Grundlagen der Programmierung (Vorlesung 21)

Ralf Möller, FH-Wedel

- Vorige Vorlesung
  - Alphabet, Formale Sprache, Grammatik, erzeugte Sprache
- Inhalt dieser Vorlesung
  - Grammatikformen, BNF-Notation
- Lernziele
  - Grundkenntnisse in der Beschreibung von Programmiersprachen (sowie Teilen davon)
  - Kennenlernen von abstrakten Maschinen zur Charakterisierung von Problemen

Grammatik

Definition. Eine *Grammatik* ist ein 4-Tupel $G = (V, \Sigma, P, S)$ das folgende Bedingungen erfüllt:

- V ist eine endliche Menge, die Menge der *Variablen*.
- Σ ist eine endliche Menge, das *Terminalalphabet*, wobei $V \cap \Sigma = \emptyset$.
- P ist die Menge der *Produktionen* oder *Regeln*. P ist eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. (Schreibweise: $(u, v) \in P$ schreibt man meist als $u \to v$.)
- $S \in V$ ist die *Startvariable*.

Übergang

Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G unmittelbar über in v), falls u und v die Form haben

$$
\begin{aligned}
u &= xyz \\
v &= xy'z \quad \text{mit } x, z \in (V \cup \Sigma)^* \text{ und} \\
y &\to y' \quad \text{eine Regel in } P \text{ ist.}
\end{aligned}
$$

(Bem: Falls klar ist, welche Grammatik gemeint ist, so schreiben wir oft auch einfach kurz $u \Rightarrow v$ anstelle von $u \Rightarrow_G v$.)

Transitive Hülle einer Relation: Motivation

- Ein Wort wird durch Produktionsregeln in ein neues Wort abgebildet
- Was in was abgebildet wird, ist durch die Übergangsrelation $\Rightarrow$ gegeben
- Im Kontext einer Grammatik können Produktionsregeln mehrfach hintereinander angewendet werden
- Es soll nun alles, was auch mehrschrittig abgeleitet werden kann, betrachtet werden
- Es ist in diesem Fall nicht die direkte Übergangsrelation zu betrachten, sondern die sogenannte Hülle der Übergangsrelation

Transitive Hülle einer Relation: Definition

- Seien R und S zwei zweistellige Relationen über einer Menge M, so daß $R \subseteq M \times M$ und $S \subseteq M \times M$
- Wir definieren

  $RS := \{ (x, y) \mid \exists z \in M. (x, z) \in R \wedge (z, y) \in S \}$
- Wir definieren $R^0 := \{ (x, x) \mid x \in M \}$ und dann $R^{n+1} := RR^n$
- Damit können wir dann definieren ($n \in N_0$):
  - $R^* := \bigcup_{n \geq 0} R^n$ (transitive, reflexive Hülle)
  - $R^+ := \bigcup_{n \geq 1} R^n$ (transitive Hülle)

Erzeugte Sprache, Ableitung

Die von G definierte (erzeugte, dargestellte) *Sprache* ist

$$L(G) := \{w \in \Sigma^* \mid S \Rightarrow_G^* w\},$$

wobei $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$ ist.

Eine Folge von Worten $(w_0, w_1, \ldots, w_n)$ mit $w_0 = S$, $w_n \in \Sigma^*$ und $w_i \Rightarrow w_{i+1}$ für $i = 0, \ldots, n-1$ heißt *Ableitung* von w_n.

Grammatikformen (1)

Definition. Jede Grammatik ist automatisch vom Typ 0. (D.h., bei Grammatiken von Typ 0 gibt es keinerlei Einschränkungen an die Regeln in P.)

Eine Grammatik ist vom *Typ 1* oder *kontextsensitiv*, falls für alle Regeln $u \to v$ in P gilt: $|u| \leq |v|$.

Eine Grammatik ist vom *Typ 2* oder *kontextfrei*, falls für alle Regeln $u \to v$ in P gilt: $u \in V$ (d.h., u ist eine einzelne Variable) und $|v| \geq 1$.

Grammatikformen (2)

Eine Grammatik ist vom *Typ 3* oder *regulär*, falls für alle Regeln $u \to v$ in P gilt: $u \in V$ (d.h., u ist eine einzelne Variable) und $w \in \Sigma \cup \Sigma V$ (d.h., v ist entweder eine einzelnes Terminalzeichen oder ein Terminalzeichen gefolgt von einer Variablen).

Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ i, $i \in \{0, 1, 2, 3\}$, falls es eine Grammatik vom Typ i gibt mit $L(G) = L$.

Sonderregelung für das leere Wort

ε-Sonderregelung: Wegen $|u| \leq |v|$ kann das leere Wort bei Typ 1,2,3 Grammatiken nicht erzeugt werden. Wir erlauben daher die folgende Sonderregelung: Ist $\varepsilon \in L(G)$ erwünscht, so ist die Regel $S \to \varepsilon$ zugelassen, falls die Startvariable S auf keiner rechten Seite einer Produktion vorkommt.

