VO Übersetzerbau 1 WS 1996/1997

1 EINLEITUNG UND ÜBERSICHT 3

1.1 GRUNDBEGRIFFE 3
1.2 STATISCHE GROBSTRUKTUR VON COMPILERN 3
1.2.1 Lexikalische Analyse 3
1.2.2 Syntaxanalyse 4
1.2.3 Semantische Analyse 4
1.2.4 Optimierung 4
1.2.5 Codegenerierung 4
1.3 DYNAMISCHE GROBSTRUKTUR 5
1.4 VORLESUNGSSTOFF 5
1.5 LITERATURHINWEIS 6

2 LEXIKALISCHE ANALYSE 6

2.1 AUFGABEN 6
2.2 ALGORITHMISCHE STRUKTUREN 7
2.3 TEILAUFGABEN 7
2.4 AUFBAU DER SCHLÜSSELWORT- UND NAMENLISTE 7
2.4.1 Namen und Schlüssel gemeinsam 7
2.4.2 Eigene Schlüsselwortliste 8
2.4.3 Schlüsselwörter in Hash-Code speichern 8
2.5 NAMENLISTBEHANDLUNG 8
2.5.1 Statische Namenliste allein 8
2.5.2 Statische Namenliste und Deskriptorenliste mit n Namen der mittleren Länge m 8
2.5.3 Dynamische Namenliste 8
2.6 BESONDERE PROBLEME 9
2.7 SCANNER-GENERATOREN 9

3 SYNTAXANALYSE I 9

3.1 OHNE FEHLERBEHANDLUNG 10
3.2 FEHLERBEHANDLUNG 11
3.2.1 Fehler erkennen und anzeigen 11
3.2.2 Folgefehler unterdrücken 12
3.2.3 Wiederaufsetzen 12

4 SYNTAXANALYSE II 13

4.1 TOPDOWN-GRAPH (ABSPEICHERN DER GRAMMATIK-TABELLE) 13
4.2 ANALYSEALGORITHMUS 15
4.3 FEHLERBEHANDLUNG 15

5 FORMALE BESCHREIBUNG VON ÜBERSETZUNGSPROZESSEN 16

5.1 SEMANTISCHE AKTIONEN 16
5.2 ATTRIBUTE 16
5.3 KONTEXTBEDINGUNGEN 17
5.4 ATTRIBUTIERTE GRAMMATIKEN (AG) 18
5.5 L-ATTRIBUTIERE GRAMMATIKEN 18

6 IMPLEMENTIERUNG DES SEMANTIKANSCHLUSSES 19

6.1 BEIM REKURSIVEN ABSTIEG 19
6.2 BEI DER TABELLENGESTEUERTEN ANALYSE 20

7 ZWISCHENSPRACHEN, ABSTRAKTE MASCHINEN UND INTERPRETATION 22

7.1 ZWISCHENSPRACHEN 22
7.2 ABSTRAKTE MASCHINEN 23
7.3 DIE ABBILDUNG DER QUELLSPRACHE AUF DIE ZIELSPRACHE 23
7.4 DIE MINIMODULA MASCHINE 23

8 DIE ÜBERSETZUNG VON ABLAUFSTRUKTUREN 23

8.1 VORBEMERKUNGEN 23
8.2 MARKEN UND SPRUNGANWEISUNGEN 24

8.3 IF-ANWEISUNGEN 25
8.4 CASE-ANWEISUNG 25
8.5 SCHLEIFENANWEISUNGEN 25

9 DATENARTEN UND DATENTYPEN 26

9.1 OBJEKTE UND IHRE BESCHREIBUNG IM COMPILER 26
9.2 AUSWIRKUNG DER DATENTYPEN AUF OPERTIONEN (S. SKRIPT) 27
9.3 SELEKTION 27

10 DYNAMISCHE SPEICHERVERTEILUNG 29

10.1 DIE SEMANTIK BLOCKSTRUKTURIERTER PROGRAMME 31
10.2 ÜBERSETZUNG BLOCKSTRUKTURIERTER SPRACHEN 33

11 INDEX 36

1 Einleitung und Übersicht

Fülle von Varianten:
- Quellensprache
- Zielsprachen
- Syntaxanalyse
- Optimierungsverfahren

Sprachenvielfalt:
- Imperative Sprachen
  - Prozedurorientiert (z. B. Pascal)
  - Modulorientiert
  - Objektorientiert
- Funktionale Sprachen
- Deklarative Sprachen (z. B. Prolog)
- Sprachen mit und ohne Parallelität

2 Ebenen (des Geschehens):
- Übersetzungs- und
- Laufzeit

1.1 Grundbegriffe

Compiler: 1:n-Übersetzung
Assembler: 1:1-Übersetzung

In welchen Sprachen soll man einen Compiler schreiben? In höheren algorithmischen Sprachen wie Pascal, Modula, Oberon.

1.2 Statische Grobstruktur von Compilern

1.2.1 Lexikalische Analyse

(Kommentare und Leerzeichen fallen weg)

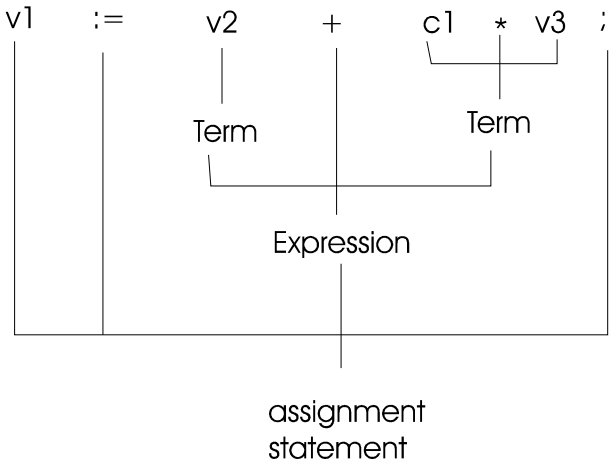
Eingabe:	*adr*	*:=*	*base*	+	*4*	*	*dist ;*
Compiler:	v_1	*:=*	v_2	+	c_1	*	v_3 *;*

	Name
1	*adr*
2	*base*
3	*dist*

	Konstante
1	*4*

Die Lexikalische Analyse entscheidet über die Übersetzungszeit um bis zu 30%.

1.2.2 Syntaxanalyse

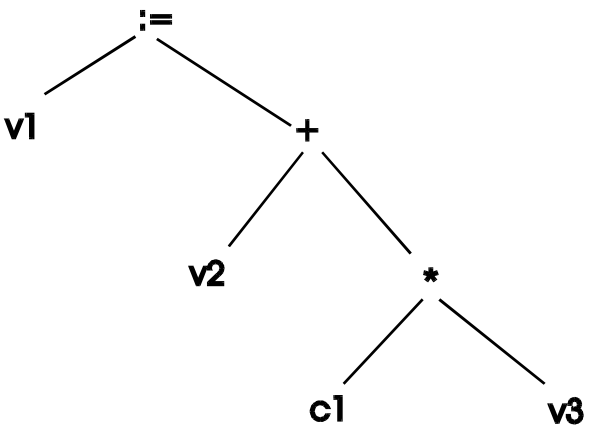


1.2.3 Semantische Analyse

(";"und Klammern fallen weg)

- passen Typ von einem Operanden mit dem Typ des anderen Operanden zusammen?
- ...

