

☒ Gr. 1, Dr. G. Kronberger Name PAPESH Konstantin Aufwand in h 8
☐ Gr. 2, Dr. H. Gruber
☐ Gr. 3, Dr. D. Auer Punkte _____ Kurzzeichen Tutor / Übungsleiter _____ / _____

1. MidiPascal

(10 Punkte)

MiniPascal ist eine ziemlich "schwache" Sprache, da man mit ihr nicht "alle" Probleme lösen kann – sofern es überhaupt (Programmier-)Sprachen gibt, mit denen man alle ... ;-). Wesentliche Sprachkonstrukte, die MiniPascal fehlen, sind Verzweigungen und Schleifen. Also erweitern wir MiniPascal um die binäre Verzweigung (*IF*-Anweisung), die Abweisschleife (*WHILE*-Schleife) sowie die Verbundanweisung (*BEGIN ... END*) – und taufen die neue Sprache MidiPascal.

Nachdem wir mit dem Datentyp *INTEGER* und ohne Erweiterungen der Ausdrücke um relationale Operatoren auskommen wollen, verwenden wir für Bedingungen in Verzweigungen und Schleifen *INTEGER*-Variablen mit der Semantik, dass jeder Wert ungleich 0 als *TRUE* und (nur) der Wert 0 als *FALSE* interpretiert wird – so wie das z. B. in der Programmiersprache C definiert ist. Folgende Tabelle zeigt zur Verdeutlichung eine Abbildung von MidiPascal auf (vollständiges) Pascal:

MidiPascal	(vollständiges) Pascal
<code>VAR x: INTEGER;</code>	<code>VAR x: INTEGER;</code>
<code>IF x THEN ...</code>	<code>IF x <> 0 THEN ...</code>
<code>WHILE x DO ...</code>	<code>WHILE x <> 0 DO ...</code>

Mit diesen Spracherweiterungen könnte man dann z. B. ein MidiPascal-Programm schreiben, das für eine vom Benutzer / von der Benutzerin eingegebene Zahl *n* die Fakultät $f = n!$ iterativ berechnet und diese ausgibt. Siehe Quelltextstück rechts.

```
f := n; n := n - 1;
WHILE n DO BEGIN
    f := n * f;
    n := n - 1;
END;
WRITE(f);
```

Damit diese neuen Sprachkonstrukte im Compiler umgesetzt werden können, sind zwei neue Bytecode-Befehle notwendig. Folgende Tabelle erläutert diese beiden Befehle:

Bytecode-Befehl	Semantik
<code>Jmp addr</code>	Springe an die Codeadresse <i>addr</i>
<code>JmpZ addr</code>	Hole oberstes Element vom Stapel und wenn dieses 0 (<i>zero</i>) ist, springe nach <i>addr</i>

Nun muss man nur noch klären, welche Bytecodestücke für die einzelnen, neuen MidiPascal-Anweisungen zu erzeugen sind. Folgende Tabelle stellt die notwendigen Transformationen anhand von Mustern dar:

MidiPascal	Bytecode (mit fiktiven Adressen)
<code>IF x THEN BEGIN</code>	1 <code>LoadVal x</code>
<i>then stats</i>	4 <code>JmpZ 99</code>
<code>END;</code>	... <i>code for then stats</i>
...	99 ...

MidiPascal	Bytecode (mit fiktiven Adressen)
IF x THEN BEGIN	1 LoadVal x
then stats	4 JmpZ 66
END	... code for then stats
ELSE BEGIN	... Jmp 99
else stats	66 code for else stats
END;	99 ...
...	
WHILE x DO BEGIN	1 LoadVal x
while stats	4 JmpZ 99
END	... code for while stats
	... Jmp 1
	99 ...

Bei der Implementierung dieser neuen Sprachkonstrukte tritt das Problem auf, für die Bedingungen auch Sprunganweisungen „nach unten“ erzeugen zu müssen, wobei die Zieladressen der Sprünge noch nicht bekannt sind. Dieses Problem kann mit dem so genannten *Anderthalbpass-Verfahren* gelöst werden: Man erzeugt zuerst eine Sprunganweisung mit einer fiktiven Adresse (z. B. 0) und korrigiert diese später, sobald die Zieladresse bekannt ist (mittels *FixUp*).

Im Moodle-Kurs finden Sie in *ForMidiPascalCompiler.zip* einen, um die beiden neuen Bytecode-Befehle und zwei neue Operationen (*CurAddr* und *FixUp*) erweiterten Code-Generator (*CodeDef.pas* und *CodeGen.pas*) und eine erweiterte MidiPascal-Maschine (*CodeInt.pas*), welche die neuen Befehle ausführen kann. Sie müssen also nur mehr den lexikalischen Analysator um die neuen Schlüsselwörter und den Syntaxanalysator mit seinen semantischen Aktionen um die neuen Anweisungen erweitern. Verwenden Sie als Basis dazu folgenden Ausschnitt der ATG für MidiPascal:

```
Stat = [ ... (*assignment, read, and write statement here, new ones below*)
```

```
| 'BEGIN' StatSeq 'END'
```

```
| 'IF' ident↑identStr      SEM IF NOT IsDecl(identStr) THEN BEGIN
                           SemError('variable not declared');
                           END; (*IF*)
                           Emit2(LoadValOpc, AddrOf(identStr));
                           Emit2(JmpZOpc, 0); (*0 as dummy address*)
                           addr := CurAddr - 2; ENDSEM
```

```
    'THEN' Stat
```

```
[ 'ELSE'                  SEM Emit2(JmpOpc, 0); (*0 as dummy address*)
                           FixUp(addr, CurAddr);
                           addr := CurAddr - 2; ENDSEM
```

```
    Stat
```

```
]                          SEM FixUp(addr, CurAddr); ENDSEM
```

```
| 'WHILE' ident↑identStr  SEM IF NOT IsDecl(identStr) THEN BEGIN
                           SemError('variable not declared');
                           END; (*IF*)
                           addr1 := CurAddr;
                           Emit2(LoadValOpc, AddrOf(identStr));
                           Emit2(JmpZOpc, 0); (*0 as dummy address*)
                           addr2 := CurAddr - 2; ENDSEM
```

```
    'DO' Stat
```

```
SEM Emit2(JmpOpc, addr1);
FixUp(addr2, CurAddr); ENDSEM
```

```
] .
```

