

Codegenerierung

- Der Code wird nacheinander für jede Prozedur gesondert erzeugt.
- Code wird innerhalb von Block für statement generiert.
- Ist der Code für eine Prozedur generiert, kann er in die Ausgabe-datei geschrieben werden.
- Die Codegenerierung erfolgt durch die Routinen, die vom Parser aufgerufen werden, wenn Bögen akzeptiert worden sind.
- Vor der eigentlichen Codegenerierung einer jeden Prozedur ist der Befehl `entryProc (CodeLen, ProcIdx, VarLen)` zu generieren, dies erfolgt in dem Nil-Bogen vor statement. CodeLen bleibt dabei zunächst 0, VarLen ist dem SpzzVar zu entnehmen.
- Der vorgestellte Compiler generiert Code für eine virtuelle Maschine. Er wird auf den nächsten Seite vorgestellt.

```

puValVrLocl,/*00 (short Displ) [Kellern Wert lokale Variable] */
puValVrMain,/*01 (short Displ) [Kellern Wert Main Variable] */
puValVrGlob,/*02 (short Displ,short Proc) [Kellern Wert globale Variable] */
puAdrVrLocl,/*03 (short Displ) [Kellern Adresse lokale Variable] */
puAdrVrMain,/*04 (short Displ) [Kellern Adresse Main Variable] */
puAdrVrGlob,/*05 (short Displ,short Proc) [Kellern Adresse globale Variable] */
puConst ,/*06 (short Index) [Kellern einer Konstanten] */
storeVal ,/*07 () [Speichern Wert -> Adresse, beides aus Keller] */
putVal ,/*08 () [Ausgabe eines Wertes aus Keller nach stdout] */
getVal ,/*09 () [Eingabe eines Wertes von stdin -> Addr. im Keller ] */
/*--- arithmetische Befehle ---*/
vzMinus ,/*0A () [Vorzeichen -] */
odd ,/*0B () [ungerade -> 0/1] */
/*--- binaere Operatoren kellern 2 Operanden aus und das Ergebnis ein ----*/
OpAdd ,/*0C () [Addition] */
OpSub ,/*0D () [Subtraktion ] */
OpMult ,/*0E () [Multiplikation ] */
OpDiv ,/*0F () [Division ] */
cmpEQ ,/*10 () [Vergleich = -> 0/1] */
cmpNE ,/*11 () [Vergleich # -> 0/1] */
cmpLT ,/*12 () [Vergleich < -> 0/1] */
cmpGT ,/*13 () [Vergleich > -> 0/1] */
cmpLE ,/*14 () [Vergleich <=-> 0/1] */
cmpGE ,/*15 () [Vergleich >=-> 0/1] */
/*--- Sprungbefehle ---*/
call ,/*16 (short ProzNr) [Prozeduraufruf] */
retProc ,/*17 () [Ruecksprung] */
jmp ,/*18 (short RelAdr) [SPZZ innerhalb der Funktion] */
jnot ,/*19 (short RelAdr) [SPZZ innerhalb der Funkt.,Beding.aus Keller] */
entryProc ,/*1A (short lenCode,short ProcIdx,short lenVar) */
putStrg ,/*1B (char[]) (Zusaetzlich, gehoert nicht zu PL/0) */
EndOfCode /*1C */

```

Funktionen zur Codegenerierung

int code(char OpCode, ...);

Befehle haben 0, 1, 2 oder 3 Parameter.

Parameter sind alle vom Typ short (2 Byte).

Operationscodes sind 1 Byte groß.

Die Parameter werden in der Byteorder little endian (intel) gespeichert.

Soll der Compiler portabel sein (und das sollte er), muss eine Funktion zum Schreiben der Parameter gebaut werden.

Achtung in java werden int-werte im bigendian Format geschrieben!

Funktionen zum Schreiben der Befehlsparameter in der geforderten Byteorder

```
/*-----*/  
void wr2ToCode(short x)  
{  
    *pCode++=(unsigned char) (x & 0xff);  
    *pCode++=(unsigned char) (x >> 8);  
}
```

Schreibt am aktuellen
Programmcounter

```
void wr2ToCodeAtP(short x, char*pD)  
{  
    * pD      =(unsigned char) (x & 0xff);  
    * (pD+1)=(unsigned char) (x >> 8);  
}
```

Schreibt an der
Übergebenen Stelle

```
int code(tCode Code,...)
```

```
{
```

```
    va_list ap;
```

```
    short sarg;
```

```
    if (pCode-vCode+MAX_LEN_OF_CODE>=LenCode)
```

```
    // Ueberwachung des Zwischencodegenerierungspuffers
```

```
    {
```

```
        char* xCode=realloc(vCode, (LenCode+=1024));
```

```
        if (xCode==NULL) Error(ENoMem);
```

```
        pCode=xCode+(pCode-vCode);
```

```
        vCode=xCode;
```

```
    }
```

```
    *pCode++=(char)Code;
```

```
    va_start(ap,Code);
```

Schreibt Befehl
Mit 0, 1, 2 oder 3
Parametern in den
Codeausgabepuffer

```

switch (Code)
{
    /* Befehle mit 3 Parametern */
    case entryProc:
        sarg=va_arg(ap,int);
        wr2ToCode(sarg);
    /* Befehle mit 2 Parametern */
    case puValVrGlob:
    case puAdrVrGlob:
        sarg=va_arg(ap,int);
        wr2ToCode(sarg);
    /* Befehle mit 1 Parameter */
    case puValVrMain:
    case puAdrVrMain:
    case puValVrLocl:
    case puAdrVrLocl:
    case puConst:
    case jmp :
    case jnot:
    case call:
        sarg=va_arg(ap,int);
        wr2ToCode(sarg);
        break;
    /* ohne Parameter */
    default      : break;
}
va_end  (ap);
return OK;
}

```

Casezweige ohne break!

Variablen zur Codegenerierung

- Prozedurnummer der aktuellen Prozedur
- Codeausgabebereich (mit Füllstand durch Pointer oder Index)
- Konstantenblock
- Datenstrukturen der Namensliste

Anmerkung zu Push-Befehlen

- Es gibt zwei Gruppen von Push-Befehlen:
 - PushVal: Kellert den Wert einer Variablen im Stack
 - PushAdr: Kellert die Adresse einer Variablen im Stack
- Befehle beider Gruppen gibt es in drei Varianten:
 - local: Die Variable ist eine Variable der gerade aktuellen Prozedur.
 - main: Die Variable ist eine Variable des Hauptprogramms.
 - Global: Die Variable ist eine Variable einer umgebenden Prozedur. Hier ist die zusätzliche Angabe der Prozedurnummer der Prozedur, zu der die Variable gehört, notwendig.