Zwischensprache:



lineare Darstellung dieser Baumstruktur:

umgekehrt polnische Notation: $v_1\ v_2\ c_1\ v_3\ *\ +\ :=$

1.2.4 Optimierung

Bäume eignen sich dazu am Besten.

1.2.5 Codegenerierung

MOVE c_1, R_1
MUL v_3, R_1
ADD v_2, R_1
MOVE R_1, v_1

Namenliste wird nur unter Umständen (Fehlermeldungen ausgeben) verwendet

In **Symbolliste** sind gespeichert: TYPE, SPEICHERKLASSE

1.3 Dynamische Grobstruktur

- **Mehrpaß-Compiler:** Hintereinanderausführung (es gibt Zwischenprogramme)
- **Einpaß-Compiler:** Syntaxanalysator steuert Lexikalischen Analysator und Zwischensprachenerzeugung ⇒ es gibt keine Zwischenprogramme

	Einpaß	**Mehrpaß**
Geschwindigkeit	**+** (da kein Dateien-schreiben erforderlich)	**-**
Speicherbedarf	**-** (da alles im Arbeitsspeicher)	**+**
Verstehbarkeit	**-**	**+**
universelle Anwendbarkeit	**--** (da z. B. Variablen-deklaration erst später möglich ist ⇒ kann keinen Code erzeugen, da vom Anfang an bekannt sein muß	**+**
Optimierung	**-**	**+**

Insgesamt sollte man, sofern es die Quellensprache erlaubt, einen Einpaßcompiler verwenden.

1.4 Vorlesungsstoff

- Lexikalische Analyse
- Syntaxanalyse
- Semantische Analyse
- Zwischencodeerzeugung
- Ausführung auf abstrakter Maschine
- Ablaufstrukturen
- Datenstrukturen
- dynamische Speicherverteilung

1.5 Literaturhinweis

N. Wirth:
Grundlagen und Techniken des Compilerbaus
Addison-Wesley 1996 (ATS 295,--)

2 Lexikalische Analyse

2.1 Aufgaben

- Zeichenweises Lesen des Quellprogrammes
- Überlesen bedeutungsloser Zeichen (mit Ausnahmen: z. B. in Zeichenketten sind Leerzeichen keine bedeutungslose Zeichen)
- Erkennen und Arithmetisieren von Terminalsymbolen (= Schlüsselwörter, Namen, Zahlen, Operatoren, Sonderzeichen)
- Speicherung aller gelesener Namen in einer Namenliste

Symbolarten:
type Symbol = (identSy, numberSy, plusSy, ..., illegalSy, noSy)

Terminalklassen (d. h., Symbolwert ≠ 0)
Terminalsymbol = (Symbolart, Symbolwert)

a := b + z ;

Terminalsymbol	Symbolart (sy)	Symbolwert (value)
a	identSy	1
:=	assignSy	0
b	identSy	2
+	plusSy	0
2	numberSy	2
;	semicolonSy	0

	Namenliste

1	a
2	b
...	...

NewSym(↑sy ↑value)
param sy : Symbol
value: Integer;

2.2 Algorithmische Strukturen

*adr := base + 4 * dist ;*
↑ : Ende von *adr* erkannt und Anfang von *:=* erkannt

Lies und Erkenne nächstes Terminalsymbol:

```
case ch of
    Buchstabe:   ErkenneNamen; sy := ...; -- identSy oder Schlüsselwort
    | Ziffer:    ErkenneZahl; TransformiereInsDezimalSystem;
                 sy := numberSy; value := ...;
    | "+":       sy := plusSy; ch := NächstesZeichen;
    | "-":
    | ...
    else         sy := illegalSy;      -- wird vom Syntaxanalysator verarbeitet
end
```

Nicht berücksichtigt:

- Zeichenketten
- Zeilenende
- Dateiende
- Tabulatorzeichen

2.3 Teilaufgaben

ch := nächstes Quellzeichen

- überlies Kommentar
- Erkenne Namen
- Name ist Schlüsselwort
- Erkenne Zahl

2.4 Aufbau der Schlüsselwort- und Namenliste

(Schlüsselwortbehandlung)

2.4.1 Namen und Schlüssel gemeinsam

m Schlüsselwörter (m ≈ 30)

n Namen maximal

type Name = Array[1 : maxnamelength] of Char;
var namelist : Array[1 : m + n] of Name;

2.4.2 Eigene Schlüsselwortliste

type Keyword = Array[1 : maxkeylength] of Char;
var keylist : Array[1 : m] of Keyword;

Nachteil:
- Suchen in 2 Listen

Vorteil:
- weniger Speicher, weil Schlüsselwörter meist kurz sind
- Schlüsselwortliste kann sortiert werden ⇒ binäres Suchen

2.4.3 Schlüsselwörter in Hash-Code speichern

- besondere Technik
- Kollissionsfrei

2.5 Namenlistbehandlung

2.5.1 Statische Namenliste allein

2.5.2 Statische Namenliste und Deskriptorenliste mit n Namen der mittleren Länge m

z. B.: n = 500 und m = 6

1	...	...	

1	
...	...

2.5.3 Dynamische Namenliste

Jeder Name ist Knoten einer dynamischen Datenstruktur

Suchverfahren in dynamischer Namenliste:

- Lineare Liste: Suchzeit ≈ n
- Baum-Liste: Suchzeit ≈ ld(2)
- Hash-Liste: Suchzeit ≈ const

2.6 Besondere Probleme

- Unsicheres Wortende (z. B. Fortran)
- Nichtreservierte Schlüsselworte (z. B. PL/1)
  if ***if*** *then* ***then*** **=** ***else*** *else* ***else*** **=** ***then****;*
- Makros
- Include-Dateien
- Compiler Kommandos

2.7 Scanner-Generatoren

3 Syntaxanalyse I

(Verfahren des rekursiven Abstiegs)
Vom Satzsymbol ausgehend, versucht man den vorgegebenen Satz zu erkennen

Voraussetzung:
Alle Alternativen einer Regel müssen mit verschiedenen Symbolen dargestellt werden

3.1 Ohne Fehlerbehandlung

Erkennung der Regel:
$A \rightarrow aBc$

```
A(↑success)
        Erkenne a(↑success); if not success then return end;
        B(↑success); if not success then return end;
        Erkenne c(↑success); if not success then return end;
end A;

Erkenne a(↑success)   -- sy ist nächstes Quellsymbol
        if sy ≠ aSy then
                success = FALSE;
                return
        end;
        NewSym(↑sy ↑value);
        success = TRUE;
end Erkenne a;
```

