Abgabetermin: Sa, 08.06.2024

☒ **Gr. 1,** S. Schöberl, MSc **Name** Elias Leonhardsberger **Aufwand in h** 10

☐ **Gr. 2,** DI (FH) G. Horn-Völlenkle, MSc

Punkte _____ **Tutor*in / Übungsleiter*in** ____ / ____

1. MidiPascal (10 Punkte)

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. Folgende Tabelle zeigt zur Verdeutlichung eine Abbildung von MidiPascal auf (vollständiges) Pascal:

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

Mit diesen Spracherweiterungen könnte man dann z. B. ein MidiPascal-Programm schreiben, das für eine 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
`Jmp addr`	Springe an die Codeadresse *addr*
`JmpZ addr`	Hole oberstes Element vom Stapel und wenn dieses 0 (*zero*) ist, springe nach *addr*

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)

`IF x THEN BEGIN`
`  then stats`
`END;`
`...`
	`1 LoadVal x`
	`4 JmpZ 99`
	`... code for then stats`
	
	`99 ...`

`IF x THEN BEGIN`
`  then stats`
	`1 LoadVal x`
	`4 JmpZ 66`
	`... code for then stats`

END ELSE BEGIN *else stats* END; ...	... **Jmp 99** **66** *code for else stats* **99** ...
WHILE x DO BEGIN *while stats* END	**1** LoadVal x 4 **JmpZ 99** ... *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. Sobald ein Ausdruck in Form eines Binärbaums im Hauptspeicher vorliegt, ist es einfach, diesen mittels Baumdurchlauf (in-, pre- oder postorder), wieder in eine Textform (In-, Präoder 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.

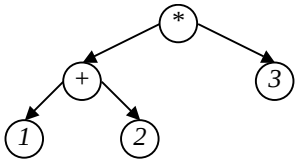
a) Ändern Sie die Erkennungsprozeduren für arithmetische Ausdrücke (*Expr*, *Term* und *Fact*) im Parser Ihres MidiPascal-Compilers so ab, dass vorerst kein Code mehr für die Ausdrücke erzeugt, sondern ein Binärbaum aufgebaut wird, dessen Knoten Zeichenketten enthalten (die vier Operatoren, die Ziffernfolge 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.

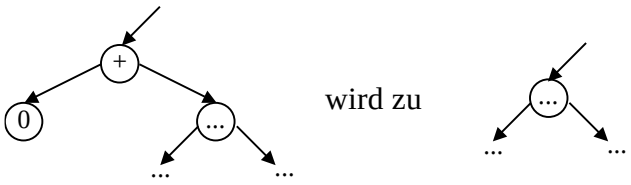
Beispiel: Für den Ausdruck (1 + 2) * 3 soll der links dargestellte Baum aufgebaut werden, und die Prozedur *EmitCodeForExprTree* soll daraus die rechts angegebene Codesequenz erzeugen:



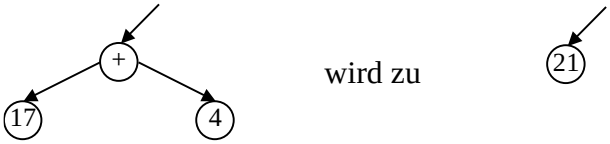

```
LoadConst 1
LoadConst 2
Add
LoadConst 3
Mul
```

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 (kleineren) Bäume werden für die Codegenerierung herangezogen.

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



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



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 dem Knoten 21 entstehen.

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen.
3. Bei Programmen: Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Programm funktioniert, und dass es auch in Fehlersituation entsprechend reagiert.

ADF2/PRO2 UE06

Elias Leonhardsberger

7. Juni 2024, Hagenberg

Inhaltsverzeichnis

1 Arithmetische Ausdrücke und Binärbäume

1.1 Lösungsidee

Für die erste Aufgabe wird MiniPascal aus der Übung kopiert und um die *IF* und *WHILE* Anweisungen erweitert. Die Lösungsidee davon ist in der Angabe schon ziemlich detailliert beschrieben, daher wird nicht mehr näher darauf eingegangen.

MPVM wurde zur Ausführung nach **/.local/bin* verschoben um es über den *$PATH* zu nutzen. MPVM selbst ist nicht in der Abgabe da sich hier nichts geändert hat.

In der ATG der Angabe werden *IF* und *WHILE* nur durch *Idents* gesteuert, da aber Expressions zur Verfügung stehen werden diese verwendet. Dadurch muss man nicht ein *ASSIGN* vor einem *WHILE* machen, wenn eine *Expression* benötigt wird.

Für die Optimierung wird erst der ExpressionParser aus Übung 5 angepasst und dann als Unit im Compiler verwendet, im Compiler selbst wird der ExpressionParser aufgerufen und dann der fertige, optimierte Baum in OpCodes zerlegt.

Der unoptimierte und der optimierte Baum wird beim kompilieren ausgegeben um die Funktionalität zu zeigen.

Die Tests sind in einem shell Script geschrieben und mit folgendem Befehl ausgeführt worden. Die Dateien *OneTwo* und *Five* sind dazu da um die Read Operationen im Skript auszuführen.

```
./TestScript.sh &> ./TestOutput.txt
```

1.2 Souce Code

1.2.1 CodeDef.pas

```
(* MiniPascal/MidiPascal VM op code definitions. *)
(* GHO, 13.05.2017 *)

UNIT CodeDef;

INTERFACE

TYPE
 (* MiniPascal/MidiPascal VM op code. *)
  OpCode = (loadConstOpc, loadValOpc, storeOpc,
            addOpc, subOpc, multOpc, divOpc,
            readOpc, writeOpc,
            endOpc,
            jumpOpc, jumpZeroOpc);

IMPLEMENTATION

END.
```