Codegenerierungsroutinen

- Auf den nachfolgenden Seiten werden die Syntaxgraphen gezeigt und an welchen Stellen Aktionen zur Codegenerierung sinnvoll angelegt werden.
- Es wird in Stichpunkten die Funktionalität dieser Routinen dargelegt.
- Wie die Routinen zur Namensliste werden auch diese Routinen durch den Parser aufgerufen. Ihre Adresse ist in die Funktionspointer in den Graphenbeschreibungen einzutragen.

Einstieg für optimales Testen

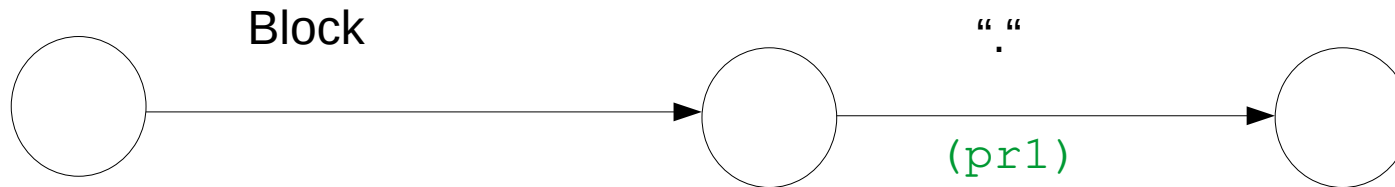
- Factor (fa1), Block (bl6, bl5), Statement (st10), Programm (pr1)
 - Nach Implementation dieser Routinen kann das Programm
! 5 .

• compiliert und mit der virtuellen Maschine getestet werden. Es sollte 5 ausgegeben.
- Das weitere Vorgehen ist ab s. 30, einschließlich geeigneter Testbeispiele und der zu generierenden Codes dargestellt.

programm

Nur eine Aktion, wenn die Compilierung bis hierhin gekommen ist.

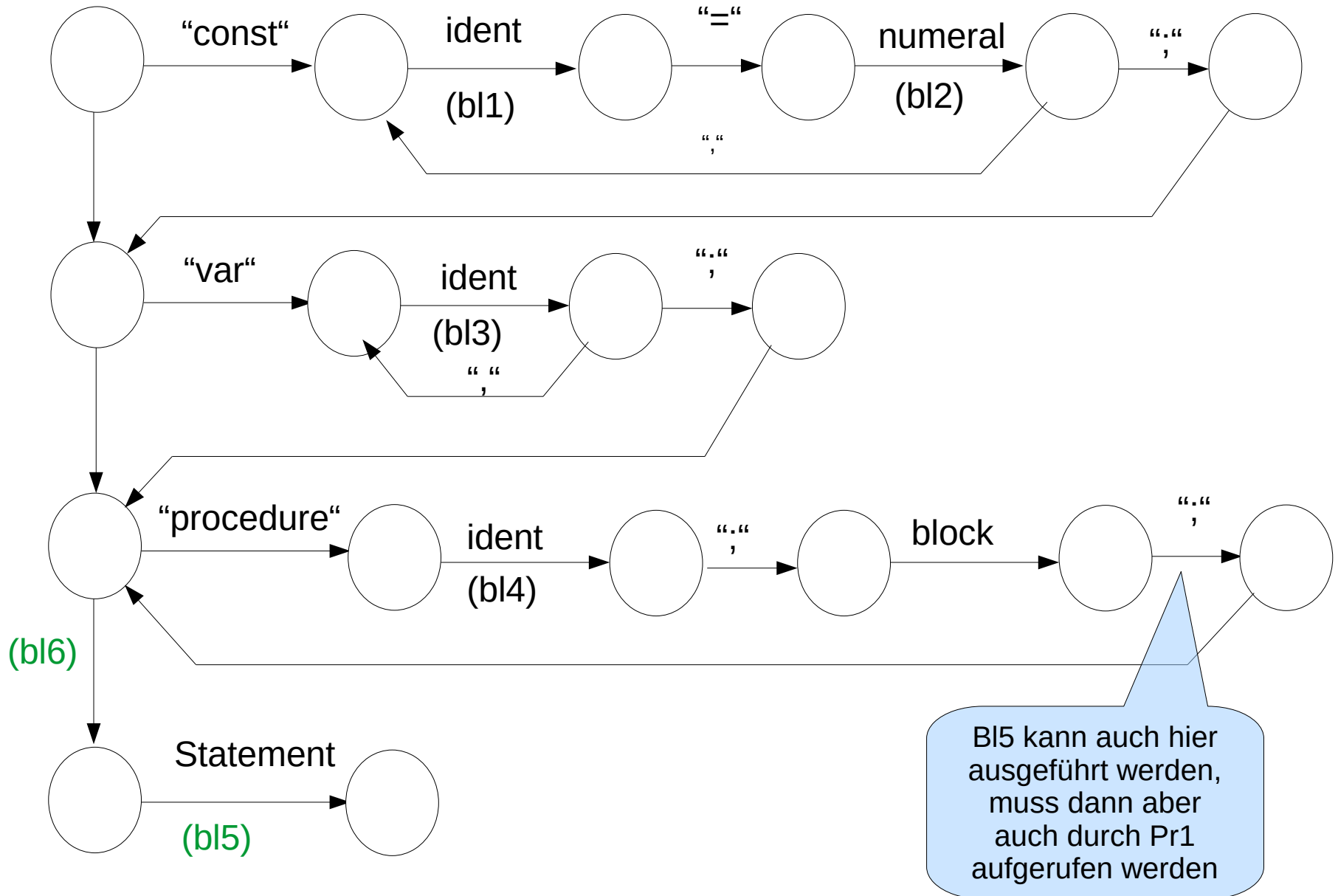
Abschlussarbeiten



pr1 (Endebehandlung):

- Aufruf von bl5 (Endebehandlung von Prozeduren), wenn bl5 bei ';' und nicht bei statement angerufen wurde
- Schreiben des Konstantenblocks in das Codefile
- Schreiben der Anzahl der Prozeduren in das Codefile am Anfang

block



bl1(Konstantenbezeichner):

lokale Suche nach dem Bezeichner

gefunden -> Fehlerbehandlung

nicht gefunden -> Bezeichner anlegen

bl2 (Konstantenwert):

Konstantenbeschreibung anlegen

Suche nach Konstante im Konstantenblock

gefunden -> Index der Konstanten eintragen in

Konstantenbeschreibung

Konstante anlegen im Konstantenblock und Index der Konstanten
eintragen in Konstantenbeschreibung

In letzten Bezeichner Zeiger auf Konstante eintragen

Diese Routinen wurden schon
Bei der Namensliste behandelt

bl3 (Variablenbezeichner):

lokale Suche nach dem Bezeichner

gefunden -> Fehlerbehandlung

nicht gefunden -> Bezeichner anlegen

Variablenbeschreibung anlegen und Pointer in Bezeichner eintragen

Relativadresse ermitteln aus SpzzVar, SpzzVar um 4 erhöhen

(Virtuelle Maschine arbeitet mit 4 Byte langen long-Werten)

bl4(Prozedurbezeichner):

lokale Suche nach dem Bezeichner

gefunden -> Fehlerbehandlung

nicht gefunden -> Bezeichner anlegen

Prozedurbeschreibung anlegen

Pointer auf Parent-Prozedur eintragen

Pointer auf Prozedurbeschreibung in letzten Bezeichner eintragen

Neue Prozedur ist jetzt aktuelle Prozedur

bl5 (Ende der Prozedurvereinbarung):

