marc-oliver pahl

Informatik II – Kapitel 18

„Übersetzung“

Zusammenfassung des Kapitel 18
Küchlin, Weber, Vorversion „Einführung in die Informatik, 3.Auflage“

2.7.2004

marc-oliver pahl

Programmcode in höherer Programmiersprache

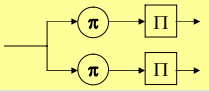
```
VAR a, b;
BEGIN
   READ a;
   …
END.
```

PL/0

scanning and parsing

„Programmcode syntaktisch korrekt?“

LL_1:



Übersetzer (compiler)

```
READ      001 000 0000001110
READ      001 000 0000001110
ADD       001 000 0000000010
…         …
```

M/0

Interpreter

z.B. Auswertung des PL/0 Programms mit Java…

marc-oliver pahl

- **Alphabet**

$$\Sigma = \{a1, \ldots, an\}$$

- Reflexiv-transitive Hülle von Σ wird mit Σ^* bezeichnet:
  - $\varepsilon \in \Sigma^*$ (das leere Wort)
  - $\forall\, a \in \Sigma: a \in \Sigma^*$
  - $\forall\, \omega \in \Sigma^*, a \in \Sigma: \omega \circ a \in \Sigma^*$
- **(Formale) Sprachen** umfassen eine **Teilmenge von Σ^***.
- **Konkatenation**: $\omega \circ \varepsilon = \varepsilon \circ \omega = \omega$ ($< \Sigma^*; \circ, \varepsilon >$ ist Monoid (Halbgruppe mit Neutralelement))
- **Produkt**: Für $L_1, L_2 \subseteq \Sigma^*$ sei $L_1 L_2 := \{ \omega_1 \circ \omega_2 \mid \omega_1 \in L_1, \omega_2 \in L_2 \}$
  - $L^0 := \{\varepsilon\}$
  - $L^+ := \cup_{n>0} L^n$
  - $L^* := \cup_{n>=0} L^n$

marc-oliver pahl

- Eine (formale) **Sprache** wird durch ihre **Grammatik** charakterisiert.

$$G = < N, T, P, S >$$

- N = Non-Terminal-Symbole
- T = Terminal-Symbole
- $S \in N$, Startsymbol
- P = Produktionen der Form $\Theta \rightarrow \vartheta$ mit $\Theta, \vartheta \subseteq (N \cup T)^+$
- $N \cap T = \emptyset$

- Als **Satzform** bezeichnet man jede gültige Kombination aus $(T, N)^*$.
- Als **Satz** bezeichnet man eine durch die Grammatik erstellbare Kombination aus T^*

Die *Vereinigung aller Sätze*, die durch die Grammatik erstellt werden können bezeichnet man als **Sprache L(G)**.

Chomsky-Hierarchie der Grammatiken

marc-oliver pahl

Chomsky **0**: Jede Grammatik ist automatisch vom Typ 0. Das heißt, bei Typ 0 sind den Regeln keinerlei Einschränkungen auferlegt.

Chomsky **1**: Eine Grammatik ist vom Typ 1 oder **kontextsensitiv**, falls für alle Regeln $\omega_1 \rightarrow \omega_2$ in P gilt $|\omega_1| \leq |\omega_2|$. *„aBc -> abDc“*

Chomsky **2**: Eine Typ 1-Grammatik ist vom Typ 2 oder **kontextfrei**, falls für alle Regeln $\omega_1 \rightarrow \omega_2$ in P gilt, dass ω_1 eine einzelne Variable ist, d.h. $\omega_1 \in N$. *„A -> a | MNO“*

Chomsky **3**: Eine Typ 2-Grammatik ist vom Typ 3 oder **regulär**, falls zusätzlich gilt: $\omega_2 \in \Sigma \cup \Sigma N$, d.h. die rechten Seiten von Regeln sind entweder einzelne Terminalzeichen oder ein Terminalzeichen gefolgt von einer Variablen. *„A -> a | aB“*

Aus: Schöning, Theoretische Informatik-kurzgefaßt, 3. Auflage, Spektrum Verlag

Chomsky-Hierarchie der Grammatiken

marc-oliver pahl

Chomsky **0**: keinerlei Einschränkungen

Chomsky **1**: **kontextsensitiv**, falls für alle Regeln $\omega_1 \rightarrow \omega_2$ in P gilt $|\omega_1| \leq |\omega_2|$.