2. Optimierender MidiPascal-Compiler

(2 + 4 + 4 + 4 Punkte)

Arithmetische Ausdrücke kann man wie folgt durch Binärbäume darstellen: aus dem Operator wird der Wurzelknoten, aus dem linken Operanden der linke und aus dem rechten Operanden der rechte Teilbaum. (Sie kennen das ja schon aus Übung 6, Aufgabe 2.) Sobald ein Ausdruck in Form eines Binärbaums im Hauptspeicher vorliegt, ist es einfach, diesen mittels Baumdurchlauf (in-, pre- oder postorder), wieder in eine Textform (z. B. In-, Prä- oder Postfix-Notation) zu übersetzen.

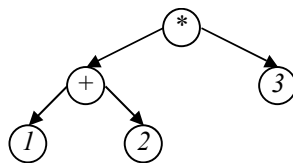
Die Repräsentation von arithmetischen Ausdrücken in Form von Binärbäumen bietet aber auch die Möglichkeit, einfache Optimierungen in den MidiPascal-Compiler einzubauen.

- a) Ändern Sie die Erkennungsprozeduren für arithmetische Ausdrücke (*Expr*, *Term* und *Fact*) im Parser Ihres MidiPascal-Compilers so ab, dass vorerst kein Code mehr für die Ausdrücke erzeugt, sondern ein Binärbaum aufgebaut wird, dessen Knoten Zeichenketten enthalten (die vier Operatoren, die Ziffernfolge einer Zahl oder den Bezeichner einer Variablen).
- b) Erweitern Sie dann das Code-Generierungsmodul um eine

```
PROCEDURE EmitCodeForExprTree(t: TreePtr);
```

die aus dem Binärbaum in einem Postorder-Durchlauf Bytecode für die Berechnung des Ausdrucks durch die virtuelle MiniPascal-Maschine erzeugt.

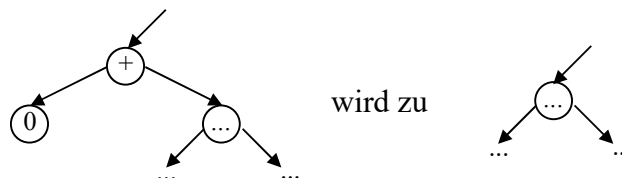
Beispiel: Für den Ausdruck $(1 + 2) * 3$ soll der links dargestellte Baum aufgebaut werden, und die Prozedur *EmitCodeForExprTree* soll daraus die rechts angegebene Codesequenz erzeugen:



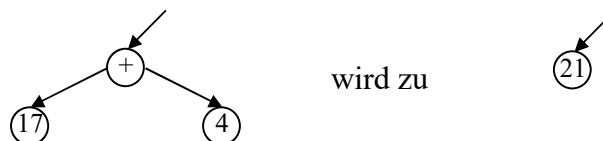
```
LoadConst 1
LoadConst 2
Add
LoadConst 3
Mul
```

Damit können Sie Ihren Compiler zwar schon testen – aber von Optimierung ist noch keine Rede. Die erzeugten Binärbäume eignen sich aber dazu, einfache Optimierungen an Ausdrücken vorzunehmen, die z. B. in modernen Compilern eingesetzt werden: die Binärbäume werden transformiert und erst die sich daraus ergebenden Bäume werden für die Codegenerierung herangezogen.

- c) Eliminieren überflüssiger Rechenoperationen,
z. B.: $0 + \dots$ oder $\dots + 0$ oder $1 * \dots$ oder $\dots * 1$ oder $\dots / 1$ wird zu \dots
oder in Baumform (für das erste Beispiel) dargestellt:



- d) „Konstantenfaltung“, Berechnung konstanter Teilausdrücke,
z. B.: $\dots + 17 + 4 + \dots$ wird zu $\dots + 21 + \dots$



Versuchen Sie, möglichst viele solcher optimierenden Baumtransformationen zu implementieren und wenden Sie diese solange auf den Baum an, als sich dadurch Verbesserungen ergeben.

Durch diese Transformationen soll z. B. aus dem Baum für $0 + (17 + 4) * 1$ ein Baum mit nur mehr einem Knoten für 21 entstehen.

ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP

– SS 2018

Übungsabgabe 8

Konstantin Papesh

30. Mai 2018

8.1 MidiPascal

8.1.1 Lösungsidee

Ein Großteil der Lösung wird bereits in der Angabe diskutiert. Um das Programm um Schleifen erweitern zu können muss das im Unterricht ausgearbeitete Beispiel um die angegebene ATG erweitert werden. Dies geschieht an den jeweiligen Stellen in *MP_SS.pas*. Um das Programm dann auch erfolgreich wieder zu disassembeln muss in der Datei *CodeDis.pas* die beiden hinzugefügten Funktionen *Imp* und *ImpZ*.