Codegenerierung: `retProc`

Codelänge in den Befehl `entryProc` als 1. Parameter nachtragen

Code aus dem Codepuffer in die Ausgabedatei schreiben (anfügen)

Namensliste mit allen Konstanten-, Variablen- und

Prozedurbeschreibungen auflösen; die Prozedur selbst muss noch erhalten bleiben

bl6(Beginn des Anweisungsteils):

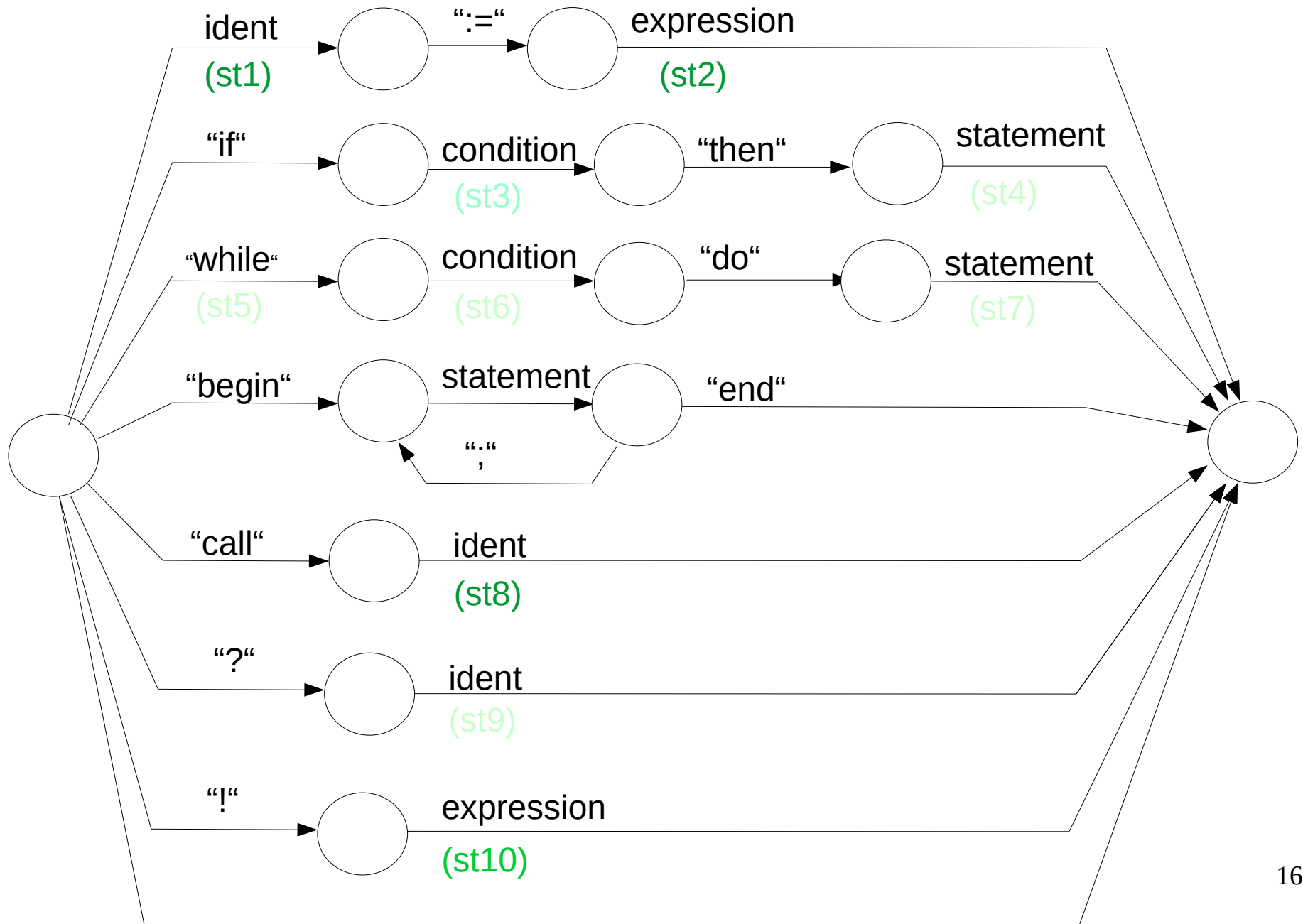
Hier muss ggf. ein Nilbogen eingefügt werden, um diese Aktion an dieser Stelle ausführen zu können!

Die Routinen werden immer nach der Akzeptanz des Bogens ausgeführt. Diese Routine muss aber vor Statement ausgeführt werden, deshalb wurde hier ein NIL-Bogen eingeführt.

Codeausgabepuffer initialisieren

Codegenerierung: entryProc mit den Parametern CodeLen, IdxProc, VarLen mit CodeLen zunächst 0, IdxProc mit pCurrProc->Idx, und VarLen aus SpzzVar der aktuellen Prozedur generieren.

statement



Zuweisung

st1(Linke Seite der Zuweisung):

Bezeichner global suchen

nicht gefunden -> Fehlerbehandlung

gefunden -> ok.

Bezeichnet der Bezeichner eine Variable?

Nein, eine Konstante oder Prozedur -> Fehlerbehandlung

ja -> ok,

Codegenerierung :

Bezeichner gefunden	Code	Parameter
Local	PushAdrVarLocal	Relativadresse
Im Hauptprogramm	PushAdrVarMain	Relativadresse
In umgebender Prozedur	PushAdrVarGlobal	Relativadresse, Prozedurnummer

st2 (Rechte Seite der Zuweisung):

Im Stack steht nun die Adresse der Zielvariable und der Wert des fertig berechneten Ausdrucks.

Codegenerierung : storeVal

st9 Eingabe:

Bezeichner global suchen:

nicht gefunden -> Fehlerbehandlung

gefunden -> ok.

Bezeichnet der Bezeichner eine Variable?

Nein, eine Konstante oder Prozedur -> Fehlerbehandlung

ja -> ok,

Codegenerierung :

Bezeichner gefunden	Code	Parameter
Local	PushAdrVarLocal	Relativadresse
Im Hauptprogramm	PushAdrVarMain	Relativadresse
In umgebender Prozedur	PushAdrVarGlobal	Relativadresse, Prozedurnummer

Codegenerierung : getVal

st10 Ausgabe:

Auszugebender Wert steht im Stack.

Codegenerierung: putVal

procedure call

st8 Prozeduraufruf:

Bezeichner global suchen

Nicht gefunden -> Fehlerbehandlung

Gefunden -> ok.