Chomsky **2**: **kontextfrei**, falls für alle Regeln $\omega_1 \rightarrow \omega_2$ in P gilt $\omega_1 \in N$

Chomsky **3**: **regulär**, falls zusätzlich gilt: $\omega_2 \in \Sigma \cup \Sigma N$

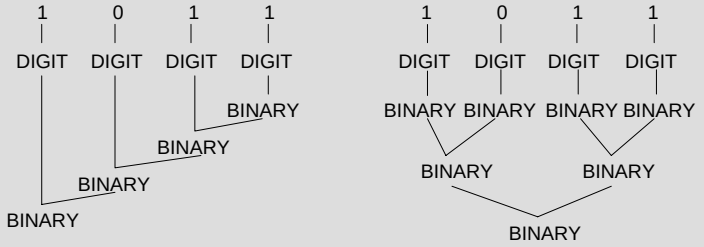
Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ 0 (Typ 1, …), falls es eine Typ 0 (Typ 1, …) -Grammatik G gibt mit L(G) = L.

$$\text{Typ } 0 \supseteq \text{Typ } 1 \supseteq \text{Typ } 2 \supseteq \text{LL(1)} \supseteq \text{Typ3}$$

Ableitungsbaeume

marc-oliver pahl

N = {BINARY, DIGIT}
T = {0,1}
S = BINARY
P = { BINARY -> DIGIT BINARY | BINARY BINARY | BINARY DIGIT
BINARY -> DIGIT
DIGIT -> 0
DIGIT -> 1
}



BNF

marc-oliver pahl

BNF:

$$A \rightarrow \varepsilon$$

$$A \rightarrow \alpha$$

$$A \rightarrow \alpha_0 \mid \alpha_1 \mid \alpha_2 \mid \alpha_3$$

EBNF:

A –> ε
A –> α
A –> $\alpha_0 \mid \alpha_1 \mid \alpha_2 \mid \alpha_3$

„Optionsklammern“

[α]

⇔ Vorkommen von [α] durch N_α ersetzen und Zusatzregel N_α –> ε | α

„Repetitionsklammern“

{ α }

⇔ –> { α } => $N_\alpha \wedge N_\alpha$ –> ε | $N_\alpha\alpha$

„Gruppierungsklammern“

(α)

⇔ (α) => $N_\alpha \wedge N_\alpha$ –> α

B_0 : Der Parser einer EBNF-Grammatik ist die Kollektion der Funktionen der Produktionen und der Hauptfunktion

```
void parse ()
{ get_token (); S(); }
```

B_1 : Jede Produktion $Id \rightarrow Expr.$ ergibt eine Funktion

```
void Id ()
{ P(Expr);}
```

B_2 : Ein Ausdruck $\tau_1|\tau_2|\ldots|\tau_n$ ergibt

```
if (curr_tok ∈ First(τ1)) { P(τ1);}
else if (curr_tok ∈ First(τ2)) { P(τ2);}
...
else if (curr_tok ∈ First(τn)) { P(τn);}
else error();
```

Bemerkung:
Wegen der LL_1 Bedingung schließen sich die Fälle wechselseitig aus. Ist `First(τi)` eine Menge mit einem einzigen Element (**singleton set**), so benutzt man ein switch statement.

B_3 : Ein Term $\phi_1\phi_2\ldots\phi_n$ ergibt

```
{ P(φ1); P(φ2); ... ; P(φn);}
```

B_4 : Ein Terminal “string” ergibt

```
if (curr_tok == Token ("string")) get_token();
else error();
```

B_5 : Ein Nichtterminal N ergibt den Funktionsaufruf

```
N();
```

B_6 : Eine Gruppierung (ξ) ergibt

```
{ P(ξ); }
```

B_7 : Eine Option [ξ] ergibt

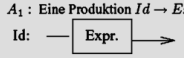
```
if (curr_tok ∈ First(ξ)) { P(ξ); }
```

B_8 : Eine Repetition {ξ} ergibt

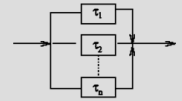
```
while (curr_tok ∈ First(ξ)) { P(ξ); }
```


A_0 : Das Syntaxdiagramm einer EBNF-Grammatik ist die Kollektion der Syntaxdiagramme der Produktionen.