Problem 1:
$A \rightarrow \alpha\beta \mid \alpha\gamma$

umschreiben zu:
$A \rightarrow \alpha\,(\beta \mid \gamma)$

```
stat =  if E then stat end |
        if E then stat else stat end
```

wird geändert zu:

```
stat =  if E then stat (ε | else stat) end
```

oder zu:

```
stat =  if E then stat [else stat] end
```

Problem 2:
Linksrekursivität
$A \rightarrow \alpha \mid A\beta$
$A \Rightarrow^{+} \alpha\beta^{*}$

$E \rightarrow E + T$
$E \rightarrow T \{"+" T\}$

```
E:
        T;
        if not success then return end;
        while sy = plusSy do
                NewSym(↑sy ↑value);
                T;
                if not success then return end;
        end;
end E;
```

3.2 Fehlerbehandlung

- Wiederaufsetzen
- Synchronisieren
- Weiterarbeiten

3.2.1 Fehler erkennen und anzeigen

Fehlerursache:
if sy ≠ aSy then success = FALSE; return end;

Fehler anzeigen:
Ersetze *return* durch *Error(↓n); return.*

```
Lexan
        static col : integer;

        NewSym(↑sy ↑value);
        ...
        end NewSym;

        Error(↓n)
                Fange neue Zeile an;
                Drucke (col - 1) Leerzeichen;
                Drucke Fehlermeldung in Abhängigkeit von n;
        end Error;
end Lexan;
```

3.2.2 Folgefehler unterdrücken

3.2.3 Wiederaufsetzen

Verfahren der speziellen Fangsymbole

Menge der *Fangsymbole* = Anweisungsenden ∪ Terminalen Anfängen von Anweisungen ∪ Dateiendesymbol

F = {semicolonSy, ifSy, whileSy, ..., eofSy}

```
Recover(↕sy ↕value)
begin  -- sy paßt nicht zur Syntax
        while sy ∈ F do
                NewSym(↑sy ↑value);
        end;
        if sy = semicolonSy then NewSym(↑sy ↑value); end;
        -- sy ist Anweisungsanfang oder eof
end Recover;
```

Durch *Recover* wird der "**panic-mode**" unterdrückt. Dies bedeutet, daß nicht beim 1. Fehler panikartig der Compiler stoppt, sondern versucht nach der Methode der speziellen Fangsymbole sinnvoll wieder aufzusetzen.

Verfahren der allgemeinen Nachfolgersymbole

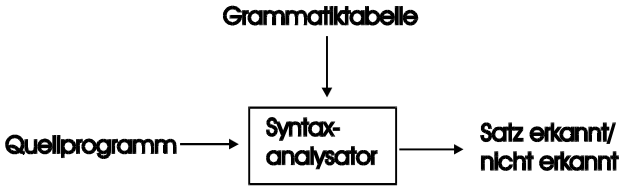
$S = E_1$ ";" .
$E = T_1$ { ("+" | "-") T_2 } .
$T = v$ | "(" E_2 ")" .

$NF(E_1) = \{";"\}$
$NF(E_2) = \{")"\}$
$NF(T_1) = \{"+", "-"\} \cup NF(E_1) \cup NF(E_2) = \{"+", "-", ";", ")"$
$NF(T_2) = NF(T_1) = \{"+", "-", ";", ")"\}$
$NF(S) = \{eof\}$

Verlangsamt die Syntaxanalyse entscheidend (auch bei fehlerlosen Programmen), da die Nachfolgemenge dem rekursiven Aufruf als Parameter mitgegeben wird!

4 Syntaxanalyse II

(Tabellengesteuerte Analyse)
Da bei Syntaxanalyse I für jedes Programm ein eigener Analysator geschrieben werden muß, sucht man nun nach einer Methode, einen Syntaxanalysator zu schreiben, der alle Grammatiken analysieren kann.



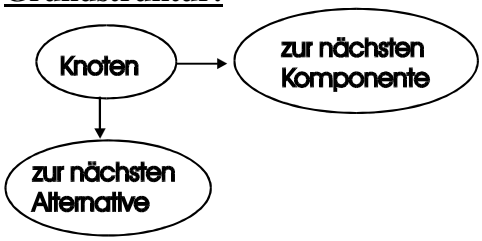
Bei einer neuen Grammatik muß nur die Grammatiktabelle neu erzeugt werden.

4.1 Topdown-Graph (Abspeichern der Grammatik-Tabelle)

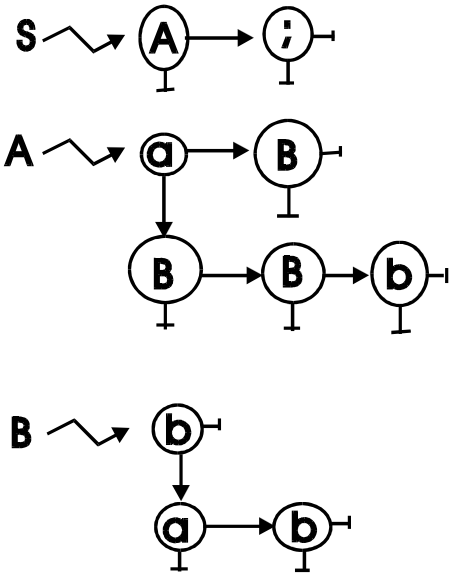
$X_1 \rightarrow \alpha_{11} \mid ... \mid \alpha_{1n}$
...
$X_n \rightarrow \alpha_{n1} \mid ... \mid \alpha_{nm}$

$\alpha_{ij} = S_{ij1} ... S_{ijp}$

Grundstruktur:



$S \rightarrow A;$
$A \rightarrow aB \mid BBb$
$B \rightarrow b \mid ab$



Vorteile:

- **Faktorisierung**
  $A \rightarrow a\ b\ c \mid a\ b\ d$ wird zu $A \rightarrow a\ b\ (c \mid d)$
  $A \rightarrow a\ b\ c \mid e\ f\ c$ wird zu $A \rightarrow (a\ b \mid e\ f)\ c$
  $A \rightarrow a\ b \mid a\ b\ c$ wird zu $A = a\ b\ [\ c\ ]$.
- Beseitigung von Linksrekursivität

$A \rightarrow a \mid A\ b$ wird zu $A = a\ [\ b\ ]$.

Repräsentation des Topdown-Graphen

```
type Graphnode = Record
(1)          kind : (terminal, nonterminal, eps);
(1)          val : Integer; (Symbol)
(2)          lp : Integer;        -- zur Alternative oder 0
(2)          rp : Integer;        -- zum Nachfolger oder 0
(6 Bytes)    end;
```

Ein Topdown-Graph wird statisch abgespeichert, da Indizes kürzer als Zeiger sind, und Grammatiken sowie so ebenfalls statisch sind.