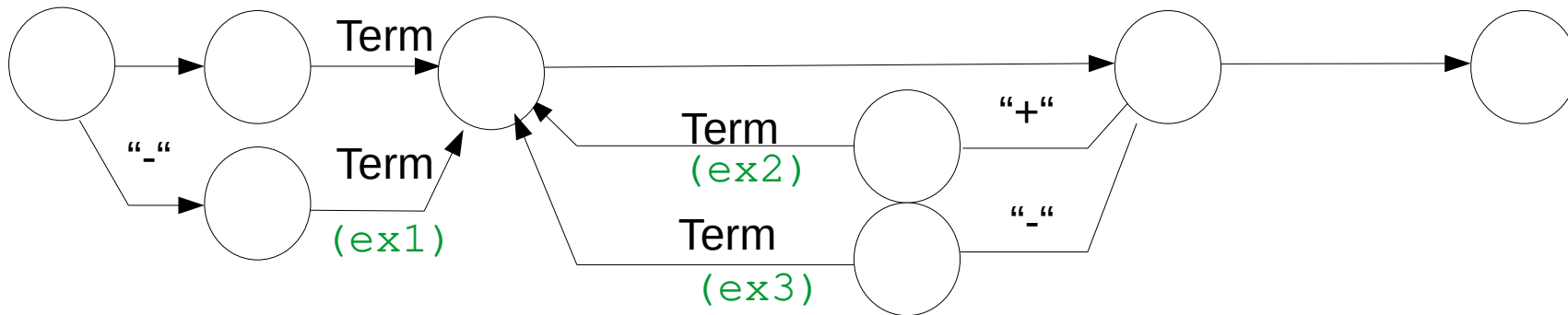
Bezeichnet der Bezeichner eine Procedure?

Nein, eine Konstante oder Prozedur -> Fehlerbehandlung

Ja -> ok

Codegenerierung call procedurenumber

expression



ex1 (negatives Vorzeichen)

- Codegenerierung `vzMinus`

ex2 (add)

- Codegenerierung `opAdd`

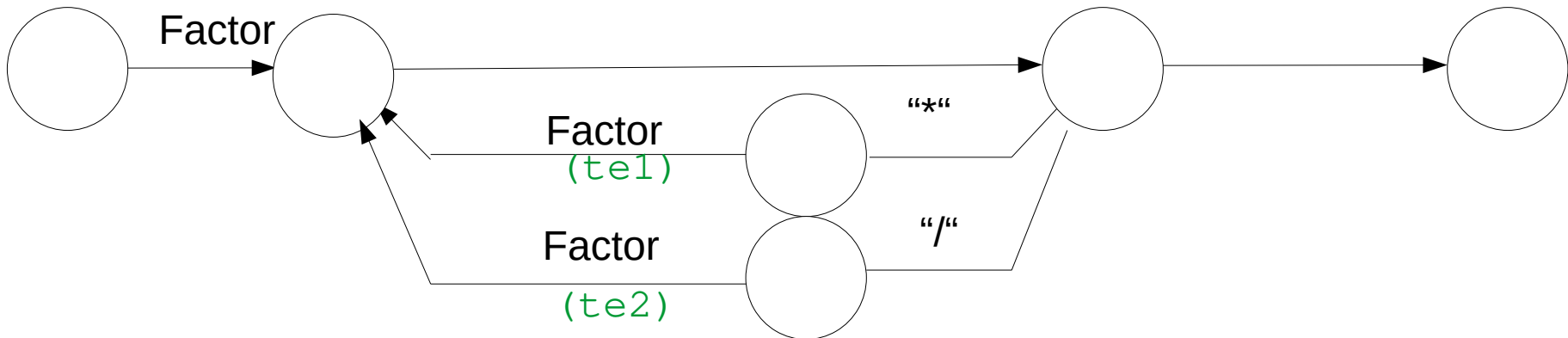
ex3 (sub)

Codegenerierung `opSub`

Anmerkung:

Es werden hier nur die additiven Operatoren generiert. Der Code zum Kellern der Operanden wurde bereits innerhalb von `Term → Faktor` erzeugt.

term



te1 (mul)

Codegenerierung `opMul`

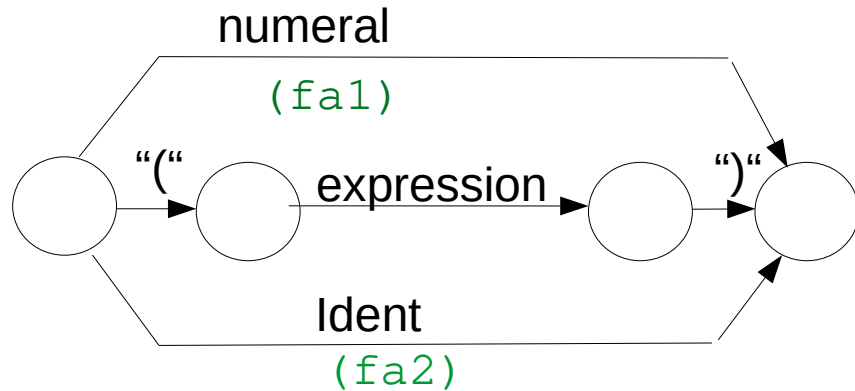
te2 (div)

Codegenerierung `opDiv`

Anmerkung:

Es werden hier nur die Multiplikativen Operatoren generiert. Der Code zum Kellern der Operanden wurde bereits innerhalb von Faktor erzeugt.

factor



fa1 (Numeral):

suchen der Konstante
ggf. anlegen der Konstanten, wenn nicht
gefunden (unbenannte Konstante)

Codegenerierung

puConst (ConstIndex)

fa2 (Ident):

Bezeichner global suchen

nicht gefunden -> Fehlerbehandlung

gefunden -> ok.

Bezeichnet der Name eine Variable oder Konstante?