8.1.2 Implementierung

Listing 8.1: MP_SS.pas

```
1 (* MPP_SS:                                     HDO, 2004-02-06
2  -----
3  Syntax analyzer and semantic evaluator for the MiniPascal parser.
4  Semantic actions to be included in MPI_SS and MPC_SS.
5  =====
6  *)
7  UNIT MPC_SS;
8  INTERFACE
9
10   VAR
11     success: BOOLEAN; (*true if no syntax errors*)
12
13   PROCEDURE S;          (*parses whole MiniPascal program*)
14
15
16 IMPLEMENTATION
17
18   USES
19     MP_Lex, SymTab, CodeDef, CodeGen;
20
21
```

```

22 FUNCTION SyIsNot(expectedSy: Symbol): BOOLEAN;
23 BEGIN
24     success:= success AND (sy = expectedSy);
25     SyIsNot := NOT success;
26 END; (*SyIsNot*)
27
28 PROCEDURE SemErr(msg: STRING);
29 BEGIN
30     WriteLn('*** Semantic error ***');
31     WriteLn(' ', msg);
32     success := FALSE;
33 end;
34
35
36 PROCEDURE MP; FORWARD;
37 PROCEDURE VarDecl; FORWARD;
38 PROCEDURE StatSeq; FORWARD;
39 PROCEDURE Stat; FORWARD;
40 PROCEDURE Expr; FORWARD;
41 PROCEDURE Term; FORWARD;
42 PROCEDURE Fact; FORWARD;
43
44 PROCEDURE S;
45 (*
-----
*)
46 BEGIN
47     WriteLn('parsing started ...');
48     success := TRUE;
49     MP;
50     IF NOT success OR SyIsNot(eofSy) THEN
51         WriteLn('*** Error in line ', syLnr:0, ', column ', syCnr:0)
52     ELSE
53         WriteLn('... parsing ended successfully ');
54 END; (*S*)
55
56 PROCEDURE MP;
57 BEGIN
58     IF SyIsNot(programSy) THEN Exit;
59     (* sem *)
60     initSymbolTable;
61     InitCodeGenerator;
62     (* endsem *)
63     NewSy;
64     IF SyIsNot(identSy) THEN Exit;
65     NewSy;
66     IF SyIsNot(semicolonSy) THEN Exit;
67     NewSy;
68     IF sy = varSy THEN BEGIN
69         VarDecl; IF NOT success THEN Exit;
70     END; (*IF*)
71     IF SyIsNot(beginSy) THEN Exit;
72     NewSy;
73     StatSeq; IF NOT success THEN Exit;
74     (* sem *)
75     Emit1(EndOpc);
76     (* endsem *)

```

```

77     IF SyIsNot(endSy) THEN Exit;
78     NewSy;
79     IF SyIsNot(periodSy) THEN Exit;
80     NewSy;
81     END; (*MP*)
82
83 PROCEDURE VarDecl;
84 var ok : BOOLEAN;
85 BEGIN
86     IF SyIsNot(varSy) THEN Exit;
87     NewSy;
88     IF SyIsNot(identSy) THEN Exit;
89     (* sem *)
90     DeclVar(identStr, ok);
91     NewSy;
92     WHILE sy = commaSy DO BEGIN
93         NewSy;
94         IF SyIsNot(identSy) THEN Exit;
95         (* sem *)
96         DeclVar(identStr, ok);
97         IF NOT ok THEN
98             SemErr('mult. decl. ');
99         (* endsem *)
100        NewSy;
101    END; (*WHILE*)
102    IF SyIsNot(colonSy) THEN Exit;
103    NewSy;
104    IF SyIsNot(integerSy) THEN Exit;
105    NewSy;
106    IF SyIsNot(semicolonSy) THEN Exit;
107    NewSy;
108    END; (*VarDecl*)
109
110 PROCEDURE StatSeq;
111 BEGIN
112     Stat; IF NOT success THEN Exit;
113     WHILE sy = semicolonSy DO BEGIN
114         NewSy;
115         Stat; IF NOT success THEN Exit;
116     END; (*WHILE*)
117 END; (*StatSeq*)
118
119 PROCEDURE Stat;
120 var destId : STRING;
121     addr, addr1, addr2 : integer;
122 BEGIN
123     CASE sy OF
124         identSy: BEGIN
125             (* sem *)
126             destId := identStr;
127             IF NOT IsDecl(destId) then
128                 SemErr('var. not decl. ');
129             ELSE
130                 Emit2(LoadAddrOpc, AddrOf(destId));
131             (* endsem *)
132             NewSy;
133             IF SyIsNot(assignSy) THEN Exit;

```

```

134     NewSy;
135     Expr; IF NOT success THEN Exit;
136     (* sem *)
137     IF IsDecl(destId) THEN
138         Emit1(StoreOpc);
139     (* endsem *)
140     END;
141 readSy: BEGIN
142     NewSy;
143     IF SyIsNot(leftParSy) THEN Exit;
144     NewSy;
145     IF SyIsNot(identSy) THEN Exit;
146     (* sem *)
147     IF NOT IsDecl(identStr) THEN
148         SemErr('var not decl. ');
149     ELSE BEGIN
150         Emit2(ReadOpc, AddrOf(identStr));
151     END;
152     (* endsem *)
153     NewSy;
154     IF SyIsNot(rightParSy) THEN Exit;
155     NewSy;
156     END;
157 writeSy: BEGIN
158     NewSy;
159     IF SyIsNot(leftParSy) THEN Exit;
160     NewSy;
161     Expr; IF NOT success THEN Exit;
162     (* sem *)
163     Emit1(WriteOpc);
164     (* endsem *)
165     IF SyIsNot(rightParSy) THEN Exit;
166     NewSy;
167     END;
168 beginSy: BEGIN
169     newSy;
170     StatSeq;
171     IF SyIsNot(endSy) THEN Exit;
172     NewSy;
173     END;
174 ifSy: BEGIN
175     NewSy;
176     IF SyIsNot(identSy) THEN Exit;
177     (* SEM *)
178     IF NOT IsDecl(identStr) THEN BEGIN
179         SemErr('var not decl. ');
180     END;
181     Emit2(LoadValOpc, AddrOf(identStr));
182     Emit2(JmpZOpc, 0); (*0 as dummy address*)
183     addr := CurAddr - 2;
184     (* ENDSEM *)
185     NewSy;
186     IF SyIsNot(thenSy) THEN Exit;
187     newSy;
188     Stat; IF NOT success THEN Exit;
189     IF sy = elseSy THEN BEGIN
190         newSy;