Beispiele (1)

Typ 3: $L = \{a^n \mid n \in \mathbb{N}\}$,

Grammatik: $S \to a$,

$S \to aS$

Typ 2: $L = \{a^n b^n \mid n \in \mathbb{N}\}$,

Grammatik: $S \to ab$,

$S \to aSb$

Beispiele (2)

Typ 1: $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$,

Grammatik: $S \to aSXY$,

$S \to aXY$,

$XY \to YX$,

$aX \to ab$,

$bX \to bb$,

$bY \to bc$,

$cY \to cc$

Backus-Naur-Form (1)

Backus-Naur-Form: Formalismus zur kompakten Darstellung von Typ 2 Grammatiken

Statt

$$\begin{aligned} A &\rightarrow \beta_1 \\ A &\rightarrow \beta_2 \\ &\vdots \\ A &\rightarrow \beta_n \end{aligned}$$

schreibt man $A ::= \beta_1 | \beta_2 | \ldots | \beta_n.$

Statt

$$\begin{aligned} A &\rightarrow \alpha\gamma \\ A &\rightarrow \alpha\beta\gamma \end{aligned}$$

schreibt man $A ::= \alpha[\beta]\gamma.$

Backus-Naur-Form (2)

Statt

$$A \rightarrow \alpha\gamma$$
$$A \rightarrow \alpha B\gamma$$
$$B \rightarrow \beta$$
$$B \rightarrow \beta B$$

schreibt man $A ::= \alpha\{\beta\}\gamma.$

Beispiel: Arithmetische Ausdrücke

<exp> ::= <term>

<exp> ::= <exp> + <term>

<term> ::= (<exp>)

<term> ::= <term> * <term>

<term> ::= $a \mid b \mid \ldots \mid z$

Aufgabe eines Compilers: Prüfe ob ein gegebener String einen gültigen arithmetischen Ausdruck darstellt und, falls ja, zerlege ihn in seine Bestandteile.

Beispiel: "Unsere" Sprache zur Notation von Algorithmen

Anweisung ::= *Zuweisung*
| *Anweisungsfolge*
| *bedingteAnweisung*
| *SchleifenAnweisung*

Zuweisung ::= *einfacheVariable* := *Ausdruck*

Anweisungsfolge ::= *Anweisung* ; *Anweisung*

bedingteAnweisung ::= if *Ausdruck*
then *Anweisung*
else *Anweisung*
end if
| if *Ausdruck*
then *Anweisung*
end if

Schleifen und einige Erweiterungen

SchleifenAnweisung ::= while *Ausdruck* do
Anweisung
end while

Ausdruck ::= ... | *Ausdrucksfolge*

Ausdrucksfolge ::= *Anweisung* ; *Ausdruck*

Ausdruck ::= ... | *bedingterAusdruck*

bedingterAusdruck ::= if $Ausdruck_0$
then $Ausdruck_1$
else $Ausdruck_2$
end if

Funktionen

FunktionsDefinition	::=	*FunktionsKopf FunktionsRumpf*
FunktionsKopf	::=	*FktName* (*formaleParamListe*)
		: *Typ*
formaleParamListe	::=	*formalerParameter*
	|	*formalerParameter* ; *formaleParamListe*
formalerParameter	::=	*Variable* : *Typ*
FunktionsRumpf	::=	*Ausdruck*
Ausdruck	::=	...
	|	*FunktionsAufruf*
FunktionsAufruf	::=	*FktName* (*aktuelleParamListe*)
aktuelleParamListe	::=	*Ausdruck*
	|	*Ausdruck* , *aktuelleParamListe*

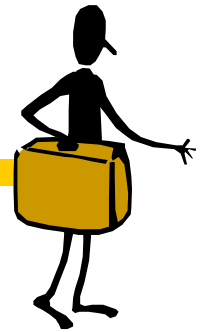
Blöcke

Anweisung	::=	... | *Block*
Block	::=	**begin** *Deklaration* ; *Anweisung* **end**
Deklaration	::=	**var** *Variablenliste*
Variablenliste	::=	*Variablendekl*
	|	*Variablendekl* ; *Variablenliste*
Variablendekl	::=	*Variablen* : *Typ*
Variablen	::=	*Variable*
	|	*Variable* , *Variablen*

Prozedurdefinition

$$
\begin{array}{lll}
\mathit{ProzedurDefinition} & ::= & \mathit{ProzedurKopf}\ \mathit{ProzedurRumpf} \\
\mathit{ProzedurKopf} & ::= & \mathit{ProzedurName}\ (\ \mathit{formaleParamListe}\ ) \\
 & | & \mathit{ProzedurName} \\
\mathit{ProzedurRumpf} & ::= & \mathit{Anweisung} \\
\mathit{Anweisung} & ::= & \ldots \mid \mathit{Prozeduraufruf} \\
\mathit{ProzedurAufruf} & ::= & \mathit{ProzedurName}\ (\ \mathit{aktuelleParamListe}\ ) \\
 & | & \mathit{ProzedurName}
\end{array}
$$

Zusammenfassung, Kernpunkte



- Formale Sprachen
- Grammatiken
- Grammatiktypen
- Entscheidungsprobleme
- Anwendungen

Was kommt beim nächsten Mal?

- Abstrakte Maschinen für spezielle Aufgaben
- Insbesondere zum Entscheiden von Wortproblemen
- Automatentheorie