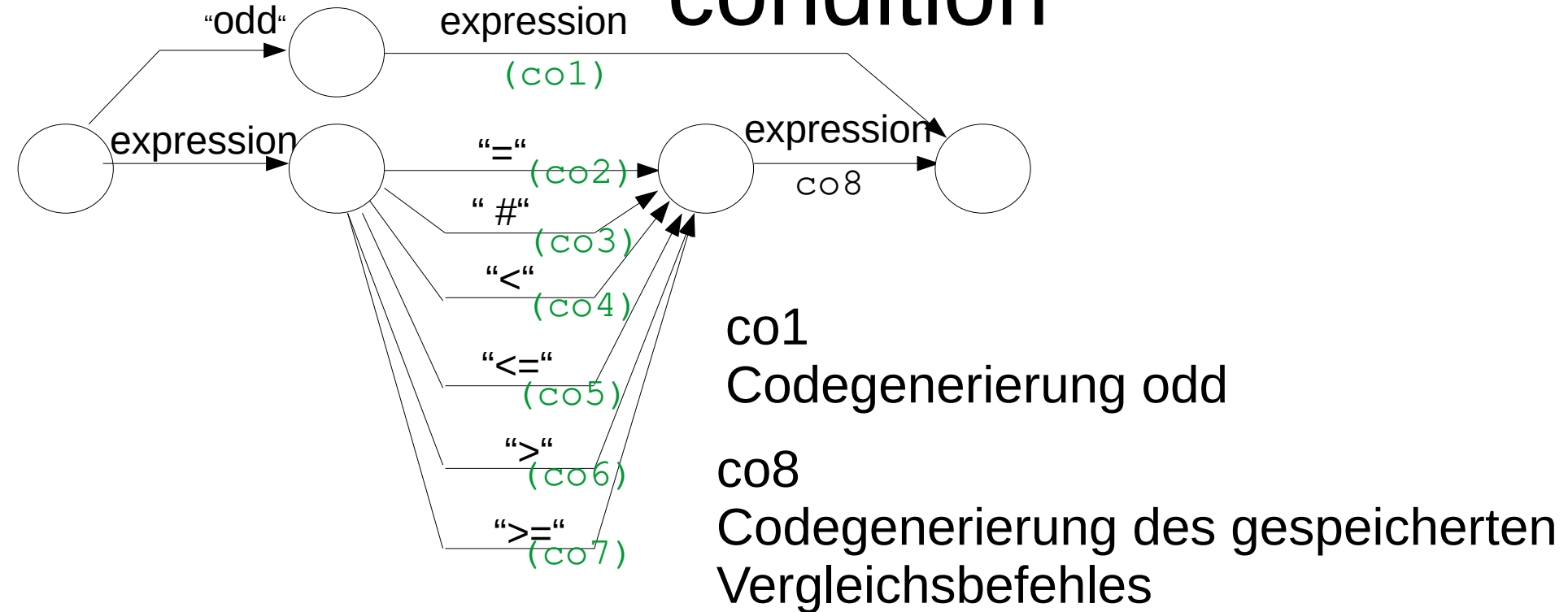
Nein, eine Prozedur -> Fehlerbehandlung

ja -> ok,

Codegenerierung :

Bezeichner gefunden	Code	Parameter
LocalVar	puValVrLocl	displ
Main Var	puValVrMain	displ
Global Var	puValVrGlob	displ, ProcedureNr
Konstante	puConst	index

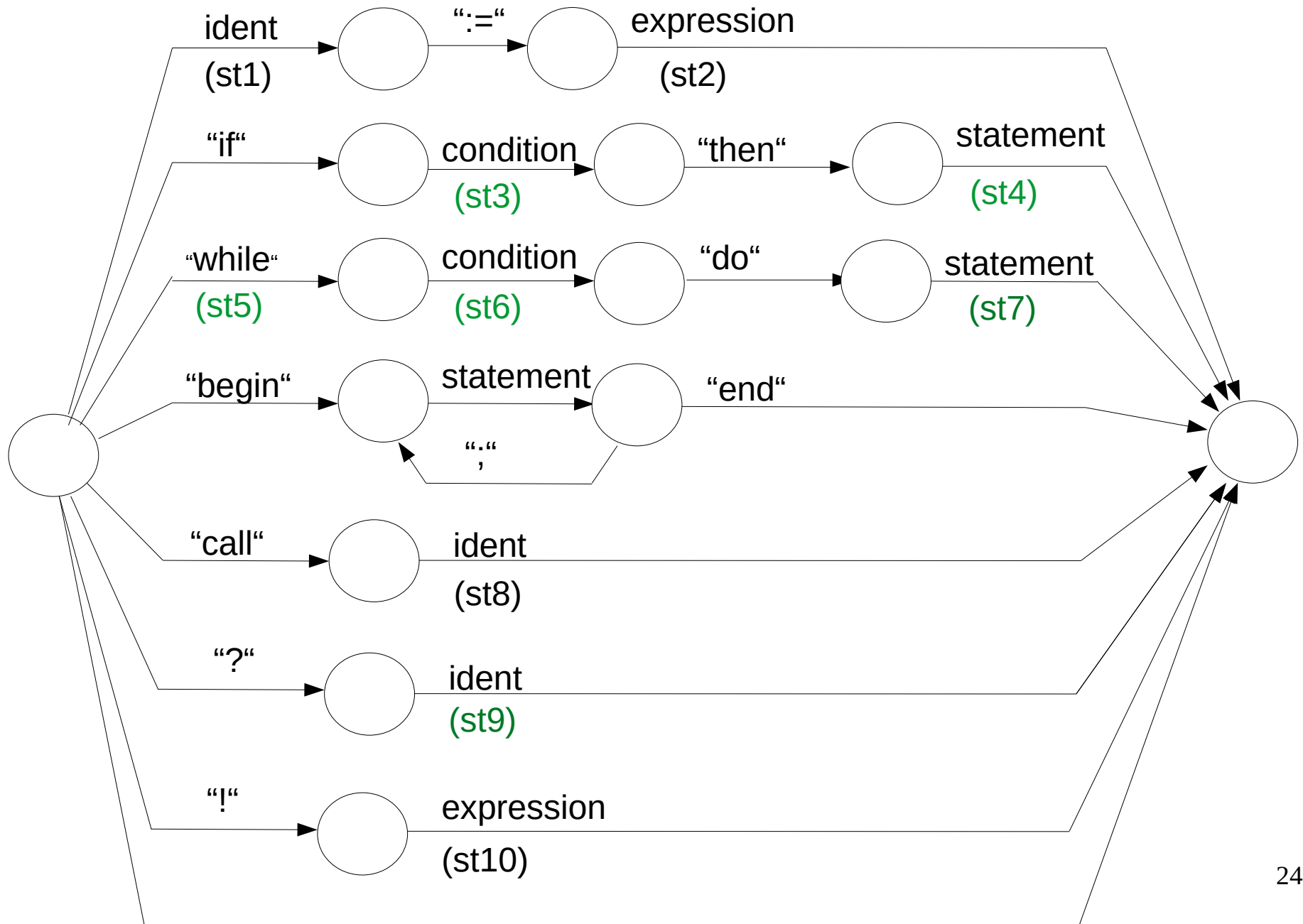
condition



co2 – co7 (Vergleichsoperator)

der später zu generierende Code für den Vergleichsoperator wird in einer einfachen Variable gespeichert. Dies ist hier möglich, da expression keinen Vergleich enthalten kann. Anderenfalls wäre ein Umbau der Syntaxbeschreibung, ähnlich expression notwendig, oder die Operatoren müssen in einem gesonderten Stack verwaltet werden.