```

```

191          (* SEM *)
192          Emit2(JmpOpc, 0);
193          FixUp(addr, CurAddr);
194          addr := CurAddr - 2;
195          (* ENDSEM *)
196          Stat; IF NOT success THEN Exit;
197      END;
198      (* SEM *)
199      FixUp(addr, CurAddr);
200      (* ENDSEM *)
201  END;
202  whileSy: BEGIN
203      NewSy;
204      IF SyIsNot(identSy) THEN Exit;
205      (* SEM *)
206      IF NOT IsDecl(identStr) THEN BEGIN
207          SemErr('var not decl. ');
208      END;
209      addr1 := CurAddr;
210      Emit2(LoadValOpc, AddrOf(identStr));
211      Emit2(JmpZOpc, 0); (*0 as dummy address*)
212      addr2 := CurAddr - 2;
213      (* ENDSEM *)
214      NewSy;
215      IF SyIsNot(doSy) THEN Exit;
216      NewSy;
217      Stat;
218      (* SEM *)
219      Emit2(JmpOpc, addr1);
220      FixUp(addr2, CurAddr);
221      (* ENDSEM *)
222  END;
223  ELSE
224      ; (*EPS*)
225  END; (*CASE*)
226  END; (*Stat*)
227
228  PROCEDURE Expr;
229  BEGIN
230      Term; IF NOT success THEN Exit;
231      WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
232          CASE sy OF
233              plusSy: BEGIN
234                  NewSy;
235                  Term; IF NOT success THEN Exit;
236                  (* sem *)
237                  Emit1(AddOpc);
238                  (* endsem *)
239              END;
240              minusSy: BEGIN
241                  NewSy;
242                  Term; IF NOT success THEN Exit;
243                  (* sem *)
244                  Emit1(SubOpc);
245                  (* endsem *)
246              END;
247          END; (*CASE*)

```



```

248     END; (*WHILE*)
249 END; (*Expr*)
250
251 PROCEDURE Term;
252 BEGIN
253     Fact; IF NOT success THEN Exit;
254     WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
255         CASE sy OF
256             timesSy: BEGIN
257                 NewSy;
258                 Fact; IF NOT success THEN Exit;
259                 (* sem *)
260                 Emit1(MulOpc);
261                 (* endsem*)
262             END;
263             divSy: BEGIN
264                 NewSy;
265                 Fact; IF NOT success THEN Exit;
266                 (* sem *)
267                 Emit1(DivOpc);
268                 (* endsem *)
269             END;
270         END; (*CASE*)
271     END; (*WHILE*)
272 END; (*Term*)
273
274 PROCEDURE Fact;
275 BEGIN
276     CASE sy OF
277         identSy: BEGIN
278             (* sem *)
279             IF NOT IsDecl(identStr) THEN
280                 SemErr('var. not decl.')
281             ELSE
282                 Emit2(LoadValOpc,AddrOf(identStr));
283             (* endsem *)
284             NewSy;
285         END;
286         numberSy: BEGIN
287             (* sem*)
288             Emit2(LoadConstOpc,numberVal);
289             (* endsem *)
290             NewSy;
291         END;
292         leftParSy: BEGIN
293             NewSy;
294             Expr; IF NOT success THEN Exit;
295             IF SyIsNot(rightParSy) THEN Exit;
296             NewSy;
297         END;
298     ELSE
299         success := FALSE;
300     END; (*CASE*)
301 END; (*Fact*)
302
303
304 END. (*MPP_SS*)

```

Listing 8.2: CodeDis.pas

```

1  (* CodeDis:                                     HDO, 2004-02-06
2  -----
3  Byte code disassembler for the MiniPascal compiler.
4  =====
5  *)
6  UNIT CodeDis;
7  INTERFACE
8
9  USES
10   CodeDef;
11
12  PROCEDURE DisassembleCode(ca: CodeArray);
13
14
15  IMPLEMENTATION
16
17  VAR
18   ca: CodeArray; (*array of opCodes and operands*)
19   pc: INTEGER;   (*program counter*)
20
21
22  PROCEDURE FetchOpc(VAR opc: OpCode);
23  BEGIN
24   opc := OpCode(ca[pc]);
25   pc := pc + 1;
26  END; (*FetchOpc*)
27
28  PROCEDURE FetchOpd(VAR opd: INTEGER);
29  BEGIN
30   opd := Ord(ca[pc])*256 + Ord(ca[pc + 1]);
31   pc := pc + 2;
32  END; (*FetchOpd*)
33
34  PROCEDURE Write1(opcStr:STRING; opc: OpCode);
35  BEGIN
36   Write('[', Ord(opc):2, '          ] ');
37   WriteLn(opcStr);
38  END; (*Write1*)
39
40  PROCEDURE Write2(opcStr:STRING; opc: OpCode; opd: INTEGER);
41  BEGIN
42   Write('[', Ord(opc):2, (opd DIV 256):4, (opd MOD 256):4, ' ] ');
43   WriteLn(opcStr, opd);
44  END; (*Write2*)
45
46
47  PROCEDURE DisassembleCode(ca: CodeArray);
48  (*
49   -----
50   *)
51  VAR
52   opc: OpCode;
53   opd: INTEGER;
54  BEGIN