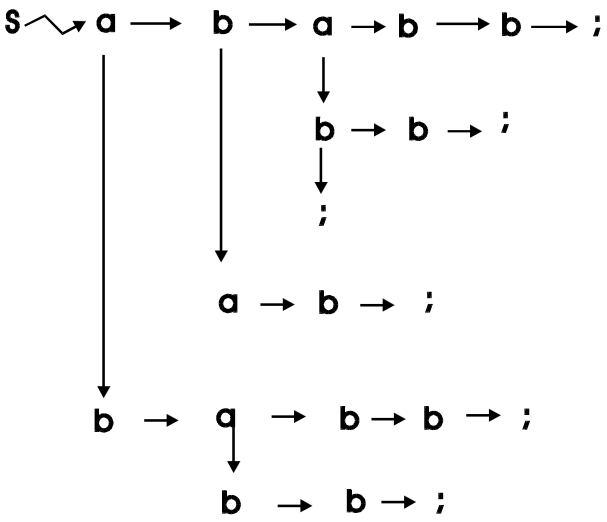
Beispiel Ada-Grammatik

160 Nonterminalsymbole
94 Terminalsymbole
614 Alternativen
1254 Knoten
7300 Byte für Topdown-Graph
320 Byte für Linke Seiten (NT-Symbole)
1920 Byte für terminale Anfänge
9540 Bytes

4.2 Analysealgorithmus

Algorithmus für Topdown-Graph nur aus Terminalsymbolen:

$S \rightarrow A;$
$A \rightarrow aB \mid BBb$
$A \rightarrow b \mid ab$



σ = *babb;*

Kommt man an ein rechtes Ende, so wurde der Satz σ erkannt, bei einem unteren Ende wurde der Satz nicht erkannt!

Match*(↓val ↓sy): BOOLEAN ⇔ NT val kann mit sy anfangen*

Root*(↓val): INTEGER ⇔ Knotennummer der Wurzel des Baumes für das NT-Symbol val*

4.3 Fehlerbehandlung

Im Fehlerfall kann man die Nachfolgesymbole direkt aus dem Graphen auslesen.

5 Formale Beschreibung von Übersetzungsprozessen

5.1 Semantische Aktionen

$A \rightarrow \omega_1$ *sem ... endsem* ω_2

Nach Erkennen von ω_1 und vor Erkennen von ω_2 möchte man "Informationen" einfügen.

Beispiel:
Eingabe: 01101
Ausgabe: 10110

Grammatik:
$S \rightarrow$ *0 S* | *1 S* | ε

$S \rightarrow$	*0 S*	*sem Write("0"); endsem*
	| *1 S*	*sem Write("1"); endsem*
	| ε	

Beispiel:
Eingabe: Nullen und Einsen
Ausgabe: zuerst alle Nullen und dann die Einsen

Grammatik:
$S \rightarrow$ *0 S* | *1 S* | ε

$S \rightarrow$	*0*	*sem Write("0"); endsem*
	S	
	| *1 S*	*sem Write("1"); endsem*
	| ε	

5.2 Attribute

Eingabe: 79 + 18
Ausgabe: 97
$c_{\uparrow x}$ + $c_{\uparrow y}$ = *sem Write(x + y); endsem*

Vereinfachung der Schreibweisen:
statt
$F_{\uparrow a} \rightarrow c_{\uparrow b}$ *sem a := b; endsem*
schreibt man
$F_{\uparrow b} \rightarrow c_{\uparrow b}$

Attributierte Grammtik mit semantischen Aktionen bei EBNF:

$S =$	$E_{\uparrow a}$	*sem Write(a); endsem*
	.	
$E_{\uparrow a} =$	$T_{\uparrow a}$ {"+" $T_{\uparrow b}$	*sem a := a + b; endsem*
	}.	
$T_{\uparrow a} =$	$F_{\uparrow a}$ {"*" $F_{\uparrow b}$	*sem a := a * b; endsem*
	}.	
$F_{\uparrow a} =$	$c_{\uparrow a}$ | "(" $E_{\uparrow a}$ ")".	

Ausgangsattribute: derived attributes oder synthesized attributes
Eingangsattribute: inherited attributes

5.3 Kontextbedingungen

Die Syntax alleine reicht nicht aus, um eine Sprache vollkommen zu beschreiben:

Beispiel:

```
PROCEDURE P;
        Var a, b, a : Integer;
BEGIN
        x := y;
END P;
```

ist ein syntaktisch korrektes aber ein Unsinn ⇒ kein korrektes Programm!

statische Semantik = Menge aller Kontextbedingungen

Beispiel:

assign = $id_{\uparrow name \uparrow type1}$ ":=" $expr_{\uparrow name \uparrow type2}$ *where (type1 = type2)*

KB oder CC: type1 = type2

Wie kann man z. B. $L = \{a^n b^n c^n : n \geq 1\}$ darstellen?

$S \rightarrow$	$A_{\uparrow p}\ B_{\uparrow q}$	*where (p = q)*
	$C_{\uparrow r}$	*where (p = r)*
$A =$	*a*	*sem p = 1 endsem*
	{a	*sem p = p + 1 endsem*
	}	

5.4 Attributierte Grammatiken (AG)

AG = (G, A, R, K)

G *Grammatik*
A *Attribute*
R *semantische Aktionene*
K *Kontextbedingungen*

Der Attributierungsprozeß:
var x, y : Integer;

declaration =	*"var" idList$_{\downarrow t}$ ":" type$_{\uparrow t1}$ ";"*	*sem t := t1 endsem*
idList$_{\downarrow t}$ =	*id$_{\uparrow name}$*	*sem NewId(↓name ↓t) endsem*
	| idList$_{\downarrow t}$ "," id$_{\uparrow name}$	*sem NewId(↓name ↓t) endsem*

- Aufstellen der nicht attributierten Grammatik
- Aufstellen des Syntaxbaumes
- Hinzufügen der semantischen Aktionen (sowohl in Syntaxbaum als auch Grammatik)

5.5 L-attributiere Grammatiken

Eine Grammatik heißt L-attributiert, wenn für jede Regel $A \rightarrow X_1 \ldots X_n$ gilt:
Ein Eingangsattribut von X_i hängt nur von den Eingangsattributen von A und den Ausgangsattributen der X_1 bis X_{i-1} ab.

6 Implementierung des Semantikanschlusses

6.1 Beim rekursiven Abstieg

- Attribute von Nonterminalsymbolen ⇒ Parameter der synt. Routine
- Attribute von Terminalsymbolen ⇒ liefert lexikalischer Analysator
- Attribute von anderen Terminalklassen ⇒ gibt es nicht
- Semantische Aktion ⇒ Codestücke
- Kontextbedingungen ⇒ wie semantische Aktionen

```
Ident(↑name)
        param name : Array[1 : 8] of Char;
        local i : Integer;
begin   -- sy und value enthält laufende Eingabe
        name(1 : 8) := " ";
        name(1) := value;
        i := 1;
        NewSym(↑sy ↑value);
        while (sy = letter) | (sy = digit) do
                i := i + 1;
                if (i < 8) then
                        name(i) := value;
                end;
                NewSym(↑sy ↑value);
        end;
        if (i > 8) then SemError end; -- Name zu lang
end Ident;
```

