreie Grammatik ist genau dann eine LL(1)-Grammatik, wenn für alle Produktionen $N \rightarrow s1|s2 \ldots |sn$ gilt:

$Predict(N \rightarrow s1) \cap Predict(N \rightarrow s2) = \emptyset$, $Predict(N \rightarrow s1) \cap Predict(N \rightarrow s3) = \emptyset$,...

Syntaxanalyse

LL(1)-Grammatik

Satz:

Eine kontextfreie Grammatik G ist genau dann eine LL(1) – Grammatik, wenn gilt:

G erlaubt eine Sackgassenfreie Top–Down–Analyse von links nach rechts, d.h. wenn es alternative Produktionen N $\rightarrow$ s1 | s2 . . . gibt, dann kann mit Kenntnis des nächsten Symbols die richtige Ableitung gewählt werden.

Syntaxanalyse

Algorithmus des rekursiven Abstieg bei LL(1)-Grammatiken:

Für jedes Nichtterminal A mit den Produktionen A $\rightarrow$ s1|s2|...|sn erzeugt man eine Funktion A(), die in Abhängigkeit vom zuletzt gelesenen Token die richtige Produktion auswählt und das Verarbeiten der rechten Seite übernimmt.

Syntaxanalyse

Problem der Rekursivität:

Produktionen der Form A $\rightarrow$ As

führen zu Funktionen der Form

```
A( )
{
      A( ); /* Rekursion -> Endlosschleife! */
      match(s);
}
```

Syntaxanalyse

Umformung in äquivalente LL(1) – Grammatiken:

Linksfaktorisierung:

Beseitigung gleicher Anfangssymbole:

$A \rightarrow s\ w1 \mid s\ w2$

Wird ersetzt durch:

$A \rightarrow s\ B$ und $B \rightarrow w1 \mid w2$

Beseitigung direkter **Linksrekursionen**

$A \rightarrow A\ \alpha \mid \beta$ (erzeugt: $\beta\alpha$ .....α)

wird ersetzt durch:

$A \rightarrow \beta\ B$

$B \rightarrow \alpha\ B \mid \varepsilon$

Syntaxanalyse

Umformung in äquivalente LL(1) – Grammatiken:

Beseitigung indirekter Linksrekursionen

$A \rightarrow Bb \mid a$

$B \rightarrow Ac \mid d$

Einsetzen von B in die erste Zeile ergibt direkte Linksrekursion:

$A \rightarrow Acb \mid db \mid a$

Syntaxanalyse

Nachweis der LL(1) – Eigenschaft durch Nachprüfen der Definition:

Eine kontextfreie Grammatik G heißt **LL(1)-Grammatik,** wenn gilt:

a) Gibt es zwei Produktionen N $\rightarrow$s1 und N $\rightarrow$ s2,

dann ist first(s1)$\cap$ first(s2) = $\varnothing$.

b) Gibt es ein Nichtterminal N mit N $\rightarrow$ ε ,

dann gilt first(N) $\cap$ follow(N) = $\varnothing$.

Syntaxanalyse

Bottom – Up – Verfahren:

Parsebaum entsteht von den Blättern zur Wurzel, erzeugt eine Rechtsableitung des Satzes.

Auch "LR-Parser"

"L " => Tokenstrom wird links->rechts verarbeitet

" R" => Parser liefert Rechtsableitung der Eingabe

Syntaxanalyse

Kellerautomat

Definition: Ein Kellerautomat ist ein 7-Tupel KA = (Z , A , K , f_k, z_0 , k_0, F) ,

Z - Menge der Zustände des Automaten

A - Eingabealphabet

K - Kelleralphabet K: Menge der Symbole, die auf dem (unbeschränkten) Keller (=Stack) abgelegt werden können.

f_k - Schaltfunktion f_k: $Z \times (A \cup \varepsilon) \times K \rightarrow Z \times K$

$z_0 \in Z$ - Anfangszustand des Automaten

$k_0 \in$ Kellerstartsymbol: Es befindet sich beim Start im Keller

$F \subset Z$ Menge der Endzustände.

Syntaxanalyse

Kellerautomat

Satz:

Zu jeder kontextfreien Grammatik G (Chomsky Typ 2) gibt es einen Kellerautomaten KA und umgekehrt, so daß L(G)=L(KA) gilt.