```

```

53   CodeDis.ca := ca;
54   WriteLn('code disassembling started ...');
55   pc := 1;
56   REPEAT
57     Write(pc:4, ': ');
58     FetchOpc(opc);
59     CASE opc OF
60       LoadConstOpc: BEGIN
61         FetchOpd(opd);
62         Write2('LoadConst', opc, opd);
63       END;
64       LoadValOpc: BEGIN
65         FetchOpd(opd);
66         Write2('LoadVal ', opc, opd);
67       END;
68       LoadAddrOpc: BEGIN
69         FetchOpd(opd);
70         Write2('LoadAddr ', opc, opd);
71       END;
72       StoreOpc: BEGIN
73         Write1('Store ', opc);
74       END;
75       AddOpc: BEGIN
76         Write1('Add ', opc);
77       END;
78       SubOpc: BEGIN
79         Write1('Sub ', opc);
80       END;
81       MulOpc: BEGIN
82         Write1('Mul ', opc);
83       END;
84       DivOpc: BEGIN
85         Write1('Div ', opc);
86       END;
87       ReadOpc: BEGIN
88         FetchOpd(opd);
89         Write2('Read ', opc, opd);
90       END;
91       WriteOpc: BEGIN
92         Write1('Write ', opc);
93       END;
94       EndOpc: BEGIN
95         Write1('End ', opc);
96       END;
97       JmpZOpc: BEGIN
98         FetchOpd(opd);
99         Write2('JmpZ ', opc, opd);
100      END;
101      JmpOpc: BEGIN
102        FetchOpd(opd);
103        Write2('Jmp ', opc, opd);
104      END;
105    ELSE
106      WriteLn('*** Error: invalid operation code');
107      HALT;
108    END; (*CASE*)
109  UNTIL opc = EndOpc;

```

```

110   WriteLn('... code disassembling ended');
111   END; (*DisassembleCode*)
112
113
114 END. (*CodeDis*)

```

Listing 8.3: compare.mp

```

1 PROGRAM test;
2   VAR
3     a, b: INTEGER;
4 BEGIN
5   READ(a);
6   READ(b);
7
8   IF a THEN BEGIN
9     WRITE(a);
10  END
11 ELSE BEGIN
12   WRITE(b);
13 END;
14
15 END.

```

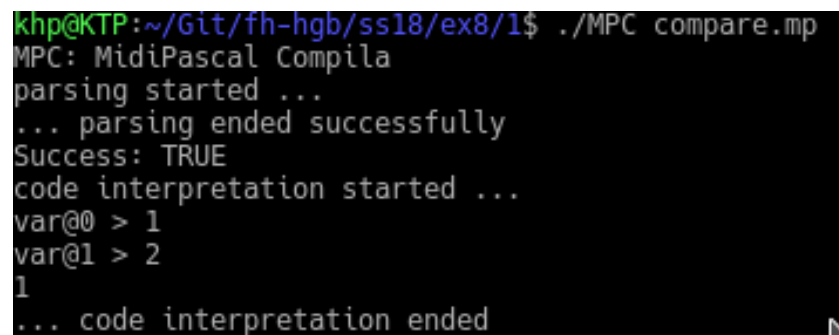
Listing 8.4: factorial.mp

```

1 PROGRAM factorial;
2 VAR
3   f,n: INTEGER;
4
5 BEGIN
6   Read(n);
7   f := n; n:=n-1;
8   WHILE n DO BEGIN
9     f := n * f;
10    n := n - 1;
11  END;
12  Write(f);
13 END.

```

8.1.3 Ausgabe

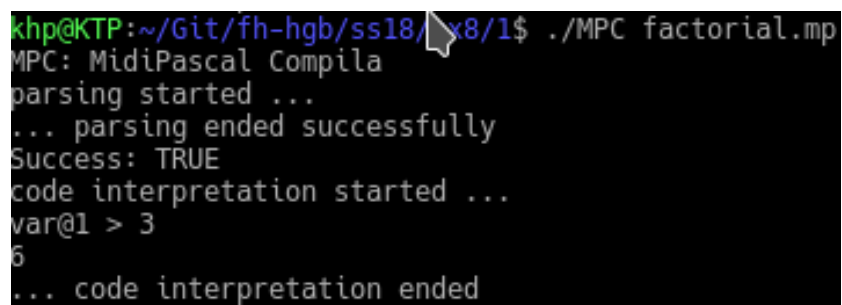


```

khp@KTP:~/Git/fh-hgb/ss18/ex8/1$ ./MPC compare.mp
MPC: MidiPascal Compila
parsing started ...
... parsing ended successfully
Success: TRUE
code interpretation started ...
var@0 > 1
var@1 > 2
1
... code interpretation ended

```

Abbildung 8.1: Vergleich zweier Variablen mithilfe von IF

A terminal window with a black background and green text. The prompt is 'khp@KTP:~/Git/fh-hgb/ss18/x8/1\$'. The command './MPC factorial.mp' has been executed. The output shows the MPC compiler parsing the file successfully and then interpreting the code. The code sets a variable 'var@1' to 3 and prints the factorial of 3, which is 6.

```
khp@KTP:~/Git/fh-hgb/ss18/x8/1$ ./MPC factorial.mp
MPC: MidiPascal Compila
parsing started ...
... parsing ended successfully
Success: TRUE
code interpretation started ...
var@1 > 3
6
... code interpretation ended
```

Abbildung 8.2: Factorial einer gegebenen Variable

8.2 Optimierender MidiPascal-Compiler

8.2.1 Lösungsidee

a

Damit ein Binärbaum aufgebaut werden kann, werden die semantischen Aktionen des vorigen Beispiels ersetzt. Dabei wird, anstatt den jeweiligen OP-Code in die Binärdatei einzufügen, dem Binärbaum ein Eintrag hinzugefügt.

b

Nun wird aus dem Binärbaum wiederum eine Binärdatei erzeugt mit den passenden OP-Codes. Der Zwischenschritt mit einem Binärbaum erlaubt es, das Programm effizient zu optimieren.