Ergebnis:
+ flexibel
+ einfach
- Code für Syntax und Semantik verwischt

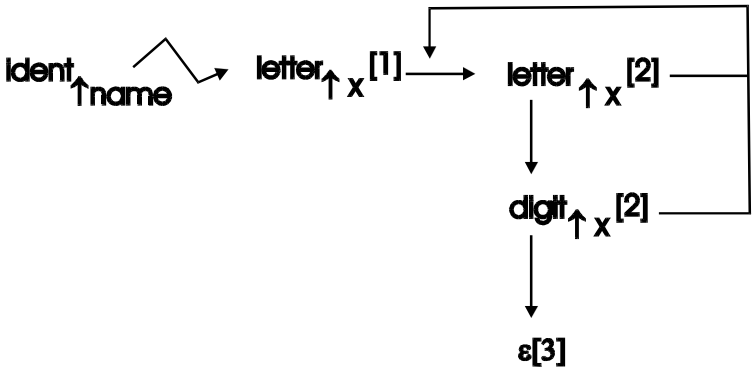
Semantikgesteuerte Syntaxanalyse

```
statement =   assignment
            | procedurecall
            | ...

assignment    = ident [ "[" explist "]" ] ":=" expr.
procedurecall = ident [ "(" exprlist ")" ].
```

```
Statement
begin
        case sy of
                ident:  if IsVariable(↓sy) then
                                assignment
                        elseif IsProcCall(↓sy) then
                                procedurecall
                        else
                                Error
                        end;
                | ...
                ...
        end;
end Statement
```

6.2 Bei der tabellengesteuerten Analyse



[...] semantische Aktionsnummer

```
[1]:    name(1 : 8) := " ";
        name(1) := x;
        i := 1;
[2]:    i := i + 1;
        if (i < 8) then name(i) := x; end;
[3]:    if (i > 8) then SemErr end;
```

```
type Graphnode = Record
        kind, val, lp, rp : ...
        sem : Integer;
end;
```

```
Parse;
begin
        ...
        if success then
                if (graph(loc).sem > 0) then
                        Semant(↓graph(loc).sem)
                end;
                if (graph(loc).kind = terminal) then
                        NewSym(↑sy ↑value)
                end;
                loc := graph(loc).rp;
        else
                loc := graph(loc).lp;
        end;
        ...
end Parse;


Semant(↓u);
        case u of
                1:      name(1 : 8) := " ";
                        ...
                | 2:    i := i + 1;
                        ...
                | ...
        end;
end Semant;
```

Semantische Aktionen können nur **nach dem Erkennen eines Symbols** ausgeführt werden. Somit ist folgende Grammatik nicht zulässig:

```
A =     sem Aktion1 endsem
        B
        |       sem Aktion2 endsem
        C.
```

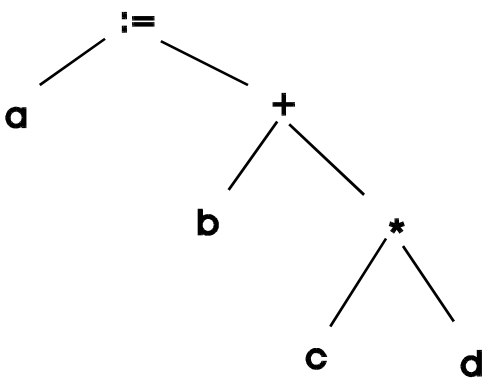
7 Zwischensprachen, abstrakte Maschinen und Interpretation

7.1 Zwischensprachen

Baumdarstellung

konkrete Syntax wird nicht verwendet, weil sie sehr viele Knotn besitzt, die keine Verzweigungen darstellen. Daher abstrakte Syntax:

*a := b + c * d*



Datenstruktur

type Nodekind = (varn, contn, plusn, ..., assignn, forn, writen, returnn);

```
Treeptr = Point to Treenode;
Treenode = Record
        kind : Nodekind;
        val : Integer;
        lp, rp : Treeptr;
end;
```

Programme sind lineare Strukturen ⇒

Linearisierte Darstellung:

- Präfix-Darstellung: := a + b * c d
- Postfix-Datstellung: a b c d * + :=
- Infix-Darstellung: braucht Klammern!

Bedeutende Zwischensprachen:

- O-Code für BCPL (Basic Compile Programming Language): Postfix-Notation
- P-Code für Pascal: erster Pascal Compiler von Wirth
- M-Code für Modula-2
- Diana für Ada: Baumstruktur

7.2 Abstrakte Maschinen

7.3 Die Abbildung der Quellsprache auf die Zielsprache

7.4 Die MiniModula Maschine

8 Die Übersetzung von Ablaufstrukturen

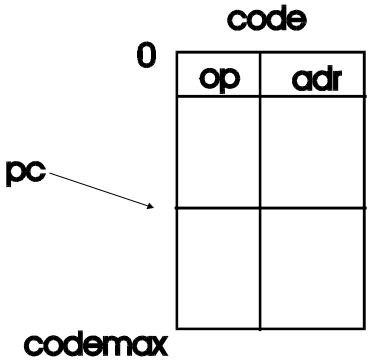
8.1 Vorbemerkungen

Gliederung der Übersetzungsvorgänge

Quellprogramm besteht aus

- **Deklaration:** keine Codeerzeugung, Speicherplatzvergabe, Speichern der Typinformation in Symbolliste
- **Anweisungen:** keine Adressierung von Daten, Code für Ablaufstrukturen, Code für Prozeduraufruf, Code für Datentransport
- **Ausdrücke** (keine Sprünge): Code für Adressieren und Laden der Operanden und Speichern der Resultate, Code für die Ausführung der Operationen in richtiger Reihenfolge, Code für den Aufruf von Funktionsprozeduren
- **Prozeduren** (Deklarationen und Aufruf): Gültigkeitsbereiche von Namen und Adressierung, Aufruf, Rücksprung und Parameterübergabe

Speicherung der erzeugten Befehle



op: *Operationscode:* *-- LV, LA, ... (1 Byte)*
adr: *Adresse* *-- Integer*
pc: *Adresse* *-- Integer*

```
code: Array[0 ... codemax] of Record
          op : Operationscode;
          adr : Adresse;
      end;
```

Zugriffsprozeduren:

Emit(↓op ↓adr);

```
        global pc;
begin
        code[pc].op := op;
        code[pc].adr := adr;
        pc := pc + 1;           -- Implementationsabhängig!
end Emit;
```

```
Fixup(↓n ↓m);               -- um bei Sprungbefehlen im Nachhinein die Adresse eintragen
                            -- zu können
begin
        code[n].adr := m;
end Fixup;
```

Einteilung der Ablaufstrukturen

- IF, CASE
- WHILE, REPEAT, FOR
- LOOP, EXIT
- GOTO L
- IF expr GOTO L

8.2 Marken und Sprunganweisungen

```
m:      x := x + 1;         -- definierendes Auftreten der Marke m
goto m                      -- angewandte Auftreten der Marke m
$1, $2, ... --- vom Compiler erzeugte Marken
```

Arithmetisierung:

labnum: Integer; -- Markennummer

Markenliste:

labnum	*adr*
1	
...	
n	

labnum	*adr*	
m	*≠ 0*	-- nach definierendem Auftreten von m
m	0	-- nach angewandten und vor definierenden Auftreten von m