Statement



Labels

- Labels sind Konstrukte der Codegenerierungsroutinen , in denen der aktuelle Stand des Schreibzeigers in den Codeausgabebereich zwischengespeichert wird.
- Labels werden für die Codegenerierung der bedingten (if) und der Schleifenanweisung (while) benötigt.
- Labels werden in einem Keller (Liste vorn einfügen, vorn entnehmen) verwaltet.

```
typedef  
struct tLABEL  
{  
    tKz Kz;  
    short iJmp;  
}tLabl;
```

Die Labelwerte
sollten relativ zum
Codepuferanfang
als Index
gespeichert werden.

Jump-Befehle

- Die Sprungbefehle jmp und jnot arbeiten mit Relativadressen
- Die als Operand angegebene Sprungdistance wird zum aktuellen Instructionpointer addiert.
- Dabei ist zu beachten, dass der aktuelle Instructionpointer zu diesem Zeitpunkt genau hinter dem Sprungbefehl steht und bereits auf den nächsten Befehl zeigt, da der Sprungbefehl ja bereits komplett gelesen ist.

			Relativadresse (11 bzw. 0xb)														
jmp	xx	xx															
Sprungbefehl														Anweisungen am Ziel			
↑ Label			Sprungdistance (11)														
			+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11			

conditional statement

st3 (if, nach Condition)

Generieren eines Labels, es zeigt auf den nächsten freien Speicherplatz des Codeausgabepuffers.

Codegenerierung jnot mit einer vorläufigen Relativadresse 0

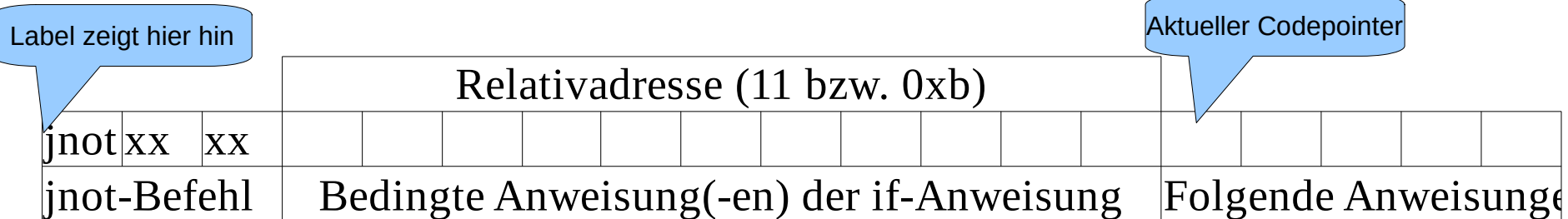
st4 (if, nach Statement)

Label auskellern

Relativadresse berechnen (akt. Codezeiger – CodePos in Label -3).

Minus 3, wenn das Label auf den jnot-Befehl zeigt.

(Es ist hilfreich, sich hier die Arbeitsweise von jmp /jnot zu verdeutlichen. Die als Parameter des jmp-Befehls übergebene Relativadresse wird zum aktuellen Instructionpointer, der zu dieser Zeit hinter den jnot-Befehl zeigt, addiert und in den Instructionpointer geschrieben.)



Die Relativadresse in der Skizze würde 11 (0xb) betragen
Relativadresse in jmp-Befehl eintragen.

loop statement

				jnot	xx	xx										jmp	yy	yy								
Condition				jnot-Befehl			Bedingte Anweisung(-en) der if-Anweisung														Folgende Anweisungen					
▲				▲															▲							

st5 (while)

Generieren eines Labels für Rücksprung am Schleifenende (linke Markierung).

st6 (while, nach Condition)

Generieren eines Labels, es zeigt auf den nächsten freien Speicherplatz des Codeausgabepuffers (mittlereMarkierung)

Codegenerierung jnot mit einer vorläufigen Relativadresse 0

st7 (while, nach Statement)

Label auskellern

Relativadresse berechnen, so ähnlich, wie bei st4, jedoch müssen noch 3 Byte für den jmp-Befehl am Ende der while-Anweisung eingerechnet werden (zu der berechneten Relativadresse muss die Länge des jmp-Befehls addiert werden addiert werden)

Label (aus st5) auskellern und jmp-Befehl generieren. Die Relativadresse muss so berechnet werden, dass der Sprung bei dem 1. Befehl von Condition landet.

Beispiel, kompletter Code

var a,fac;	EntryProc	0032, 0001, 0004		
	PushAdrVarLocal	0000		
Procedure p1;	PushValVarMain	0000		
var b;	StoreVal		EntryProc	001A, 0000, 0008
begin	PushAdrVarMain	0000	PushAdrVarMain	0000
b := a;	PushValVarMain	0000	GetVal	
a := a-1;	PushConst	0000	PushAdrVarMain	0004
if a>1 then	Sub		PushConst	0000
call p1;	StoreVal		StoreVal	
fac := fac*b	PushValVarMain	0000	Call	0001
end;	PushConst	0000	PushValVarMain	0004
	CmpGreaterThen		PutVal	
begin	JmpNot	0003	ReturnProc	
?a;	Call	0001	Const 0000:0001	
fac := 1;	PushAdrVarMain	0004		
call p1;	PushValVarMain	0004		
!fac	PushValVarLocal	0000		
end.	Mul			
	StoreVal			
	ReturnProc			

Reihenfolge der Bearbeitung

- Nachfolgende Folien geben eine Empfehlung zur Reihenfolge der Bearbeitung bei der Programmierung der Codeerzeugung.
- Die Reihenfolge orientiert sich an der Testbarkeit von Beispielen.
- Bereits nach der Implementation weniger Befehle, sollte sich das Programm

!5.

compilieren und mittels der bereitgestellten virtuellen Maschine ausführen lassen.

Reihenfolge

Factor (fa1), Block (bl6, bl5), Statement (st10), Pogramm (pr1)

PL0	Code
!5.	0000: 1A EntryProc 000C,0000,0000 <-- Procedure
	0007: 06 PushConst 0000
	000A: 08 PutVal
	000B: 17 ReturnProc
	Const 0000:0005

Testen Sie auch
das Programm:

```
const fuenf=5;  
!fuenf.
```

```
beck@beck-TUXEDO:~/PL0/X64$ cat t1_1.pl0  
const fuenf=5;  
!fuenf.  
beck@beck-TUXEDO:~/PL0/X64$ cpl0 t1_1.pl0  
  
OK  
beck@beck-TUXEDO:~/PL0/X64$ r t1_1  
5  
beck@beck-TUXEDO:~/PL0/X64$ hexdump -C t1_1.cl0  
00000000 01 00 00 00 1a 0c 00 00 00 00 00 06 00 00 08 17 |.....|  
00000010 05 00 00 00 |....|  
00000014  
beck@beck-TUXEDO:~/PL0/X64$
```

Expression, Term

PL0	Code
const c=1; !5+c.	0000: 1A EntryProc 0010,0000,0000 <-- Procedure 0007: 06 PushConst 0001 000A: 06 PushConst 0000 000D: 0C Add 000E: 08 PutVal 000F: 17 ReturnProc Const 0000:0001 Const 0001:0005