Beweis: J. Hromkovic, Theoretische Informatik, Kapitel 10.4

Syntaxanalyse

Bottom – Up – Verfahren:

Bearbeitung findet mit Hilfe eines Kellers statt.

mögliche Aktionen:

- Shift: Einlesen in den Keller
- Reduce: Reduktion nach Produktion n (die obersten Kellersymbole werden vom Keller entfernt und durch die linke Seite einer Produktion ersetzt)
- Halt/Accept: Startsymbol X liegt allein im Keller und der Eingaberest ist leer: Ende der Analyse.

Syntaxanalyse

Bottom – Up – Verfahren:

Mögliche Konfliktsituationen:

1. reduce/reduce-Konflikt:
   - Der aktuelle Stack lässt unterschiedliche Reduktionen zu, welche soll ausgewählt werden?
2. shift/reduce-Konflikt
   - Der aktuelle Stack lässt Reduktion zu. Soll reduziert werden oder sollen weitere Eingabesymbole gelesen werden (shift), um ein anderes "handle" zu erhalten ?

Die Lösung der Konfliktsituationen ist vom Stack-Inhalt und von der Vorausschau abhängig. 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.

Syntaxanalyse

Bottom – Up – Verfahren:

Zustandsautomat

Analyse des Kellers auf mögliche Reduktionen ist aufwändig.

Alternativ: Interner Zustand des Parsens merken und darauf aufbauend weiterarbeiten.

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

Konsequenz: Anstelle Analyse des Stacks (ineffzient !), Information in Form eines DEA-Zustands berechnen.

Aktueller Zustand wird auf dem Stack abgelegt, enthält notwendige Information über den darunterliegenden Stack-Inhalt.

Damit kann in Abhängigkeit des nächsten Zeichens entschieden werden, ob Shift-Befehl oder Reduce Befehl durchzuführen ist.

Syntaxanalyse

Bottom – Up – Verfahren:

Parsertabelle

Parser (=Kellerautomat) wechselt zwischen verschiedenen inneren Zuständen.

Zustände werden bestimmt durch die bereits gelesene Folge von Tokens, die den Kellerinhalt bestimmen, und den erwarteten Token, die als nächste zu lesen sind.

Zustände beschreiben das Ergebnis der bisherigen Analyse vollständig und drücken damit auch eine „Erwartungshaltung“ für das nächste Token aus. Die Schaltfunktion des Kellerautomaten wird durch die Parsertabelle beschrieben

Syntaxanalyse

Bottom – Up – Verfahren:

- Keller: Besteht aus Symbolkeller und Zustandskeller:
  - Symbolkeller enthält alle Symbole, die eingelesen wurden und zur Verarbeitung anstehen
  - Zustandskeller enthält den Zustand des jeweiligen Symbols, d.h. welchen Zustand im DEA hat das entsprechende Zeichen im Symbolkeller.
- Im Keller werden alle noch nicht reduzierten Eingabezeichen bzw. bereits reduzierten Eingabezeichen angesammelt.
- Aktionstabelle: Sie gibt zu jedem Eintrag (aktueller Zustand, gelesenes Token) an, ob shift– oder reduce–Aktion durchgeführt wird. Bei syntaktisch nicht erlaubten Kombinationen steht ein Fehlereintrag

Syntaxanalyse

Bottom – Up – Verfahren:

- Shift–Eintrag: „sk“: Einlesen des nächsten Zeichens aus der Eingabe in den Keller und Ablage des Zustandes k im Keller.
- Reduce–Eintrag: „rk“: Reduziere nach Produktion k (A $\rightarrow$s). Nimm alle zur Reduktion gehörenden Einträge (Anzahl: |s|) vom Keller (Symbol- und Zustandskeller!) und lege A auf den Keller. A wird als neues Eingabezeichen aufgefasst. Es bestimmt mittels der Sprungtabelle den Folgezustand. Dieser wird auf den Keller gelegt.
- Accept: Erfolgreiches Ende der Syntaxanalyse.
- Sprungtabelle: Der Eintrag (Zustand, Nichtterminal) = (Z, A) enthält den Folgezustand, der nach der Reduktion s $\leftarrow$ A angenommen wird.