8.3 IF-Anweisungen

```
IF expr THEN StatSeq END
        | Code für expr
        | ...
FJMP $1                                          -- False Jump
        | Code für StatSeq
        | ...
$1:     ...
```

```
stat =                  -- global pc, local jumpadr : Adresse
        IF expr         sem jmpadr := pc; Emit(↓FJMP ↓0); endsem
        THEN StatSeq    sem Fixup(↓jmpadr ↓pc); endsem
        END
```

8.4 CASE-Anweisung

8.5 Schleifenanweisungen

WHILE-Schleife

```
stat =                  --global pc, local startadr, jmpadr : Adresse
        WHILE           sem startadr := pc; endsem
        expr            sem jmpadr := pc; Emit(↓FJMP ↓0); endsem
        DO StatSeq      sem Emit(↓JMP ↓startadr); Fixump(↓jmpadr ↓pc); endsem
        END
```

FOR-Schleife:
FOR v := expr1 TO expr2 DO StatSeq END

```
        v := expr1
$1:     if v > expr2 then goto $2 end
        statseq
        v := v + 1
        goto $1
$2:     ...
```

Probleme:
- beim Verlassen der Schleife ist *v* um 1 größer als *expr2*
- *expr2* kann in *StatSeq* geändert werden
- *expr2* muß wiederholt ausgewertet werden
- was passiert bei einem Überlauf?

```
        v := expr1
        h := expr2
        if v > h then goto $3 else goto $2 end
$1:     v := v + 1
$2:     statseq
        if v < h then goto $1 end
$3:     ...
```

9 Datenarten und Datentypen

9.1 Objekte und ihre Beschreibung im Compiler

- **Objektarten:**
  - Konstanten
  - Variablen
  - Prozeduren
  - (Module, Recordfelder)
- **Objekttypen:**
  - INTEGER
  - REAL
  - CHAR
  - BOOLEAN
  - ARRAY OF
  - (SET OF ..., FILE OF ...)

Attributarten:

Objektart	Allgemeine Attribute	Individuelle Attribute
Variable	Name Typ	Adresse (Blocktiefe, Parameterkennzahlen)
Konstante	(Name) Typ	Wert
Prozedur	Name (Typ)	Startadresse, Parameterzeiger

MiniModula: (Objektdeskriptor)

```
op :    Record
        kind: (conkind, varkiond, parkind, vregkind, aregkind, prockind, forwkind);
        typ : (simint, simchar, simbool, arrint, arrchr, arrbool, proctype);
        size : Integer;
        valadr : Integer;       -- value oder Adresse, je nach kind
        level : Integer;                -- Schachtelungstiefe
end;
```

vregkind ... Wert des Operanden zur Laufzeit am Kellerende
aregkind ... Adresse des Operanden zur Laufzeit am Kellerende

9.2 Auswirkung der Datentypen auf Opertionen (s. Skript)

stat = $ref_{\uparrow op1}$ ":=" $expr_{\uparrow op2}$

where (op1.typ = op2.typ) & (op1.size = op2.size)
sem Assign(↓op1 ↓op2); endsem
-- eigentlich 16 Fallunterscheidungen, die Assign abnimmt

9.3 Selektion

(= Adreßberechnung von zusammengesetzten Typen und Arrays)

factor = desig | c | "(" expr ")".
desig = ident {"." ident | "[" exprlist "]"}.
exprlist = expr {"," expr}.

```
a[i] := 1;          -- Adresse in wertempfangender (linker) Position
LA 0, Adr(a[i])
LIT 1
STO

x := a[i];          -- Adresse in wertliefernder (rechter) Position
LA 0, Adr(x)
LV 0, Adr(a[i])
STO

x := a[i];
LA 0, Adr(x)
LA 0, Adr(a[i])
LI
STO
```

Eindimensionale Felder:

Gegeben:
var a : Array(u : o) of Elementtype
a[i] in Anweisung

Gesucht:
Code zum Laden von *Adr(a[i])*

Adr(a)
u
o
e

Elementlänge

Adr*(a[i]) = f(Adr(a), u, o, e, i)* **... Speicherabbildungsfunktion**

*Adr(a[i]) = Adr(a) + (i - u)*e = Adr(a) - u*e + i*e = Adr(a[0]) + i*e*

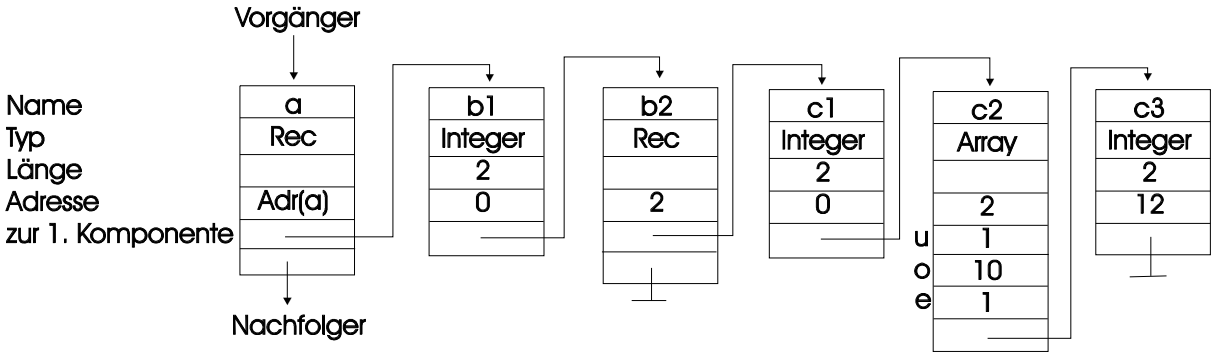
Adr(a[0]) ... virtuelle Feldadresse; zur Compilezeit bekannt (= Basisadresse)

*i*e* ist allerdings erst zur Laufzeit bekannt (= Offset), außer man hat ein konstantes *i*, wodurch die gesamte Adressberechnung bereits zur Compilezeit erfolgen kann.

Verbunde

	Startadresse
var a : Record	*Adr(a)*

```
        b1 : Integer;                                   Adr(a)
        b2 : Record                                     Adr(a) + 2
                c1 : Integer;                           Adr(a) + 2
                c2 : Array(1 : 10) of Char;             Adr(a) + 2 + 2
                c3 : Integer;                           Adr(a) + 2 + 2 + 10
        end;
end;
```



x := a.b2.c2[i];

Adr(a) aus Symbolliste
Adr(a.b2) = Adr(a) + 2
Adr(a.b2.c2) = Adr(a) + 2 + 2
*Adr(a.b2.c2[i]) = Adr(a) + 2 + 2 -1 + i*1 = Adr(a) + 3 - i*