1.2.2 CodeGen.pas

```
(* Code generator for MiniPascal/MidiPascal compiler. *)
{GHO, 2.5.2011}

UNIT CodeGen;

INTERFACE

USES
CodeDef;

(* Resets code generator. *)
PROCEDURE ResetCodeGenerator;

(* Emits op code. *)
(* IN opC: Op code to emit. *)
PROCEDURE Emit1(opC: OpCode);

(* Emits op code with single argument. *)
(* IN opC: Op code to emit. *)
(* IN arg: Argument to emit. *)
PROCEDURE Emit2(opC: OpCode; arg: INTEGER);

(* Gets current address. *)
```

```
24  (* RETURNS: Current address. *)
25  FUNCTION CurrentAddress: INTEGER;
26
27  (* Fixes jump target. *)
28  (* IN posOfJumpTarget: Position in code where jump address is. *)
29  (* IN addrToJumpTo: Address where jump target should point to. *)
30  PROCEDURE FixUpJumpTarget(posOfJumpTarget, addrToJumpTo: INTEGER);
31
32  (* Writes currently stored bytecode to given file. *)
33  (* REF file: File to write currently stored bytecode to. *)
34  PROCEDURE WriteCodeToFile(VAR outputFile: FILE);
35
36  IMPLEMENTATION
37
38  TYPE
39    CodeArray = ARRAY[1..1] OF BYTE;
40    CodeArrayPtr = ^CodeArray;
41
42  VAR
43    code: CodeArrayPtr;
44    currentCapacity: LONGINT;
45    currentPosition: LONGINT;
46
47  PROCEDURE EmitByte(b: BYTE);
48  VAR
49    newArr: CodeArrayPtr;
50    i: LONGINT;
51  BEGIN
52    IF currentPosition = currentCapacity THEN
53      BEGIN
54        GetMem(newArr, 2 * currentCapacity * SizeOf(BYTE));
55        FOR i := 1 TO currentPosition DO
56     (*$R-*)
57          newArr^[i] := code^[i] (*$R+*);
58        FreeMem(code, currentCapacity * SizeOf(BYTE));
59        code := newArr;
60        currentCapacity := currentCapacity * 2;
61      END;
62    Inc(currentPosition);
63   (*$R-*)
64    code^[currentPosition] := b;
65   (*$R+*)
66  END;
67
68  PROCEDURE EmitWord(w: INTEGER);
69  BEGIN
70    EmitByte(w DIV 256);
```

```
71    EmitByte(w MOD 256);
72  END;
73
74  PROCEDURE ResetCodeGenerator;
75  BEGIN
76    IF code <> NIL THEN
77      BEGIN
78        FreeMem(code, currentCapacity * SizeOf(BYTE));
79        code := NIL;
80      END;
81    currentCapacity := 10;
82    GetMem(code, currentCapacity * SizeOf(BYTE));
83    currentPosition := 0;
84  END;
85
86  PROCEDURE Emit1(opC: OpCode);
87  BEGIN
88    EmitByte(Ord(opC));
89  END;
90
91  PROCEDURE Emit2(opC: OpCode; arg: INTEGER);
92  BEGIN
93    EmitByte(Ord(opC));
94    EmitWord(arg);
95  END;
96
97  FUNCTION CurrentAddress: INTEGER;
98  BEGIN
99    CurrentAddress := currentPosition + 1;
100 END;
101
102 PROCEDURE FixUpJumpTarget(posOfJumpTarget, addrToJumpTo: INTEGER);
103 VAR
104   temp: LONGINT;
105 BEGIN
106   temp := currentPosition;
107   currentPosition := posOfJumpTarget - 1;
108   EmitWord(addrToJumpTo);
109   currentPosition := temp;
110 END;
111
112 PROCEDURE WriteCodeToFile(VAR outputFile: FILE);
113 BEGIN
114   BlockWrite(outputFile, code^, currentPosition);
115 END;
116
117 BEGIN
```

```
118   code := NIL;
119   ResetCodeGenerator;
120 END.
```