PL0	Code
const c=2; !5+c*3.	0000: 1A EntryProc 0014,0000,0000 <-- Procedure 0007: 06 PushConst 0001 000A: 06 PushConst 0000 000D: 06 PushConst 0002 0010: 0E Mul 0011: 0C Add 0012: 08 PutVal 0013: 17 ReturnProc Const 0000:0002 Const 0001:0005 Const 0002:0003

Statement Assignment

PL0	Code
const c=2;	0000: 1A EntryProc 001F,0000,0004 <-- Procedure
var a;	0007: 04 PushAdrVarMain 0000
begin	000A: 06 PushConst 0000
a:=2*c;	000D: 06 PushConst 0000
!5+c*a	0010: 0E Mul
end.	0011: 07 StoreVal
	0012: 06 PushConst 0001
	0015: 06 PushConst 0000
	0018: 01 PushValVarMain 0000
	001B: 0E Mul
	001C: 0C Add
	001D: 08 PutVal
	001E: 17 ReturnProc
	Const 0000:0002
	Const 0001:0005

Statement input

PL0	Code
const c=2;	0000: 1A EntryProc 0018,0000,0004 <-- Procedure
var a;	0007: 04 PushAdrVarMain 0000
begin	000A: 09 GetVal
?a;	000B: 01 PushValVarMain 0000
!a*a+c	000E: 01 PushValVarMain 0000
end.	0011: 0E Mul
	0012: 06 PushConst 0000
	0015: 0C Add
	0016: 08 PutVal
	0017: 17 ReturnProc
	Const 0000:0002

Statement Conditional

PL0	Code
const c=2;	0000: 1A EntryProc 002C,0000,000C <-- Procedure
var a,b,m;	0007: 04 PushAdrVarMain 0000
begin	000A: 09 GetVal
?a;	000B: 04 PushAdrVarMain 0004
?b;	000E: 09 GetVal
m:=a;	000F: 04 PushAdrVarMain 0008
if b>m then	0012: 01 PushValVarMain 0000
m:=b;	0015: 07 StoreVal
!m	0016: 01 PushValVarMain 0004
end.	0019: 01 PushValVarMain 0008
	001C: 13 CmpGreaterThen
	001D: 19 JmpNot 0007
	0020: 04 PushAdrVarMain 0008
	0023: 01 PushValVarMain 0004
	0026: 07 StoreVal
	0027: 01 PushValVarMain 0008
	002A: 08 PutVal
	002B: 17 ReturnProc
	Const 0000:0002

Weitere Beispiele testen

Statement Procedure Call

PL0	Code
const c=2;	0000: 1A EntryProc 0024,0001,0000 <-- Procedure
var a,b,m;	0007: 04 PushAdrVarMain 0008
procedure p;	000A: 01 PushValVarMain 0000
begin	000D: 07 StoreVal
m:=a;	000E: 01 PushValVarMain 0004
if b>m then	0011: 01 PushValVarMain 0008
m:=b;	0014: 13 CmpGreaterThen
!m	0015: 19 JmpNot 0007
end;	0018: 04 PushAdrVarMain 0008
begin	001B: 01 PushValVarMain 0004
?a;	001E: 07 StoreVal
?b;	001F: 01 PushValVarMain 0008
call p	0022: 08 PutVal
end.	0023: 17 ReturnProc
	0024: 1A EntryProc 0013,0000,000C <-- Procedure
	002B: 04 PushAdrVarMain 0000
	002E: 09 GetVal
	002F: 04 PushAdrVarMain 0004
	0032: 09 GetVal
	0033: 16 Call 0001
	0036: 17 ReturnProc
	Const 0000:0002

Procedure Call mit lokaler Variable

PL0	Code	
const c=2;	0000: 1A EntryProc	0027,0001,0004 <-- Procedure
var a,b,m;	0007: 03 PushAdrVarLocal	0000
procedure p;	000A: 01 PushValVarMain	0000
var x;	000D: 07 StoreVal	
begin	000E: 01 PushValVarMain	0004
x:=a;	0011: 00 PushValVarLocal	0000
if b>x then	0014: 13 CmpGreaterThen	
x:=b;	0015: 19 JmpNot	0007
m:=x	0018: 03 PushAdrVarLocal	0000
end;	001B: 01 PushValVarMain	0004
begin	001E: 07 StoreVal	
?a;	001F: 04 PushAdrVarMain	0008
?b;	0022: 00 PushValVarLocal	0000
call p;	0025: 07 StoreVal	
!m	0026: 17 ReturnProc	
end.	0027: 1A EntryProc	0017,0000,000C <-- Procedure
	002E: 04 PushAdrVarMain	0000
	0031: 09 GetVal	
	0032: 04 PushAdrVarMain	0004
	0035: 09 GetVal	
	0036: 16 Call	0001
	0039: 01 PushValVarMain	0008
	003C: 08 PutVal	
	003D: 17 ReturnProc	
	Const 0000:0002	

Procedure Call mit globaler Variable

PL0	Code	
const c=2;	0000: 1A EntryProc	002F,0002,0000 <-- Procedure
var a,b,m;	0007: 05 PushAdrVarGobal	0000, 0001
procedure p;	000C: 01 PushValVarMain	0000
var x;	000F: 07 StoreVal	
procedure q;	0010: 01 PushValVarMain	0004
begin	0013: 02 PushValVarGobal	0000,0001
x:=a;	0018: 13 CmpGreaterThen	
if b>x	0019: 19 JmpNot	0009
then	001C: 05 PushAdrVarGobal	0000, 0001
x:=b;	0021: 01 PushValVarMain	0004
m:=x	0024: 07 StoreVal	
end;	0025: 04 PushAdrVarMain	0008
call q;	0028: 02 PushValVarGobal	0000,0001
begin	002D: 07 StoreVal	
?a;	002E: 17 ReturnProc	
?b;	002F: 1A EntryProc	000B,0001,0004 <-- Procedure
call p;	0036: 16 Call	0002
!m	0039: 17 ReturnProc	
end.	003A: 1A EntryProc	0017,0000,000C <-- Procedure
	0041: 04 PushAdrVarMain	0000
	0044: 09 GetVal	
	0045: 04 PushAdrVarMain	0004
	0048: 09 GetVal	
	0049: 16 Call	0001
	004C: 01 PushValVarMain	0008
	004F: 08 PutVal	
	0050: 17 ReturnProc	
	Const 0000:0002	