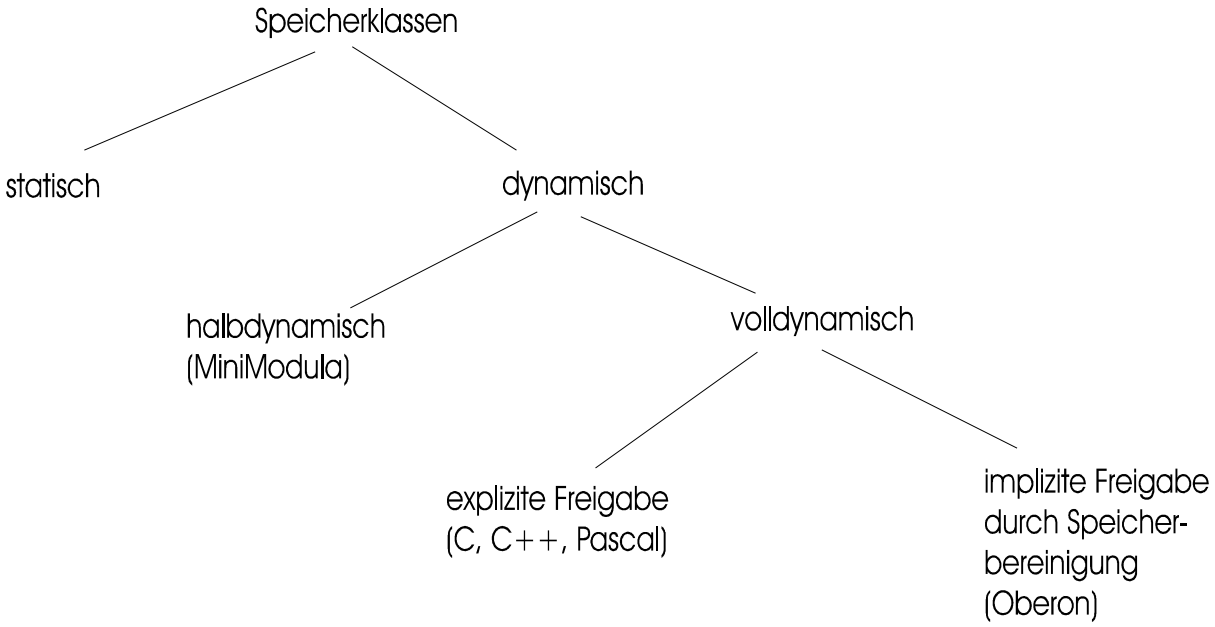
10 Dynamische Speicherverteilung

- zur Laufzeit
- beim Eintritt in eine Prozedur wird ein Aktivierungssatz (AS) angelegt, beim Verlassen derselben freigegeben

NEU:
- Blockstruktur
- interne Prozeduren
- explizite Reservierung und Freigabe von Datenspeicherplatz zur Laufzeit

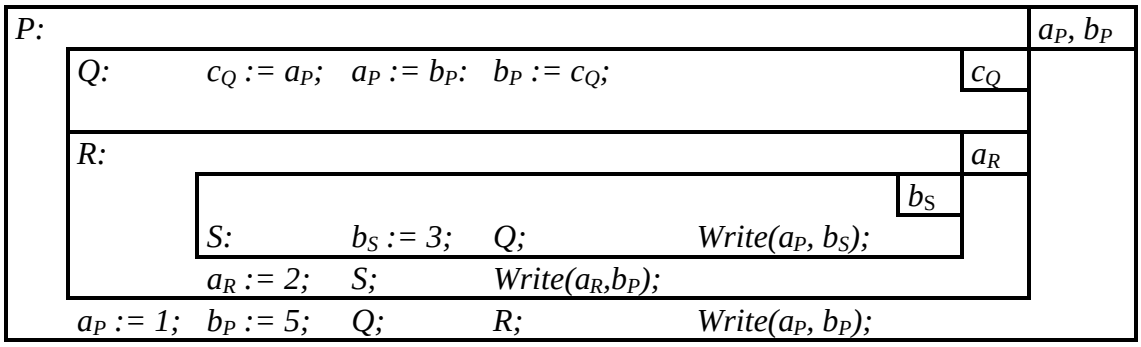
Nachteil:
- Laufzeit geringfügig langsamer
- Compiler schwerer zu bauen

Speicherklassen



10.1 Die Semantik blockstrukturierter Programme

(Buch von Gries)




```
P:    aP = 1; bP = 5;
      Q;...
      aP = 5; bP = 1;
      R:    aR = 2;
            S:    bS = 3;
                  Q; ...
                  aP = 1; bP = 5;
                  Write(2, 3);    -- aR, bS
            Write(2,5);           -- aR, bP
      Write(1,5);                 -- aP, bP
```

Lebensdauer (LD) und Gültigkeitsbereich (GB)

Speicherklasse	**Lebensdauer**
statisch	gesamte Ausführungszeit des Programms
automatisch	Ausführungsdauer des die Deklaration enthaltenen Blocks
unbeschränkt	von der Erzeugung (*NEW*) bis zum Programmende
kontrolliert	von der Erzeugung (*NEW*) bis zur Freigabe (*DISPOSE*)

Gültigkeitsart	**Deklarationsstelle**
lokal	im Block
...	...

Parameterübergabe

```
Main
        local m, n : Integer;
2       P(↕x ↕y) : Integer;
3               x := x + 1;
4               m := m + y;
5               return (x + y);
        end P
7       m := 1;
8       n := P(↕m ↕m + 3);
9       Write(m, n);
end Main;
```

Call by value
formaler (lokaler) Parameter wird bei Prozeduraufruf durch den aktuellen Parameter ersetzt.

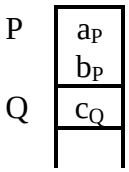
Call by reference
nicht der Wert der Variablen, sondern die Adresse wird übergeben

Call by name
der formale Parameter wird wörtlich durch den aktuellen ersetzt (name = Ausdruck des aktuellen Parameters) im ALGOL 60 implementiert (anfänglich)

	value	***reference***	***name***
7:	*m := 1*	*m := 1*	*m := 1*
2:	*x := m = 1;*	*(x = m)*	*("x" = "m")*
	y := m + 3 = 4	*(y = 4)*	*("y" = "m + 3")*
3:	*x := x + 1 = 2*	*m := m + 1 = 2*	*m := m + 1 = 2*
4:	*m := m + y = 5*	*m := m + 4 = 6*	*m := m + m + 3 = 7*
5,8:	*n := x + y = 6*	*n := m + 4 = 10*	*n := m + m + 3 = 17*
9:	*Write(5, 6)*	*Write(6, 10)*	*Write(7, 17)*

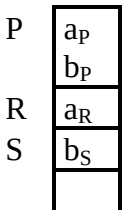
10.2 Übersetzung blockstrukturierter Sprachen

Symbollistenorganisation (zur Übersetzungszeit)



wird von unten nach oben durchlaufen, wenn gesucht wird; schreiben von oben nach unten

Nach dem Verlassen von Q wird c_Q gelöscht!



Durch die Kellerarchitektur ist das Auffinden des richtigen Sichtbarkeitsbereichs einfach.