Dabei wird der Binärbaum in InOrder durchgegangen und die jeweilige Aktion in die Binärdatei eingefügt. Dabei ist auf die richtige Reihenfolge der Befehle zu achten¹.

c

Der Binärbaum wieder wieder Node für Node durchgegangen und die Zeichenkette des Knoten beachtet. Wenn in dieser Zeichenkette ein Operator sich befindet, überprüft die Funktion, ob in einem der beiden Ästen eine 'unnütze' Zahl sich befindet². Sollte dies der Fall sein, wird der aktuelle Knoten mit dem anderen Knoten ersetzt, welcher nicht diese Zahl enthält.

Dadurch wird der Baum um unnötige Rechenoperationen erleichtert.

d

Der Binärbaum wird ein letztes Mal durchgegangen um Knoten mit zwei konstanten Children zu ersetzen, da diese Rechenoperation immer das gleiche Ergebnis liefert. Daher kann der Knoten mit dem Ergebnis dieser Rechenoperation ersetzt werden.

Sind also zwei Konstanten vorhanden, wird die Rechenoperation schon beim Kompilieren ausgeführt und der Knoten mit dem Ergebnis dieser ersetzt. Also wird aus den Knoten $5 + 3$ der Knoten 8.

8.2.2 Implementierung

Listing 8.5: MP_SS.pas

```

1 (* MPP_SS:                                     HDO, 2004-02-06
2  -----
3  Syntax analyzer and semantic evaluator for the MiniPascal parser.
4  Semantic actions to be included in MPI_SS and MPC_SS.
5  =====
6  *)
6 UNIT MPC_SS;
```

¹Zuerst Variablen oder Konstanten laden, dann erst Rechenoperation

²Bei * und / ob eine Zahl 1 oder 0 ist, bei + oder - ob eine Zahl 0 ist.

```

7
8 INTERFACE
9
10  VAR
11    success: BOOLEAN; (*true if no syntax erros*)
12
13    PROCEDURE S;          (*parses whole MiniPascal program*)
14
15
16 IMPLEMENTATION
17
18  USES
19    MP_Lex, SymTab, CodeDef, CodeGen, BinTree;
20
21
22  FUNCTION SyIsNot(expectedSy: Symbol): BOOLEAN;
23  BEGIN
24    success:= success AND (sy = expectedSy);
25    SyIsNot := NOT success;
26  END; (*SyIsNot*)
27
28 PROCEDURE SemErr(msg: STRING);
29 BEGIN
30   WriteLn('*** Semantic error ***');
31   WriteLn(' ', msg);
32   success := FALSE;
33 end;
34
35
36  PROCEDURE MP(var binaryTree: TreePtr);    FORWARD;
37  PROCEDURE VarDecl; FORWARD;
38  PROCEDURE StatSeq(var binaryTree: TreePtr); FORWARD;
39  PROCEDURE Stat(var binaryTree: TreePtr);  FORWARD;
40  PROCEDURE Expr(var binaryTree: TreePtr);  FORWARD;
41  PROCEDURE Term(var binaryTree: TreePtr);  FORWARD;
42  PROCEDURE Fact(var binaryTree: TreePtr);  FORWARD;
43  PROCEDURE EmitCodeForExprTree(t: TreePtr); FORWARD;
44  PROCEDURE Simplify(var t: TreePtr);      FORWARD;
45  PROCEDURE ConstantFolding(var t: TreePtr); FORWARD;
46
47
48  PROCEDURE S;
49  (*
50
51      -----
52      *)
50 VAR
51   binaryTree: TreePtr;
52 BEGIN
53   binaryTree := newTree('');
54   WriteLn('parsing started ...');
55   success := TRUE;
56   MP(binaryTree);
57   IF NOT success OR SyIsNot eofSy THEN
58     WriteLn('*** Error in line ', syLnr:0, ' ', column ' ', syCnr:0)
59   ELSE
60     WriteLn('... parsing ended successfully ');
61   Simplify(binaryTree);

```

```

62     ConstantFolding(binaryTree);
63     writeTreeInOrder(binaryTree);
64 END; (*S*)
65
66 PROCEDURE MP(var binaryTree: TreePtr);
67 BEGIN
68     IF SyIsNot(programSy) THEN Exit;
69     (* sem *)
70     initSymbolTable;
71     InitCodeGenerator;
72     (* endsem *)
73     NewSy;
74     IF SyIsNot(identSy) THEN Exit;
75     NewSy;
76     IF SyIsNot(semicolonSy) THEN Exit;
77     NewSy;
78     IF sy = varSy THEN BEGIN
79         VarDecl; IF NOT success THEN Exit;
80     END; (*IF*)
81     IF SyIsNot(beginSy) THEN Exit;
82     NewSy;
83     StatSeq(binaryTree); IF NOT success THEN Exit;
84     (* sem *)
85     EmitCodeForExprTree(binaryTree);
86     Emit1(EndOpc);
87     (* endsem *)
88     IF SyIsNot(endSy) THEN Exit;
89     NewSy;
90     IF SyIsNot(periodSy) THEN Exit;
91     NewSy;
92 END; (*MP*)
93
94 PROCEDURE VarDecl;
95 var ok : BOOLEAN;
96 BEGIN
97     IF SyIsNot(varSy) THEN Exit;
98     NewSy;
99     IF SyIsNot(identSy) THEN Exit;
100    (* sem *)
101    DeclVar(identStr, ok);
102    NewSy;
103    WHILE sy = commaSy DO BEGIN
104        NewSy;
105        IF SyIsNot(identSy) THEN Exit;
106        (* sem *)
107        DeclVar(identStr, ok);
108        IF NOT ok THEN
109            SemErr('mult. decl. ');
110        (* endsem *)
111        NewSy;
112    END; (*WHILE*)
113    IF SyIsNot(colonSy) THEN Exit;
114    NewSy;
115    IF SyIsNot(integerSy) THEN Exit;
116    NewSy;
117    IF SyIsNot(semicolonSy) THEN Exit;
118    NewSy;