1.2.3 ExpressionParser.pas

```
1
2  UNIT ExpressionParser;
3
4  INTERFACE
5
6  USES
7  ExpressionTree;
8
9  PROCEDURE ParseExpression(VAR result: TreePtr; VAR ok: BOOLEAN; VAR
   ↪  errorMessage: STRING);
10
11 IMPLEMENTATION
12
13 USES
14 MpScanner, SymTblC;
15
16 VAR
17   success: BOOLEAN;
18   errMessage: STRING;
19
20 PROCEDURE SemErr(message: STRING);
21 BEGIN (* SemErr *)
22   success := FALSE;
23   errMessage := message;
24 END; (* SemErr *)
25
26 PROCEDURE Expr(VAR e: NodePtr); FORWARD;
27 PROCEDURE Term(VAR t: NodePtr); FORWARD;
28 PROCEDURE Fact(VAR f: NodePtr); FORWARD;
29
30 PROCEDURE ParseExpression(VAR result: TreePtr; VAR ok: BOOLEAN; VAR
   ↪  errorMessage: STRING);
31 BEGIN (* Parse *)
32   success := TRUE;
33   errMessage := '';
34
35   Expr(result);
36
37   ok := success;
38   errorMessage := errMessage;
39 END; (* Parse *)
40
```

```
41 PROCEDURE Expr(VAR e: NodePtr);
42 {local}
43 VAR
44   temp, t: NodePtr;
45   operatorChar: CHAR;
46 {endlocal}
47 BEGIN (* Expr *)
48   Term(e);
49   IF NOT success THEN EXIT;
50
51   WHILE (GetCurrentSymbol() = plusSym) OR (GetCurrentSymbol() = minusSym)
     ↪ DO
52     BEGIN (* WHILE *)
53       CASE GetCurrentSymbol() OF
54         plusSym:
55                   BEGIN (* plusSym *)
56                     ReadNextSymbol();
57                     Term(t);
58                     {sem} operatorChar := '+'; {endsem}
59                     IF NOT success THEN EXIT;
60                   END; (* plusSym *)
61         minusSym:
62                    BEGIN (* minusSym *)
63                      ReadNextSymbol();
64                      Term(t);
65                     {sem} operatorChar := '-'; {endsem}
66                      IF NOT success THEN EXIT;
67                    END; (* minusSym *)
68       END; (* CASE *)
69
70       {sem}
71       temp := NEW(NodePtr);
72
73       IF (temp = NIL) THEN
74         BEGIN (* IF *)
75           SemErr('Out of memory');
76           EXIT;
77         END; (* IF *)
78
79       temp^.val := operatorChar;
80       temp^.left := e;
81       temp^.right := t;
82       temp^.isNumber := FALSE;
83       e := temp;
84       {endsem}
85     END; (* WHILE *)
86 END; (* Expr *)
```

```
87
88  PROCEDURE Term(VAR t: NodePtr);
89  {local}
90  VAR
91    temp, f: NodePtr;
92    operatorChar: CHAR;
93  {endlocal}
94  BEGIN (* Term *)
95    Fact(t);
96    IF NOT success THEN EXIT;
97
98    WHILE (GetCurrentSymbol() = multSym) OR (GetCurrentSymbol() = divSym)
      ↪  DO
99      BEGIN (* WHILE *)
100       CASE GetCurrentSymbol() OF
101         multSym:
102                   BEGIN (* multSym *)
103                     ReadNextSymbol();
104                     Fact(f);
105                     {sem} operatorChar := '*'; {endsem}
106                     IF NOT success THEN EXIT;
107                   END; (* multSym *)
108         divSym:
109                  BEGIN (* divSym *)
110                    ReadNextSymbol();
111                    Fact(f);
112                     {sem} operatorChar := '/'; {endsem}
113                    IF NOT success THEN EXIT;
114                  END; (* divSym *)
115       END; (* CASE *)
116
117       {sem}
118       temp := NEW(NodePtr);
119
120       IF (temp = NIL) THEN
121         BEGIN (* IF *)
122           SemErr('Out of memory');
123           EXIT
124         END; (* IF *)
125
126       temp^.val := operatorChar;
127       temp^.left := t;
128       temp^.right := f;
129       temp^.isNumber := FALSE;
130       t := temp;
131       {endsem}
132     END; (* WHILE *)
```

```
133 END; (* Term *)
134
135 PROCEDURE Fact(VAR f: NodePtr);
136 BEGIN (* Fact *)
137   CASE GetCurrentSymbol() OF
138     numberSym:
139                 BEGIN (* numberSym *)
140                  {sem}
141                   f := NEW(NodePtr);
142
143                   IF (f = NIL) THEN
144                     BEGIN (* IF *)
145                       SemErr('Out of memory');
146                       EXIT;
147                     END; (* IF *)
148
149                   f^.val := GetCurrentNumberString();
150                   f^.left := NIL;
151                   f^.right := NIL;
152                   f^.isNumber := TRUE;
153                  {endsem}
154
155                   ReadNextSymbol();
156                 END; (* numberSym *)
157     identSym:
158                BEGIN (* identSym *)
159                  {sem}
160                  IF NOT IsDeclared(GetCurrentIdentName()) THEN
161                    BEGIN (* IF *)
162                      SemErr('Variable not declared');
163                      EXIT;
164                    END; (* IF *)
165
166                  f := NEW(NodePtr);
167
168                  IF (f = NIL) THEN
169                    BEGIN (* IF *)
170                      SemErr('Out of memory');
171                      EXIT;
172                    END; (* IF *)
173
174                  f^.val := GetCurrentIdentName();
175                  f^.left := NIL;
176                  f^.right := NIL;
177                  f^.isNumber := FALSE;
178                  {endsem}
179
```

```
180                 ReadNextSymbol();
181               END; (* identSym *)
182     leftParSym:
183               BEGIN (* leftParSym *)
184                 ReadNextSymbol();
185                 Expr(f);
186                 IF NOT success THEN EXIT;
187
188                 IF GetCurrentSymbol() <> rightParSym THEN
189                   BEGIN (* IF *)
190                     success := FALSE;
191                     EXIT;
192                   END; (* IF *)
193
194                 ReadNextSymbol();
195               END; (* leftParSym *)
196     ELSE
197       BEGIN (* ELSE *)
198         success := FALSE;
199         EXIT;
200       END; (* ELSE *)
201   END; (* CASE *)
202 END; (* Fact *)
203
204 END.
```