LR-Parsing Beispiel 1

R1: S -> AB

R2: A -> aa

R3: B -> ba

	a	b	$	A	B	S
0	s1			goto 3		goto 7
1	s2					
2		r2				
3		s4			goto 6	
4	s5					
5			r3			
6			r1			
7			accept			

LR-Parsing Beispiel 2

Grammatik Beispiel 2:

0: S-> E$
1: E-> E + T
2: E-> E – T
3: E-> T
4: T-> T * F
5: T-> T / F
6: T-> F
7: F-> (E)
8: F-> z

Syntaxanalyse

Fehlerbehandlung:

- Syntaktische Fehlererkennung ist Bestandteil der Syntaxanalyse
- Fortsetzung der Analyse eines fehlerhaften Programms?
  - Heuristische Vorgehensweise beim Wiederaufsetzen:
    - Jeder Fehlersituation ist eine Hypothese über die Art des Fehlers zuzuordnen, z.B.:
      - Terminalsymbol fehlt
      - Schreibfehler beim Lesen des Tokens (Scanner erkennt Fehler und liefert spezielles error-Token)
      - Fehler in einem Schlüsselwort (Scanner liefert Bezeichner)

Syntaxanalyse

Fehlerbehandlung:

- Je nach Fehlerhypothese werden fehlende Symbole ergänzt oder auch fehlerhafte Programmteile ignoriert
- Wiederaufsetzpunkte für die Analyse sind Symbole aus FOLLOW(S)
- Grammatik-Erweiterungen durch Fehler-Produktionen:

  S: ABC | error(1)

  A: aaA| b | error(2)

Syntaxanalyse

Wertung Analyseverfahren

Umfangreiche Parsing-Theorien und -Verfahren verfügbar:

Parser-Klasse	Grammatik-Klasse
Top-Down	
Rekursiver Abstieg	ohne Linksrekursion, LL(k)-Eigenschaft

Syntaxanalyse

Wertung Analyseverfahren

Umfangreiche Parsing-Theorien und -Verfahren verfügbar:

Parser-Klasse	Grammatik-Klasse
Bottom-Up	
SLR-Verfahren	einfachste LR-Methode
LALR-Verfahren	mächtiger, aber komplexer als SLR
kanonisches LR(k)-Verfahren	Konfliktlösung durch Vorausschau auf k Token
Earley-Verfahren	beliebige kontextfreie Grammatik

Parser-Generator yacc

Yacc = yet another compiler-compiler

- unter GNU-Lizenz als bison verfügbar
- Eingabe ist eine Grammatik, die die Syntax der vom Parser zu analysierenden Eingabe beschreibt
- Generator erzeugt aus Grammatik einen tabellengesteuerten Bottom-Up-Parser in Form eines C- (oder C++)-Quelltexts.
- wesentlicher Bestandteil des generierten Parsers ist die Definition der Analysefunktion int yyparse(void):
  - prüft Eingaben auf syntaktische Korrektheit gemäss Grammatik.
  - weitere Verarbeitungsschritte, z.B. Typprüfungen, Erzeugung Syntaxbaum, Symboltabelle Zwischencodeerzeugung möglich
  - Generator unterstützt die Möglichkeit, semantische Aktionen in Form von C-Anweisungen in die Regeln der Grammatik einzubinden.

Parser-Generator yacc

Eingabe-Dateiformat für Yacc

Eingabedatei für Yacc hat folgende Struktur:

%{

C-Deklarationen

%}

yacc-Deklarationen

%%

Grammatik + Aktionen

%%

Parser-Generator yacc

Analyse einer Grammatik

Generator kann verwendet werden, um Grammatik zu testen (z.B. Konfliktfreiheit, Aufbau LR-Parsertabelle):

bison -dtv parser.y

Aufrufoptionen:

-d erzeugt die C-Definitionen für die Tokenklassen in separatem Header-File parser.tab.h

-t erzeugt Code zum Debuggen des generierten Parsers

-v Konfliktmeldungen werden in output-file ausgegeben

Parser-Generator yacc

Ausgabedateien:

parser.tab.h Scanner-Schnittstelle mit Token-Definitionen

parser.tab.c Parser-Quelltext: yyparse()

parser.output Parser-Beschreibungsdatei zum Debuggen der Grammatik