```



```

119  END; (*VarDecl*)
120
121 PROCEDURE StatSeq(var binaryTree: TreePtr);
122  BEGIN
123    Stat(binaryTree); IF NOT success THEN Exit;
124    WHILE sy = semicolonSy DO BEGIN
125      NewSy;
126      Stat(binaryTree); IF NOT success THEN Exit;
127    END; (*WHILE*)
128  END; (*StatSeq*)
129
130 PROCEDURE Stat(var binaryTree: TreePtr);
131  var destId : STRING;
132      addr, addr1, addr2 : integer;
133  BEGIN
134    CASE sy OF
135      identSy: BEGIN
136        (* sem *)
137        destId := identStr;
138        IF NOT IsDecl(destId) THEN
139          SemErr('var. not decl.')
140        ELSE
141          Emit2(LoadAddrOpc, AddrOf(destId));
142          (* endsem *)
143          NewSy;
144          IF SyIsNot(assignSy) THEN Exit;
145          NewSy;
146          Expr(binaryTree); IF NOT success THEN Exit;
147          (* sem *)
148          IF IsDecl(destId) THEN
149            Emit1(StoreOpc);
150          (* endsem *)
151        END;
152      readSy: BEGIN
153        NewSy;
154        IF SyIsNot(leftParSy) THEN Exit;
155        NewSy;
156        IF SyIsNot(identSy) THEN Exit;
157        (* sem *)
158        IF NOT IsDecl(identStr) THEN
159          SemErr('var not decl.')
160        ELSE BEGIN
161          Emit2(ReadOpc, AddrOf(identStr));
162        END;
163        (* endsem *)
164        NewSy;
165        IF SyIsNot(rightParSy) THEN Exit;
166        NewSy;
167      END;
168      writeSy: BEGIN
169        NewSy;
170        IF SyIsNot(leftParSy) THEN Exit;
171        NewSy;
172        Expr(binaryTree); IF NOT success THEN Exit;
173        (* sem *)
174        Emit1(WriteOpc);
175        (* endsem *)

```

```

176         IF SyIsNot(rightParSy) THEN Exit;
177         NewSy;
178     END;
179     beginSy: BEGIN
180         newSy;
181         StatSeq(binaryTree);
182         IF SyIsNot(endSy) THEN Exit;
183         NewSy;
184     END;
185     ifSy: BEGIN
186         NewSy;
187         IF SyIsNot(identSy) THEN Exit;
188         (* SEM *)
189         IF NOT IsDecl(identStr) THEN BEGIN
190             SemErr('var not decl. ');
191         END;
192         Emit2(LoadValOpc, AddrOf(identStr));
193         Emit2(JmpZOpc, 0); (*0 as dummy address*)
194         addr := CurAddr - 2;
195         (* ENDSEM *)
196         NewSy;
197         IF SyIsNot(thenSy) THEN Exit;
198         newSy;
199         Stat(binaryTree); IF NOT success THEN Exit;
200         IF sy = elseSy THEN BEGIN
201             newSy;
202             (* SEM *)
203             Emit2(JmpOpc, 0);
204             FixUp(addr, CurAddr);
205             addr := CurAddr - 2;
206             (* ENDSEM *)
207             Stat(binaryTree); IF NOT success THEN Exit;
208         END;
209         (* SEM *)
210         FixUp(addr, CurAddr);
211         (* ENDSEM *)
212     END;
213     whileSy: BEGIN
214         NewSy;
215         IF SyIsNot(identSy) THEN Exit;
216         (* SEM *)
217         IF NOT IsDecl(identStr) THEN BEGIN
218             SemErr('var not decl. ');
219         END;
220         addr1 := CurAddr;
221         Emit2(LoadValOpc, AddrOf(identStr));
222         Emit2(JmpZOpc, 0); (*0 as dummy address*)
223         addr2 := CurAddr - 2;
224         (* ENDSEM *)
225         NewSy;
226         IF SyIsNot(doSy) THEN Exit;
227         NewSy;
228         Stat(binaryTree);
229         (* SEM *)
230         Emit2(JmpOpc, addr1);
231         FixUp(addr2, CurAddr);
232         (* ENDSEM *)

```

```

233         END;
234     ELSE
235         ; (*EPS*)
236     END; (*CASE*)
237 END; (*Stat*)
238
239 PROCEDURE Expr(var binaryTree: TreePtr);
240 var t: TreePtr;
241 BEGIN
242     Term(binaryTree); IF NOT success THEN Exit;
243     WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
244         CASE sy OF
245             plusSy: BEGIN
246                 NewSy;
247                 Term(binaryTree^.right); IF NOT success THEN Exit;
248                 (* sem *)
249                 t := newTree('+');
250                 t^.left := binaryTree;
251                 t^.right := newTree('');
252                 binaryTree := t;
253                 (* endsem *)
254             END;
255             minusSy: BEGIN
256                 NewSy;
257                 Term(binaryTree^.right); IF NOT success THEN Exit;
258                 (* sem *)
259                 t := newTree('-');
260                 t^.left := binaryTree;
261                 t^.right := newTree('');
262                 binaryTree := t;
263                 (* endsem *)
264             END;
265         END; (*CASE*)
266     END; (*WHILE*)
267 END; (*Expr*)
268
269 PROCEDURE Term(var binaryTree: TreePtr);
270 var t: TreePtr;
271 BEGIN
272     Fact(binaryTree); IF NOT success THEN Exit;
273     WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
274         CASE sy OF
275             timesSy: BEGIN
276                 NewSy;
277                 Fact(binaryTree^.right); IF NOT success THEN Exit;
278                 (* sem *)
279                 t := newTree('*');
280                 t^.left := binaryTree;
281                 t^.right := newTree('');
282                 binaryTree := t;
283                 (* endsem *)
284             END;
285             divSy: BEGIN
286                 NewSy;
287                 Fact(binaryTree^.right); IF NOT success THEN Exit;
288                 (* sem *)
289                 t := newTree('/');