1.2.4 ExpressionTree.pas

```
1
2  UNIT ExpressionTree;
3
4  INTERFACE
5
6  TYPE
7    NodePtr = ^Node;
8    Node = RECORD
9      left, right: NodePtr;
10     val: STRING;
11     isNumber: BOOLEAN;
12   END;
13   TreePtr = NodePtr;
14
15 PROCEDURE DisposeTree(VAR root: TreePtr);
16
17 PROCEDURE OptimizeTree(VAR root: TreePtr; VAR ok: BOOLEAN);
18
19 PROCEDURE PrintTreePostOrder(root: TreePtr);
20
```

```
21 IMPLEMENTATION
22
23 PROCEDURE PrintTreePostOrder(root: TreePtr);
24 BEGIN (* PrintTreePostOrder *)
25   IF (root <> NIL) THEN
26     BEGIN (* IF *)
27       PrintTreePostOrder(root^.left);
28       PrintTreePostOrder(root^.right);
29       Write(root^.val, ' ');
30     END; (* IF *)
31 END; (* PrintTreePostOrder *)
32
33 PROCEDURE DisposeTree(VAR root: TreePtr);
34 BEGIN (* DisposeTree *)
35   IF (root <> NIL) THEN
36     BEGIN (* IF *)
37       DisposeTree(root^.left);
38       DisposeTree(root^.right);
39       DISPOSE(root);
40       root := NIL;
41     END; (* IF *)
42 END; (* DisposeTree *)
43
44 PROCEDURE OptimizeAdd(VAR root: TreePtr);
45 VAR
46   temp: NodePtr;
47   l, r: INTEGER;
48   result: STRING;
49 BEGIN (* OptimizeAdd *)
50   IF (root^.left^.val = '0') THEN
51     BEGIN (* IF *)
52       temp := root^.right;
53       DISPOSE(root^.left);
54       DISPOSE(root);
55       root := temp;
56     END (* IF *)
57   ELSE
58     IF (root^.right^.val = '0') THEN
59       BEGIN (* ELSE IF *)
60         temp := root^.left;
61         DISPOSE(root^.right);
62         DISPOSE(root);
63         root := temp;
64       END (* ELSE IF *)
65   ELSE
66     IF (root^.left^.isNumber) AND (root^.right^.isNumber) THEN
67       BEGIN (* ELSE IF *)
```

```
68          Val(root^.left^.val, l);
69          Val(root^.right^.val, r);
70          Str(l + r, result);
71          root^.val := result;
72          root^.isNumber := TRUE;
73          DISPOSE(root^.left);
74          DISPOSE(root^.right);
75          root^.left := NIL;
76          root^.right := NIL;
77        END (* ELSE IF *)
78  END; (* OptimizeAdd *)
79
80  PROCEDURE OptimizeSub(VAR root: TreePtr);
81  VAR
82    temp: NodePtr;
83    l, r: INTEGER;
84    result: STRING;
85  BEGIN (* OptimizeSub *)
86    IF (root^.right^.val = '0') THEN
87      BEGIN (* IF *)
88        temp := root^.left;
89        DISPOSE(root^.right);
90        DISPOSE(root);
91        root := temp;
92      END (* IF *)
93    ELSE
94      IF (root^.left^.isNumber) AND (root^.right^.isNumber) THEN
95        BEGIN (* ELSE IF *)
96          Val(root^.left^.val, l);
97          Val(root^.right^.val, r);
98          Str(l - r, result);
99          root^.val := result;
100         root^.isNumber := TRUE;
101         DISPOSE(root^.left);
102         DISPOSE(root^.right);
103         root^.left := NIL;
104         root^.right := NIL;
105       END (* ELSE IF *)
106 END; (* OptimizeSub *)
107
108 PROCEDURE OptimizeMul(VAR root: TreePtr);
109 VAR
110   temp: NodePtr;
111   l, r: INTEGER;
112   result: STRING;
113 BEGIN (* OptimizeMul *)
114   IF (root^.left^.val = '0') OR (root^.right^.val = '0') THEN
```

```
115       BEGIN (* IF *)
116         DisposeTree(root^.left);
117         DisposeTree(root^.right);
118         root^.val := '0';
119         root^.isNumber := TRUE;
120       END (* IF *)
121     ELSE
122       IF (root^.left^.val = '1') THEN
123         BEGIN (* ELSE IF *)
124           temp := root^.right;
125           DISPOSE(root^.left);
126           DISPOSE(root);
127           root := temp;
128         END (* ELSE IF *)
129     ELSE
130       IF (root^.right^.val = '1') THEN
131         BEGIN (* ELSE IF *)
132           temp := root^.left;
133           DISPOSE(root^.right);
134           DISPOSE(root);
135           root := temp;
136         END (* ELSE IF *)
137     ELSE
138       IF (root^.left^.isNumber) AND (root^.right^.isNumber) THEN
139         BEGIN (* ELSE IF *)
140           Val(root^.left^.val, l);
141           Val(root^.right^.val, r);
142           Str(l * r, result);
143           root^.val := result;
144           root^.isNumber := TRUE;
145           DISPOSE(root^.left);
146           DISPOSE(root^.right);
147           root^.left := NIL;
148           root^.right := NIL;
149         END (* ELSE IF *)
150 END; (* OptimizeMul *)
151
152 PROCEDURE OptimizeDiv(VAR root: TreePtr; VAR ok: BOOLEAN);
153 VAR
154   temp: NodePtr;
155   l, r: INTEGER;
156   result: STRING;
157 BEGIN (* OptimizeDiv *)
158   IF (root^.right^.val = '0') THEN
159     BEGIN (* IF *)
160       ok := FALSE;
161     END (* IF *)
```

```
162     ELSE
163       IF (root^.left^.val = '0') THEN
164         BEGIN (* ELSE IF *)
165           DISPOSE(root^.left);
166           DisposeTree(root^.right);
167           root^.val := '0';
168           root^.isNumber := TRUE;
169         END (* ELSE IF *)
170     ELSE
171       IF (root^.right^.val = '1') THEN
172         BEGIN (* ELSE IF *)
173           temp := root^.left;
174           DISPOSE(root^.right);
175           DISPOSE(root);
176           root := temp;
177         END (* ELSE IF *)
178     ELSE
179       IF (root^.left^.isNumber) AND (root^.right^.isNumber) THEN
180         BEGIN (* ELSE IF *)
181           Val(root^.left^.val, l);
182           Val(root^.right^.val, r);
183           Str(l DIV r, result);
184           root^.val := result;
185           root^.isNumber := TRUE;
186           DISPOSE(root^.left);
187           DISPOSE(root^.right);
188           root^.left := NIL;
189           root^.right := NIL;
190         END (* ELSE IF *)
191 END; (* OptimizeDiv *)
192
193 PROCEDURE OptimizeTree(VAR root: TreePtr; VAR ok: BOOLEAN);
194 BEGIN (* OptimizeTree *)
195   IF (root <> NIL) THEN
196     BEGIN (* IF *)
197       OptimizeTree(root^.left, ok);
198       OptimizeTree(root^.right, ok);
199
200       IF (ok) THEN
201         BEGIN (* IF *)
202           IF (root^.val = '+') THEN
203             BEGIN (* IF *)
204               OptimizeAdd(root);
205             END (* IF *)
206           ELSE
207             IF (root^.val = '-') THEN
208               BEGIN (* ELSE IF *)
```

```
209                 OptimizeSub(root);
210               END (* ELSE IF *)
211             ELSE
212               IF (root^.val = '*') THEN
213                 BEGIN (* ELSE IF *)
214                   OptimizeMul(root);
215                 END (* ELSE IF *)
216             ELSE
217               IF (root^.val = '/') THEN
218                 BEGIN (* ELSE IF *)
219                   OptimizeDiv(root, ok);
220                 END; (* ELSE IF *)
221           END; (* IF *)
222       END; (* IF *)
223   END; (* OptimizeTree *)
224
225   END. (* ExpressionTree *)
```