Statement while

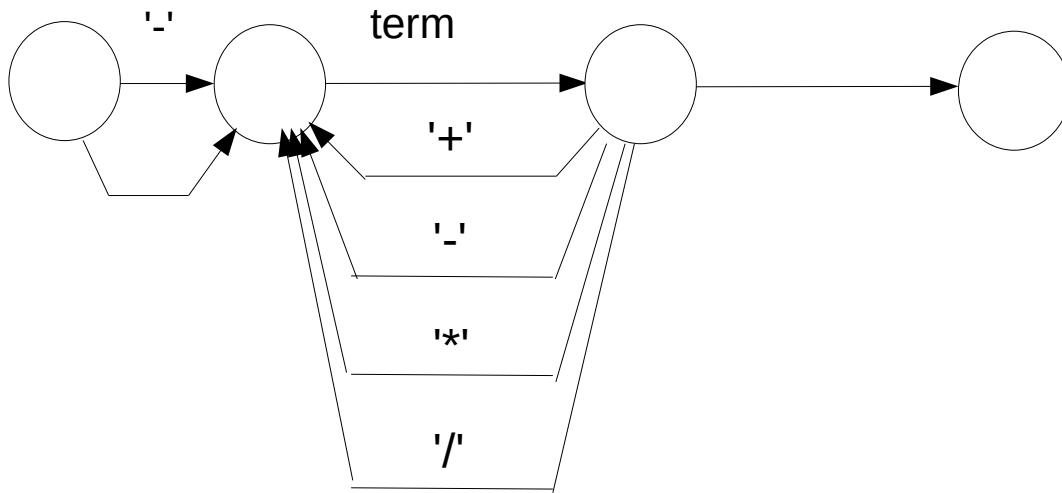
PL0	Code	
const c=10;	0000: 1A EntryProc	002B,0000,0004 <-- Procedure
var i;	0007: 04 PushAdrVarMain	0000
begin	000A: 06 PushConst	0001
i:=0;	000D: 07 StoreVal	
while i<c do	000E: 01 PushValVarMain	0000
begin	0011: 06 PushConst	0000
!i;	0014: 12 CmpLessThen	
i:=i+1	0015: 19 JmpNot	0012
end	0018: 01 PushValVarMain	0000
end.	001B: 08 PutVal	
	001C: 04 PushAdrVarMain	0000
	001F: 01 PushValVarMain	0000
	0022: 06 PushConst	0002
	0025: 0C Add	
	0026: 07 StoreVal	
	0027: 18 Jmp	FFE4
	002A: 17 ReturnProc	
	Const 0000:000A	
	Const 0001:0000	
	Const 0002:0001	

Alternative Möglichkeit zur Implementation von Expressions

(nur zur weiterführenden Information, nicht zu implementieren)

- Häufige rekursive Aufrufe innerhalb von Expression bedingen eine relativ langsame Ausführung des Compilers.
- Man kann hier auch ein anderes Verfahren einsetzen, in dem man die Graphen umgestaltet und die Routinen dazu neu fasst.
- Ein gängiges Verfahren besteht darin, den Operatoren Prioritäten zuzuordnen und sie entsprechend Ihrer Priorität zu bewerten (Code zu generieren) oder sie zu kellern.

Operatorprioritäten (precedence rules)



Verfahren:

Operanden werden durch Generierung der Push-Befehle unverändert behandelt.

Operatoren werden gekellert.

Hat der aktuelle Operator eine kleinere Priorität als der oberste Operator im Stack, so wird dieser ausgekellert und sein Code generiert

Dies wiederholt sich, solange der aktuelle Operator eine kleinere Priorität als der oberste Operator im Stack hat.

Der Stack wird mit einem Startoperator (Priorität 0) initialisiert.

Ergebnis ist eine Ausdrucksberechnung nach umgekehrt polnischer Notation

Operator	Priorität
- (unär)	3
*	2
/	2
+	1
-	1

Beispiel: $2+5*4-6/2$

Ende des Ausdrucks:
Alles auskellern

	* (2)		/ (2)		
+ (1)	+ (1)	- (1)	- (1)		
\$ (0)	\$ (0)	\$ (0)	\$ (0)	\$ (0)	\$ (0)

Push 2

Push 5

Push 4

Mul

Add

Push 6

Push 2

Div

Sub

Die Werte in runden Klammern geben die Priorität der Operatoren an.

Im PL/0 Compiler:

```

tBog gExpr[] =
{
/* 0*/ {BgSy, { (unsigned long) '+' }, Ex2, 3, 1}, /* (0) --- '+' ---> (1) */
/* 1*/ {BgSy, { (unsigned long) '-' }, Ex1, 3, 2}, /* +---- '-' ----> (1) */
/* 2*/ {BgNl, { (unsigned long) 0 }, Ex2, 3, 0}, /* +-----> (1) */
/* 3*/ {BgGr, { (unsigned long) iFact }, NULL, 4, 0}, /* (1) --- fact --> (2) */
/* 4*/ {BgSy, { (unsigned long) '+' }, Ex4, 3, 5}, /* (2) ---- '+' --> (1) */
/* 5*/ {BgSy, { (unsigned long) '-' }, Ex5, 3, 6}, /* +----- '-' --> (1) */
/* 6*/ {BgSy, { (unsigned long) '*' }, Ex6, 3, 7}, /* +----- '*' --> (1) */
/* 7*/ {BgSy, { (unsigned long) '/' }, Ex7, 3, 8}, /* +----- '/' --> (1) */
/* 8*/ {BgNl, { (unsigned long) 0 }, Ex8, 9, 0}, /* (2) -----> (3) */
/* 9*/ {BgEn, { (unsigned long) 0 }, NULL, 0, 0} /* (2) -----> (ENDE) */
};

```

```

int Ex4 (void)
/* (2) ---- '+' ----> (1) */
{
    tnode* ptmp;
    while ((ptmp=opread())->p>=10)
    {
        code(ptmp->op);
        oppop();
    }
    return oppush(OpAdd);
}

```

```

int Ex8(void) /* (2) -----> (3) */
{
    // Auskellern bis OP==0
    tnode* ptmp;
    while ((ptmp=opread())->op!=0)
    {
        code(ptmp->op);
        oppop();
    }
    oppop();
    Return OK;
}

```

```

typedef struct t_node
{
    struct t_node * nxt;
    char op;
    char p;
}tnode;
tnode * pStack=0;

int oppush(char op) // Operatinscode kellern
{
    tnode* ptmp;
    ptmp=malloc(sizeof(tnode));
    if(ptmp)
    {
        switch(op)
        {
            case vzMinus: ptmp->p=30; break;
            case OpMult:
            case OpDiv :  ptmp->p=20; break;
            case OpAdd:
            case OpSub:  ptmp->p=10; break;
            case 0      :  ptmp->p= 0; break;
        }
        ptmp->op=op;
        ptmp->nxt=pStack;
        pStack=ptmp;
    }
    return ptmp?1:0;
}

```

Operatorenstack (einfach verkettete Liste)

```

// Operatinscode auskellern
{
void oppop ()
    tnode* ptmp;
    if(pStack)
    {
        ptmp=pStack;
        pStack=pStack->nxt;
        free(ptmp);
    }
}
// obersten Operatinscode lesen
tnode* oread()
{
    if(pStack) return pStack;
    else      return NULL;
}

```