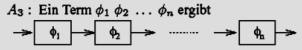
A_1 : Eine Produktion $Id \rightarrow Expr.$ ergibt das Diagramm



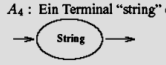
A_2 : Ein Ausdruck $\tau_1|\tau_2|\ldots|\tau_n$ ergibt



A_3 : Ein Term $\phi_1\ \phi_2\ \ldots\ \phi_n$ ergibt

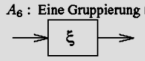


A_4 : Ein Terminal “string” ergibt

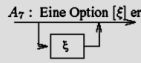


A_5 : Ein Nichtterminal N ergibt

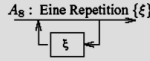
A_6 : Eine Gruppierung (ξ) ergibt

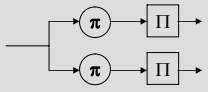


A_7 : Eine Option [ξ] ergibt



A_8 : Eine Repetition {ξ} ergibt





πϵT, ΠϵN

<- NICHT LL_1

Eine Sprache ist **LL_1**, wenn man an jeder Verzweigung des zugehörigen Syntaxdiagramms in **jedem Zweig** auf ein **anderes Terminalsymbol** trifft.

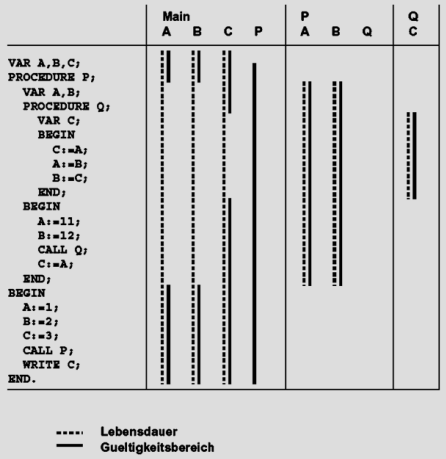
$$first(\xi) = \{\sigma \in T | \xi \rightarrow^* \sigma\eta, \eta \in (N \cup T)^*\}$$
$$follow(\xi) = \{\sigma \in T | S \rightarrow^* \alpha\beta\sigma\gamma, \text{wobei } \xi \rightarrow^* \beta, \alpha, \gamma \in (N \cup T)^*\}.$$

Für eine LL(1) Grammatik muß gelten:

1. Jeder EBNF-Ausdruck $\xi = \tau_1|\tau_2|\ldots|\tau_n$ muß disjunkte First-Mengen der Alternativen τ_1 - τ_n haben, also
$$first(\tau_i) \cap first(\tau_j) = \emptyset \qquad \forall i \neq j.$$
2. Jeder EBNF-Ausdruck ξ aus dem das leere Wort ε erzeugt werden kann, muß disjunkte First- und Follow-Mengen haben, also:
$$\xi \rightarrow^* \varepsilon \Rightarrow first(\xi) \cap follow(\xi) = \emptyset.$$

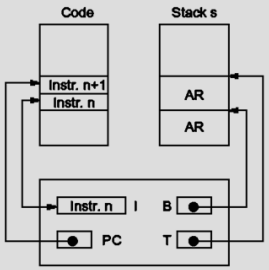
Lebensdauer Symbole

marc-oliver pahl



Uebersetzung

marc-oliver pahl



- code: Programmspeicher (Array of Instructions)
- s: Datenspeicher (Stack)

Uebersetzung

marc-oliver pahl

Mnemonic	Beschreibung
LIT a	Lade a auf Stack.
NEG	Negiere oberstes Stapelelement.
ADD SUB MUL DIV	Arithmetische Operationen: Die beiden obersten Stapelelemente werden miteinander verknüpft und durch das Ergebnis ersetzt.
ODD	Ersetzt oberstes Stapelelement durch 1 falls dieses ungerade ist, sonst durch 0.
EQ NEQ LT LTE GT GTE	Vergleichsoperationen: Die beiden obersten Stapelelemente werden verglichen und durch 1 ersetzt, falls die Auswertung true ergibt, sonst werden sie durch 0 ersetzt.
READ	Lade eingegebenen Wert auf Stapel.
WRITE	Gebe oberstes Stapelelement aus.
LOD l,a	Lade Variable mit der relativen Adresse (l,a) auf Stapel.
STO l,a	Speichere oberstes Stapelelement an relativer Adresse (l,a).
CAL l,a	Rufe Unterprogramm auf. Das UP beginnt an Adresse a und wurde l Blöcke außerhalb vereinbart.
RET	Springe aus Unterprogramm zurück.
INT a	Erhöhe Stapelzeiger T um a.
JMP a	Springe zur Adresse a.
JPC a	Springe zur Adresse a, falls oberstes Stapelelement 0 ist.