1.2.5 MpParser.pas

```
1
2   UNIT MpParser;
3
4   INTERFACE
5
6   USES
7   CodeDef;
8
9   PROCEDURE Parse(VAR inputFile: text; VAR ok: BOOLEAN; VAR errorLine,
    ↪  errorCol: INTEGER; VAR errorMessage: STRING; VAR outputFile: FILE);
10
11  IMPLEMENTATION
12
13  USES
14  MpScanner, SymTblC, CodeGen, ExpressionParser, ExpressionTree;
15
16  VAR
17    success: BOOLEAN;
18    errMessage: STRING;
19
20  PROCEDURE SemErr(message: STRING);
21  BEGIN (* SemErr *)
22    success := FALSE;
23    errMessage := message;
24  END; (* SemErr *)
25
26  PROCEDURE EmitCodeForExprNode(t: TreePtr);
27  VAR
```

```
28    n: INTEGER;
29    errorCode: WORD;
30    errorCodeString: STRING;
31  BEGIN (* EmitCodeForExprNode *)
32    IF (t <> NIL) THEN
33      BEGIN (* IF *)
34        EmitCodeForExprNode(t^.left);
35        EmitCodeForExprNode(t^.right);
36
37        IF (Length(t^.val) = 0) THEN
38          BEGIN (* IF *)
39            SemErr('Invalid Expression Node');
40            EXIT;
41          END; (* IF *)
42
43        IF (t^.isNumber) THEN
44          BEGIN (* IF *)
45            Val(t^.val, n, errorCode);
46            IF (errorCode <> 0) THEN
47              BEGIN (* IF *)
48                Str(errorCode, errorCodeString);
49                SemErr(Concat('Invalid Number ', t^.val, ' (Error Code: ',
                   ↪  errorCodeString, ')'));
50                EXIT;
51              END; (* IF *)
52            Emit2(LoadConstOpc, n);
53          END (* IF *)
54        ELSE
55          IF (t^.val = '+') THEN
56            BEGIN (* ELSE IF *)
57              Emit1(AddOpc)
58            END (* ELSE IF *)
59        ELSE
60          IF (t^.val = '-') THEN
61            BEGIN (* ELSE IF *)
62              Emit1(SubOpc)
63            END (* ELSE IF *)
64        ELSE
65          IF (t^.val = '*') THEN
66            BEGIN (* ELSE IF *)
67              Emit1(MultOpc)
68            END (* ELSE IF *)
69        ELSE
70          IF (t^.val = '/') THEN
71            BEGIN (* ELSE IF *)
72              Emit1(DivOpc)
73            END (* ELSE IF *)
```

```
74        ELSE
75          BEGIN (* ELSE *)
76            Emit2(LoadValOpc, AddressOF(t^.val));
77          END (* ELSE *)
78      END; (* IF *)
79  END; (* EmitCodeForExprNode *)
80
81  PROCEDURE EmitCodeForExprTree(t: TreePtr);
82  VAR
83    ok: BOOLEAN;
84  BEGIN (* EmitCodeForExprTree *)
85    WriteLn('Unoptimized Expression Tree:');
86    PrintTreePostOrder(t);
87    WriteLn();
88    OptimizeTree(t, ok);
89    WriteLn('Optimized Expression Tree:');
90    PrintTreePostOrder(t);
91    WriteLn();
92    WriteLn();
93
94    IF (NOT ok) THEN
95      BEGIN (* IF *)
96        SemErr('Division by zero in expression.');
97        DisposeTree(t);
98        EXIT;
99      END; (* IF *)
100
101   EmitCodeForExprNode(t);
102   DisposeTree(t);
103 END; (* EmitCodeForExprTree *)
104
105 PROCEDURE MP(VAR outputFile: FILE); FORWARD;
106 PROCEDURE VarBlock; FORWARD;
107 PROCEDURE Variable; FORWARD;
108 PROCEDURE StatementSeq; FORWARD;
109 PROCEDURE Statement; FORWARD;
110
111 PROCEDURE Parse(VAR inputFile: text; VAR ok: BOOLEAN; VAR errorLine,
    ↪  errorCol: INTEGER; VAR errorMessage: STRING; VAR outputFile: FILE);
112 BEGIN (* Parse *)
113   success := TRUE;
114   errMessage := '';
115
116   InitScanner(inputFile);
117   MP(outputFile);
118
119   ok := success;
```

```
120   GetCurrentSymbolPosition(errorLine, errorCol);
121   errorMessage := errMessage;
122 END; (* Parse *)
123
124 PROCEDURE MP(VAR outputFile: FILE);
125 BEGIN (* MP *)
126   IF GetCurrentSymbol() <> programSym THEN
127     BEGIN (* IF *)
128       success := FALSE;
129       EXIT;
130     END; (* IF *)
131
132   {sem}
133   ResetSymbolTable();
134   ResetCodeGenerator();
135   {endsem}
136
137   ReadNextSymbol();
138
139   IF GetCurrentSymbol() <> identSym THEN
140     BEGIN (* IF *)
141       success := FALSE;
142       EXIT;
143     END; (* IF *)
144
145   ReadNextSymbol();
146
147   IF GetCurrentSymbol() <> semicolonSym THEN
148     BEGIN (* IF *)
149       success := FALSE;
150       EXIT;
151     END; (* IF *)
152
153   ReadNextSymbol();
154
155   VarBlock();
156   IF NOT success THEN EXIT;
157
158   IF GetCurrentSymbol() <> beginSym THEN
159     BEGIN (* IF *)
160       success := FALSE;
161       EXIT;
162     END; (* IF *)
163
164   ReadNextSymbol();
165
166   StatementSeq();
```

```
167     IF NOT success THEN EXIT;
168
169     IF GetCurrentSymbol() <> endSym THEN
170       BEGIN (* IF *)
171         success := FALSE;
172         EXIT;
173       END; (* IF *)
174