Datenspeicherorganisation (zur Laufzeit; dynamisch, kellerartig)

sl *static link*
pc *Befehlszählerstand bei Blockeintritt*
ra *Rücksprungadresse*
dl *dynamic link*
db *data base*

Zweidimensionale Adressierung
Datenadresse zur Laufzeit einer Variablen = AS = Adresse + Distanz im AS

Datenadresse = AS - Adresse + Distanz im AS

AS - Adresse: erst zur Laufzeit bekannt
Distanz im AS: zur Übersetzungszeit bekannt

1. Möglichkeit:
mit Basisregister und statischer Verweiskette

2. Möglichkeit:
Basisregister für lokale und globale Größen

c := a

```
MOVE 0(db), AC      -- AC ist Register
MOVE 12(AC), AC     -- AC enthält a
MOVE AC, 12(db)     -- c := a
```

Adr(x) = (Blocktiefendifferenz n, Distanz d)

```
MOVE 0(db), AC      -- AC = Adresse des 1. statischen Vorgängers
MOVE 0(AC), AC
...                 (n-1) mal
MOVE 0(AC), AC
MOVE d(AC), AC
```

Prozedur Ein- und Austritt
Ansprung:
- AS für den Gerufenen reservieren
- Parameter übergeben
- Gerufenen anspringen

Rücksprung:
- AS freigeben
- zum Rufer zurückspringen (eventuell Funktionswert zurückgeben)

```
Call Q:
        Reserviere Speicherplatz für 3 Objekte;
        Fülle statischen Verweis;
        Fülle dynamischen Verweis;
        Stelle Basisregister auf AS von Q;
        Fülle Rücksprungadresse;
        Springe Q an;
end Call;
Start von Q:
        Reserviere Speicherplatz für den eigenen AS;
```

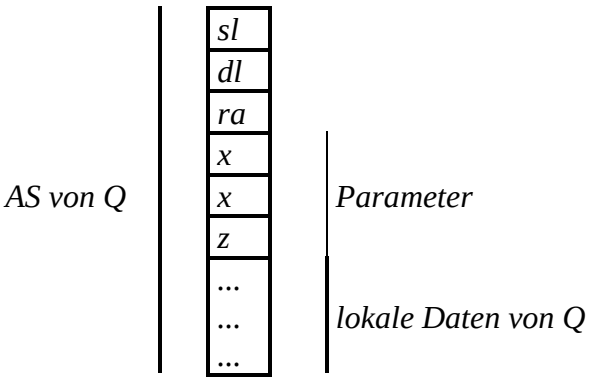
Return:
Gibt den eigenen AS frei;
Basisregister := dynamischer Verweis;
Springe zum Rufer;
end Return;

n := BT(Rufer) - BT(Gerufener) + 1 *-- Anzahl der Schritte für Durchlaufen der statischen Verweiskette*

Prozedurparameter und -variablen

Parameterübergabe

Call Q(x, y, z);



Reserviere Platz für sl, dl, ra;	*Prepare Stack*
Fülle sl, dl;	
Push(x);	
Push(y);	
Push(z);	
Verstelle db;	*Call*
Fülle ra;	
Springe zum Gerufenen;	

Offene Felder

Proc A(var b : Array of Integer);

sl
dl
ra
Adr(b)
Länge von b

f : Array (u : o) of Integer;

sl
dl
ra
u
o
f(u:o) of Integer

11 Index

A

Abbildung der Quellsprache auf die Zielsprache ..23
Abspeichern der Grammatik-Tabelle 13
abstrakte Maschinen..... 22; 23
Algorithmische Strukturen 7
allgemeinen Nachfolgersymbole..... 12
Analysealgorithmus..... 15
Attributarten..... 27
Attribute..... 16
Attributierte Grammatiken (AG)..... 18
Attributierte Grammtik mit semantischen Aktionen bei EBNF 17
Attributierungsprozeß 18
Aufbau der Schlüsselwort- und Namenliste..... 8
Aufgaben der lexikalischen Analyse..... 6
Ausgangsattribute 17
Auswirkung der Datentypen auf Opertionen (s. Skript) 27

B

Besondere Probleme 9

C

Call by name 32
Call by reference 32
CASE-Anweisung..... 25
Codegenerierung 5

D

Datenarten und Datentypen..... 26
Datenspeicherorganisation 33
derived attributes 17
Die MiniModula Maschine 23
Dynamische Grobstruktur 5
Dynamische Namenliste..... 9
Dynamische Speicherverteilung..... 30

E

Eigene Schlüsselwortliste..... 8
Eindimensionale Felder 28
Eingangsattribute 17
Einleitung und Übersicht 3
Einpaß-Compiler..... 5
Emit 24

F

Fangsymbole..... 12
Fehler erkennen und anzeigen..... 11
Fehlerbehandlung..... 11; 15
Fixup..... 24
Folgefehler unterdrücken 12
Formale Beschreibung von Übersetzungsprozessen16
FOR-Schleife 26

G

Gliederung der Übersetzungsvorgänge 23
Graphnode 14; 20
Grundbegriffe 3
Gültigkeitsbereich..... 31

I

Ident..... 19
IF-Anweisungen..... 25
Implementierung des Semantikanschlusses 19
inherited attributes 17
Interpretation 22

K

Kontextbedingungen..... 17

L

L-attributiere Grammatiken 18
Lebensdauer..... 31
Lexan 11
Lexikalische Analyse..... 3; 6
Literaturhinweis..... 6

M

Marken und Sprunganweisungen..... 24
Markenliste 25
Match..... 15
Mehrpaß-Compiler 5

N

Namen und Schlüssel gemeinsam..... 8
Namenlistbehandlung 8
Namenliste..... 5
Nodekind 22

O

Objektarten 26
Objekte und ihre Beschreibung im Compiler..... 26
Objekttypen 26
Offene Felder..... 36
Ohne Fehlerbehandlung..... 10
Optimierung..... 4

P

Parameterübergabe 32; 35
Parse 21
Prozedur Ein- und Austritt..... 34

R

Recover..... 12
Rekursiven Abstiegs 10
Repräsentation des Topdown-Graphen 14
Root 15

S

Scanner-Generatoren....9
Schleifenanweisungen....25
Schlüsselwörter in Hash-Code speichern....8
Selektion....28
Semantik blockstrukturierter Programme....31
Semantikanschluß bei der tabellengesteuerten Analyse....20
Semantikanschluß beim rekursiven Abstieg....19
Semantische Aktionen....16
Semantische Analyse....4
Speicherklassen....30
speziellen Fangsymbole....12
Sprachenvielfalt....3
Statische Grobstruktur von Compilern....3
Statische Namenliste allein....8
Statische Namenliste und Deskriptorenliste mit n Namen....8
Suchverfahren in dynamischer Namenliste....9
Symbolarten....6
Symbolliste....5
Symbollistenorganisation....33
Syntaxanalyse....4
Syntaxanalyse I (Verfahren des rekursiven Abstiegs10
Syntaxanalyse II (Tabellengesteuerte Analyse)....13
synthesized attributes....17

T

Tabellengesteuerte Analyse....13
Teilaufgaben....7
Topdown-Graph....13

U

Übersetzung blockstrukturierter Sprachen....33
Übersetzung von Ablaufstrukturen....23

V

Varianten....3
Verbunde....29

W

WHILE-Schleife....25
Wiederaufsetzen....12

Z

Zugriffsprozeduren....24
Zweidimensionale Adressierung....34
Zwischensprache....4
Zwischensprachen....22