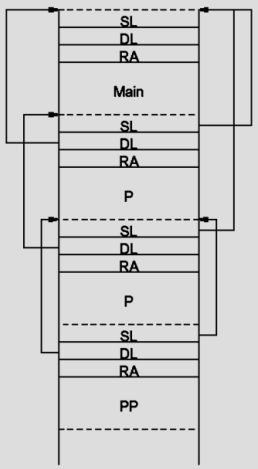
Uebersetzung

marc-oliver pahl

Assembler	(f,l,a)	Funktion	Beschreibung
LIT 0,a	(0,0,a)	T=T+1; s[T]=a	Load Immediate
OPR 0,0	(1,0,0)	T=B-1; PC=s[T+3]; B=s[T+2]	Return
OPR 0,1	(1,0,1)	s[T]=-s[T]	Negate
OPR 0,2	(1,0,2)	T=T-1; s[T]=s[T]+s[T+1]	Add
OPR 0,3	(1,0,3)	T=T-1; s[T]=s[T]-s[T+1]	Subtract
OPR 0,4	(1,0,4)	T=T-1; s[T]=s[T]*s[T+1]	Multiply
OPR 0,5	(1,0,5)	T=T-1; s[T]=s[T]/s[T+1]	Divide
OPR 0,6	(1,0,6)	s[T]=ORD(ODD(s[T]))	Odd
OPR 0,7	(1,0,7)		No Operation
OPR 0,8	(1,0,8)	T=T-1; s[T]=ORD(s[T]==s[T+1])	Equal
OPR 0,9	(1,0,9)	T=T-1; s[T]=ORD(s[T]#s[T+1])	Not Equal
OPR 0,10	(1,0,10)	T=T-1; s[T]=ORD(s[T]<s[T+1])	Less
OPR 0,11	(1,0,11)	T=T-1; s[T]=ORD(s[T]>=s[T+1])	Greater or Equal
OPR 0,12	(1,0,12)	T=T-1; s[T]=ORD(s[T]>s[T+1])	Greater
OPR 0,13	(1,0,13)	T=T-1; s[T]=ORD(s[T]<=s[T+1])	Less or Equal
OPR 0,14	(1,0,14)	T=T+1; read s[T]	Read
OPR 0,15	(1,0,15)	write s[T]; T=T-1	Write
LOD l,a	(2,l,a)	T=T+1; s[T]=s[base(l)+a]	Load
STO l,a	(3,l,a)	s[base(l)+a]=s[T]; T=T-1	Store
CAL l,a	(4,l,a)	s[T+1]=base(l); s[T+2]=B; s[T+3]=PC; B=T+1; PC=a	Call
INT 0,a	(5,0,a)	T=T+a	Increment T
JMP 0,a	(6,0,a)	PC=a	Jump
JPC 0,a	(7,0,a)	if (s[T]==0) PC=a; T=T-1	Cond. Jump

Uebersetzung

marc-oliver pahl



Der dynamische Link (**DL**) zeigt auf den Aufruf-Stack (z.B. rekursiver Aufruf von sich selbst). Dieser wird gültig, sobald die Prozedur zurückkehrt.

Der statische Link (**SL**) zeigt auf den Statischen Vater (nächste Ebene, z.B. um Variablen zu finden LOD l, a).

Die Rücksprungadresse (**RA**) gibt den ProgramCounter (PC) an, bei dem nach Beendigung des Aufrufs fortgefahren werden soll.

Beispielprogramm

marc-oliver pahl

Das Programm soll:

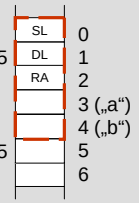
- 2 Werte einlesen
- Diese addieren
- Den Betrag des Ergebnisses ausgeben

```
VAR a, b;
BEGIN
  READ a;
  READ b;
  a := a + b;
  IF a<0 THEN a := -a;
  WRITE a;
END.
```

Beispielprogramm

marc-oliver pahl

PL/0	*M/0*		*Durch die Operation betroffene **Stack**position(en)*
		CAL 0,@1	
	@2:	JMP @2	
	@1:	INT 5	
VAR a, b;			
BEGIN			
READ a;		READ	5
		STO 0,3	(5) 3
READ b;		READ	5
		STO 0,4	(5) 4
a := a + b;		LOD 0,3	(3) 5
		LOD 0,4	(4) 6
		ADD	(5,6)5
		STO 0,3	(5) 3
IF a<0 THEN a := -a;		LOD 0,3	(3) 5
		LIT 0	6
		LS	(5,6)5
		JPC @3	(5)
		LOD 0,3	(3) 5
		NEG	5
		STO 0,3	(5) 3
	@3:	LOD 0,3	(3) 5
WRITE a;		WRITE	(5)
END.		RET	



(?) = lesender Zugriff