175     ReadNextSymbol();
176
177     IF GetCurrentSymbol() <> periodSym THEN
178       BEGIN (* IF *)
179         success := FALSE;
180         EXIT;
181       END; (* IF *)
182
183     ReadNextSymbol();
184
185     {sem}
186     Emit1(endOpc);
187     WriteCodeToFile(outputFile);
188     {endsem}
189   END; (* MP *)
190
191   PROCEDURE VarBlock;
192   BEGIN (* VarBlock *)
193     IF GetCurrentSymbol() <> varSym THEN
194       BEGIN (* IF *)
195         EXIT;
196       END; (* IF *)
197
198     ReadNextSymbol();
199
200     Variable();
201     IF NOT success THEN EXIT;
202
203     WHILE GetCurrentSymbol() = identSym DO
204       BEGIN (* WHILE *)
205         ReadNextSymbol();
206         Variable();
207         IF NOT success THEN EXIT;
208       END; (* WHILE *)
209   END; (* VarBlock *)
210
211   PROCEDURE Variable;
212   VAR
213   {local} ok : BOOLEAN;{endlocal}
```

```
214 BEGIN (* Variable *)
215   IF GetCurrentSymbol() <> identSym THEN
216     BEGIN (* IF *)
217       success := FALSE;
218       EXIT;
219     END; (* IF *)
220
221   {sem}
222   DeclareVar(GetCurrentIdentName(), ok);
223   IF NOT ok THEN
224     BEGIN (* IF *)
225       SemErr('Variable already declared');
226       EXIT;
227     END; (* IF *)
228   {endsem}
229
230   ReadNextSymbol();
231
232   WHILE GetCurrentSymbol() = commaSym DO
233     BEGIN (* WHILE *)
234       ReadNextSymbol();
235       IF GetCurrentSymbol() <> identSym THEN
236         BEGIN (* IF *)
237           success := FALSE;
238           EXIT;
239         END; (* IF *)
240
241       {sem}
242       DeclareVar(GetCurrentIdentName(), ok);
243       IF NOT ok THEN
244         BEGIN (* IF *)
245           SemErr('Variable already declared');
246           EXIT;
247         END; (* IF *)
248       {endsem}
249
250       ReadNextSymbol();
251     END; (* WHILE *)
252
253   IF (GetCurrentSymbol() <> colonSym) THEN
254     BEGIN (* IF *)
255       success := FALSE;
256       EXIT;
257     END; (* IF *)
258
259   ReadNextSymbol();
260
```

```
261   IF (GetCurrentSymbol() <> integerSym) THEN
262     BEGIN (* IF *)
263       success := FALSE;
264       EXIT;
265     END; (* IF *)
266
267   ReadNextSymbol();
268
269   IF GetCurrentSymbol() <> semicolonSym THEN
270     BEGIN (* IF *)
271       success := FALSE;
272       EXIT;
273     END; (* IF *)
274
275   ReadNextSymbol();
276 END; (* Variable *)
277
278 PROCEDURE StatementSeq;
279 BEGIN (* StatementSeq *)
280   Statement();
281   IF NOT success THEN EXIT;
282
283   WHILE GetCurrentSymbol() = semicolonSym DO
284     BEGIN (* WHILE *)
285       ReadNextSymbol();
286       Statement();
287       IF NOT success THEN EXIT;
288     END; (* WHILE *)
289 END; (* StatementSeq *)
290
291 PROCEDURE Statement;
292 {local}
293 VAR
294   identName: STRING;
295   addr1, addr2: INTEGER;
296   exprTree: TreePtr;
297 {endlocal}
298 BEGIN (* Statement *)
299   CASE GetCurrentSymbol() OF
300     identSym:
301               BEGIN (* identSym *)
302                 {sem}
303                 identName := GetCurrentIdentName();
304
305                 IF NOT IsDeclared(identName) THEN
306                   BEGIN (* IF *)
307                     SemErr('Variable not declared');
```

```
308                     EXIT;
309                   END; (* IF *)
310                 {endsem}
311
312                 ReadNextSymbol();
313
314                 IF GetCurrentSymbol() <> assignSym THEN
315                   BEGIN (* IF *)
316                     success := FALSE;
317                     EXIT;
318                   END; (* IF *)
319
320                 ReadNextSymbol();
321                 ParseExpression(exprTree, success, errMessage);
322                 IF NOT success THEN EXIT;
323
324                 {sem}
325                 EmitCodeForExprTree(exprTree);
326                 IF NOT success THEN EXIT;
327                 Emit2(storeOpc, AddressOF(identName));
328                 {endsem}
329               END; (* identSym *)
330     readSym:
331             BEGIN (* readSym *)
332               ReadNextSymbol();
333
334               IF GetCurrentSymbol() <> leftParSym THEN
335                 BEGIN (* IF *)
336                   success := FALSE;
337                   EXIT;
338                 END; (* IF *)
339
340               ReadNextSymbol();
341
342               IF GetCurrentSymbol() <> identSym THEN
343                 BEGIN (* IF *)
344                   success := FALSE;
345                   EXIT;
346                 END; (* IF *)
347
348               {sem}
349               identName := GetCurrentIdentName();