```

```

290         t^.left := binaryTree;
291         t^.right := newTree('');
292         binaryTree := t;
293         (* endsem *)
294     END;
295     END; (*CASE*)
296     END; (*WHILE*)
297 END; (*Term*)
298
299 PROCEDURE Fact(var binaryTree: TreePtr);
300 var t: TreePtr;
301 BEGIN
302     CASE sy OF
303         identSy: BEGIN
304             (* sem *)
305             t := newTree(identStr);
306             t^.left := binaryTree;
307             t^.right := newTree('');
308             binaryTree := t;
309             (* endsem *)
310             NewSy;
311         END;
312         numberSy: BEGIN
313             (* sem *)
314             t := newTree(numberVal);
315             t^.left := binaryTree;
316             t^.right := newTree('');
317             binaryTree := t;
318             (* endsem *)
319             NewSy;
320         END;
321         leftParSy: BEGIN
322             NewSy;
323             Expr(binaryTree); IF NOT success THEN Exit;
324             IF SyIsNot(rightParSy) THEN Exit;
325             NewSy;
326         END;
327     ELSE
328         success := FALSE;
329     END; (*CASE*)
330 END; (*Fact*)
331
332 PROCEDURE EmitCodeForExprTree(t: TreePtr);
333 var tVal : string;
334     i, code : integer;
335     opChar: char;
336 begin
337     if(t <> NIL) then begin
338         EmitCodeForExprTree(t^.left);
339         tVal := t^.val;
340         val(tVal, i, code);
341         if(code = 0) then
342             Emit2(LoadConstOpc,i)
343         else
344             if(length(tVal) > 1) then begin
345                 IF NOT IsDecl(identStr) THEN
346                     SemErr('var. not decl.')

```

```

347         ELSE
348             Emit2(LoadValOpc,AddrOf(identStr));
349     end else begin
350         opChar := tVal[1];
351         CASE opChar OF
352             '+': BEGIN
353                 Emit1(AddOpc);
354             END;
355             '-': BEGIN
356                 Emit1(SubOpc);
357             END;
358             '*': BEGIN
359                 Emit1(MulOpc);
360             END;
361             '/': BEGIN
362                 Emit1(DivOpc);
363             END;
364             ELSE BEGIN
365                 IF NOT IsDecl(identStr) THEN
366                     SemErr('var. not decl.')
367                 ELSE
368                     Emit2(LoadValOpc,AddrOf(identStr));
369                 END;
370             END;
371         end;
372         EmitCodeForExprTree(t^.right);
373     end;
374 end;
375
376 procedure Simplify(var t: TreePtr);
377 var tVal : string;
378     i, code : integer;
379     opChar: char;
380 begin
381     if(t <> NIL) then begin
382         Simplify(t^.left);
383         Simplify(t^.right);
384         tVal := t^.val;
385         val(tVal, i, code);
386         if((code <> 0) and (length(tVal) = 1)) then begin
387             opChar := tVal[1];
388             CASE opChar OF
389                 '+': BEGIN
390                     if(t^.left^.val = '0') then
391                         t := t^.right
392                     else if(t^.right^.val = '0') then
393                         t := t^.left;
394                     END;
395                 '-': BEGIN
396                     if(t^.left^.val = '0') then
397                         t := t^.right
398                     else if(t^.right^.val = '0') then
399                         t := t^.left;
400                     END;
401                 '*': BEGIN
402                     if(t^.left^.val = '1') then
403                         t := t^.right

```

```

404         else if(t^.right^.val = '1') then
405             t := t^.left;
406         END;
407         '/': BEGIN
408             if(t^.left^.val = '1') then
409                 t := t^.right
410             else if(t^.right^.val = '1') then
411                 t := t^.left;
412             END;
413         ELSE BEGIN
414             IF NOT IsDecl(identStr) THEN
415                 SemErr('var. not decl.')
416             ELSE
417                 Emit2(LoadValOpc, AddrOf(identStr));
418             END;
419         END;
420     end;
421 end;
422 end;
423
424 procedure ConstantFolding(var t: TreePtr);
425 var tValLeft, tValRight : string;
426     valString: string;
427     iLeft, iRight, codeLeft, codeRight : integer;
428     opChar: char;
429 begin
430     if(t <> NIL) then begin
431         valString := '';
432         ConstantFolding(t^.left);
433         ConstantFolding(t^.right);
434         if(t^.left <> NIL) and (t^.right <> NIL) then begin
435             tValLeft := t^.left^.val;
436             tValRight := t^.right^.val;
437             val(tValLeft, iLeft, codeLeft);
438             val(tValRight, iRight, codeRight);
439             if(codeLeft = 0) and (codeRight = 0) then begin
440                 opChar := t^.val[1];
441                 CASE opChar OF
442                     '+': BEGIN
443                         Str(iLeft + iRight, valString);
444                     END;
445                     '-': BEGIN
446                         Str(iLeft - iRight, valString);
447                     END;
448                     '*': BEGIN
449                         Str(iLeft * iRight, valString);
450                     END;
451                     '/': BEGIN
452                         Str(iLeft DIV iRight, valString);
453                     END;
454                 END;
455                 t^.val := valString;
456             end;
457         end;
458     end;
459 end;
460

```

```
461      BEGIN  
462 END. (*MPP_SS*)
```