350
351               IF NOT IsDeclared(identName) THEN
352                 BEGIN (* IF *)
353                   SemErr('Variable not declared');
354                   EXIT;
```

```
355                   END; (* IF *)
356
357                  Emit2(readOpc, AddressOF(identName));
358                 {endsem}
359
360                  ReadNextSymbol();
361
362                  IF GetCurrentSymbol() <> rightParSym THEN
363                    BEGIN (* IF *)
364                      success := FALSE;
365                      EXIT;
366                    END; (* IF *)
367
368                  ReadNextSymbol();
369                END; (* readSym *)
370       writeSym:
371                BEGIN (* writeSym *)
372                  ReadNextSymbol();
373
374                  IF GetCurrentSymbol() <> leftParSym THEN
375                    BEGIN (* IF *)
376                      success := FALSE;
377                      EXIT;
378                    END; (* IF *)
379
380                  ReadNextSymbol();
381                  ParseExpression(exprTree, success, errMessage);
382                  IF NOT success THEN EXIT;
383
384                  {sem}
385                  EmitCodeForExprTree(exprTree);
386                  IF NOT success THEN EXIT;
387                  Emit1(writeOpc);
388                  {endsem}
389
390                  IF GetCurrentSymbol() <> rightParSym THEN
391                    BEGIN (* IF *)
392                      success := FALSE;
393                      EXIT;
394                    END; (* IF *)
395
396                  ReadNextSymbol();
397                END; (* writeSym *)
398       beginSym:
399                BEGIN (* beginSym *)
400                  ReadNextSymbol();
401                  StatementSeq();
```

```
402                 IF NOT success THEN EXIT;
403
404                 IF GetCurrentSymbol() <> endSym THEN
405                   BEGIN (* IF *)
406                     success := FALSE;
407                     EXIT;
408                   END; (* IF *)
409
410                 ReadNextSymbol();
411               END; (* beginSym *)
412     ifSym:
413           BEGIN (* ifSym *)
414             ReadNextSymbol();
415             ParseExpression(exprTree, success, errMessage);
416             IF NOT success THEN EXIT;
417
418            {sem}
419             EmitCodeForExprTree(exprTree);
420             IF NOT success THEN EXIT;
421             Emit2(JumpZeroOpc, 0);
422             addr1 := CurrentAddress() - 2;
423            {endsem}
424
425             IF GetCurrentSymbol() <> thenSym THEN
426               BEGIN (* IF *)
427                 success := FALSE;
428                 EXIT;
429               END; (* IF *)
430
431             ReadNextSymbol();
432             Statement();
433             IF NOT success THEN EXIT;
434
435             IF GetCurrentSymbol() = elseSym THEN
436               BEGIN (* IF *)
437                {sem}
438                 Emit2(JumpZeroOpc, 0);
439                 FixUpJumpTarget(addr1, CurrentAddress());
440                 addr1 := CurrentAddress() - 2;
441                {endsem}
442
443                 ReadNextSymbol();
444                 Statement();
445                 IF NOT success THEN EXIT;
446               END; (* IF *)
447
448            {sem}FixUpJumpTarget(addr1, CurrentAddress());{endsem}
```

```
449             END; (* ifSym *)
450     whileSym:
451               BEGIN (* whileSym *)
452                 ReadNextSymbol();
453                 {sem}addr1 := CurrentAddress();{endsem}
454                 ParseExpression(exprTree, success, errMessage);
455                 IF NOT success THEN EXIT;
456
457                 {sem}
458                 EmitCodeForExprTree(exprTree);
459                 IF NOT success THEN EXIT;
460                 Emit2(JumpZeroOpc, 0);
461                 addr2 := CurrentAddress() - 2;
462                 {endsem}
463
464                 IF GetCurrentSymbol() <> doSym THEN
465                   BEGIN (* IF *)
466                     success := FALSE;
467                     EXIT;
468                   END; (* IF *)
469
470                 ReadNextSymbol();
471                 Statement();
472
473                 {sem}
474                 Emit2(JumpOpc, addr1);
475                 FixUpJumpTarget(addr2, CurrentAddress());
476                 {endsem}
477               END; (* whileSym *)
478     ELSE
479       BEGIN (* ELSE *)
480         EXIT;
481       END; (* ELSE *)
482   END; (* CASE *)
483 END; (* Statement *)
484
485 END.
```

1.2.6 MpScanner.pas

```
1
2  UNIT MpScanner;
3
4  INTERFACE
5
6  TYPE
7    Symbol = (
8               noSym,
```

```
9               beginSym, endSym, integerSym, readSym, writeSym, varSym,
                ↪  programSym,
10              plusSym, minusSym, multSym, divSym,
11              leftParSym, rightParSym,
12              commaSym, semicolonSym, colonSym, assignSym, periodSym,
13              numberSym, identSym,
14              ifSym, elseSym, thenSym, whileSym, doSym
15             );
16
17  PROCEDURE InitScanner(VAR inputFile: Text);
18
19  PROCEDURE ReadNextSymbol;
20
21  FUNCTION GetCurrentSymbol: Symbol;
22
23  PROCEDURE GetCurrentSymbolPosition(VAR line, col: INTEGER);
24
25  FUNCTION GetCurrentNumberValue: INTEGER;
26
27  FUNCTION GetCurrentNumberString: STRING;
28
29  FUNCTION GetCurrentIdentName: STRING;
30
31  IMPLEMENTATION
32
33  CONST
34    Tab = CHR(9);
35    LF = CHR(10);
36    CR = CHR(13);
37    Space = ' ';
38
39  VAR
40    currentSymbol: Symbol;
41    currentLine, currentCol, symbolLine, symbolCol, currentNumberValue:
      ↪  INTEGER;
42    CurrentIdentName: STRING;
43    currentChar: CHAR;
44    inFile: Text;
45
46  PROCEDURE ReadNextChar;
47  BEGIN (* ReadNextChar *)
48    Read(inFile, currentChar);
49
50    IF (currentChar = LF) THEN
51      BEGIN (* IF *)
52        currentLine := currentLine + 1;
53        currentCol := 0;
```

```
54      END (* IF *)
55    ELSE
56      BEGIN (* ELSE *)
57        currentCol := currentCol + 1;
58      END; (* ELSE *)
59  END; (* ReadNextChar *)
60
61  PROCEDURE InitScanner(VAR inputFile: Text);
62  BEGIN (* InitScanner *)
63    inFile := inputFile;
64    currentLine := 1;
65    currentCol := 0;
66    ReadNextChar();
67    ReadNextSymbol();
68  END; (* InitScanner *)
69
70  FUNCTION GetKeyword(s: STRING): Symbol;
71  BEGIN (* GetKeyword *)
72    IF s = 'begin' THEN
73      GetKeyword := beginSym
74    ELSE IF s = 'end' THEN
75            GetKeyword := endSym
76    ELSE IF s = 'integer' THEN
77            GetKeyword := integerSym
78    ELSE IF s = 'read' THEN
79            GetKeyword := readSym
80    ELSE IF s = 'write' THEN
81            GetKeyword := writeSym
82    ELSE IF s = 'var' THEN
83            GetKeyword := varSym
84    ELSE IF s = 'program' THEN
85            GetKeyword := programSym
86    ELSE IF s = 'if' THEN
87            GetKeyword := ifSym
88    ELSE IF s = 'then' THEN
89            GetKeyword := thenSym
90    ELSE IF s = 'while' THEN
91            GetKeyword := whileSym
92    ELSE IF s = 'do' THEN
93            GetKeyword := doSym
94    ELSE IF s = 'else' THEN
95            GetKeyword := elseSym
96    ELSE
97      GetKeyword := identSym;
98  END; (* GetKeyword *)
99
100 PROCEDURE ReadNextSymbol;
```

```
101 BEGIN (* ReadNextSymbol *)
102   WHILE (currentChar = Space) OR (currentChar = LF) OR (currentChar = CR)
      ↪  OR (currentChar = Tab) DO
103     BEGIN (* WHILE *)
104       ReadNextChar();
105     END; (* WHILE *)
106
107   symbolLine := currentLine;
108   symbolCol := currentCol;
109
110   CASE currentChar OF
111     '+':
112          BEGIN (* + *)
113            currentSymbol := plusSym;
114            ReadNextChar();
115          END; (* + *)
116     '-':
117          BEGIN (* - *)
118            currentSymbol := minusSym;
119            ReadNextChar();
120          END; (* - *)
121     '*':
122          BEGIN (* * *)
123            currentSymbol := multSym;
124            ReadNextChar();
125          END; (* * *)
126     '/':
127          BEGIN (* / *)
128            currentSymbol := divSym;
129            ReadNextChar();
130          END; (* / *)
131     '(':
132          BEGIN (* ( *)
133            currentSymbol := leftParSym;
134            ReadNextChar();
135          END; (* ( *)
136     ')':
137          BEGIN (* ) *)
138            currentSymbol := rightParSym;
139            ReadNextChar();
140          END; (* ) *)
141     ',':
142          BEGIN (* , *)
143            currentSymbol := commaSym;
144            ReadNextChar();
145          END; (* , *)
146     ':':
```

```
147             BEGIN (* : *)
148               ReadNextChar();
149               IF currentChar = '=' THEN
150                 BEGIN (* IF *)
151                   currentSymbol := assignSym;
152                   ReadNextChar();
153                 END (* IF *)
154               ELSE
155                 BEGIN (* ELSE *)
156                   currentSymbol := colonSym;
157                 END; (* ELSE *)
158             END; (* : *)
159         ';':
160             BEGIN (* ; *)
161               currentSymbol := semicolonSym;
162               ReadNextChar();
163             END; (* ; *)
164         '.':
165             BEGIN (* . *)
166               currentSymbol := periodSym;
167               ReadNextChar();
168             END; (* . *)
169         '0'..'9':
170                 BEGIN (* 0..9 *)
171                   currentSymbol := numberSym;
172                   currentNumberValue := 0;
173
174                   WHILE currentChar IN ['0' .. '9'] DO
175                     BEGIN (* WHILE *)
176                       currentNumberValue := currentNumberValue * 10 +
                          ↪  Ord(currentChar) - Ord('0');
177                       ReadNextChar();
178                     END; (* WHILE *)
179                 END; (* 0..9 *)
180         'a'..'z', 'A'..'Z', '_':
181                                 BEGIN (* a..z, A..Z, _ *)
182                                   CurrentIdentName := '';
183
184                                   WHILE currentChar IN ['a' .. 'z', 'A' ..
                                      ↪  'Z', '0' .. '9', '_'] DO
185                                     BEGIN (* WHILE *)
186                                       CurrentIdentName := CurrentIdentName +
                                          ↪  LowerCase(currentChar);
187                                       ReadNextChar();
188                                     END; (* WHILE *)
189
```

```
190                                   currentSymbol :=
                                       ↪  GetKeyword(CurrentIdentName);
191                                 END; (* a..z, A..Z, _ *)
192       ELSE
193         BEGIN (* ELSE *)
194           currentSymbol := noSym;
195         END; (* ELSE *)
196     END; (* CASE *)
197   END; (* ReadNextSymbol *)
198
199   FUNCTION GetCurrentSymbol: Symbol;
200   BEGIN (* GetCurrentSymbol *)
201     GetCurrentSymbol := currentSymbol;
202   END; (* GetCurrentSymbol *)
203
204   PROCEDURE GetCurrentSymbolPosition(VAR line, col: INTEGER);
205   BEGIN (* GetCurrentSymbolPosition *)
206     line := symbolLine;
207     col := symbolCol;
208   END; (* GetCurrentSymbolPosition *)
209
210   FUNCTION GetCurrentNumberValue: INTEGER;
211   BEGIN (* GetCurrentNumberValue *)
212     IF (currentSymbol <> numberSym) THEN
213       BEGIN (* IF *)
214         WriteLn('Error: Current symbol is not a number');
215         Halt(1);
216       END; (* IF *)
217
218     GetCurrentNumberValue := currentNumberValue;
219   END; (* GetCurrentNumberValue *)
220
221   FUNCTION GetCurrentNumberString: STRING;
222   VAR
223     result: STRING;
224   BEGIN (* GetCurrentNumberString *)
225     IF (currentSymbol <> numberSym) THEN
226       BEGIN (* IF *)
227         WriteLn('Error: Current symbol is not a number');
228         Halt(1);
229       END; (* IF *)
230
231     Str(currentNumberValue, result);
232     GetCurrentNumberString := result;
233   END; (* GetCurrentNumberString *)
234
235   FUNCTION GetCurrentIdentName: STRING;
```

```
236 BEGIN (* GetCurrentIdentName *)
237   IF currentSymbol <> identSym THEN
238     BEGIN (* IF *)
239       WriteLn('Error: Current symbol is not an identifier');
240       Halt(1);
241     END; (* IF *)
242
243   GetCurrentIdentName := CurrentIdentName;
244 END; (* GetCurrentIdentName *)
245
246 END.
```

1.2.7 SymTblC.pas

```
1  (* Symbol table for MiniPascal/MidiPascal compiler. *)
2  (* GHO, 13.05.2017 *)
3  UNIT SymTblC;
4
5  INTERFACE
6
7  (* Resets symbol table clearing all declared variables. *)
8  PROCEDURE ResetSymbolTable;
9
10 (* Declares new variable. *)
11 (* IN name: Name of variable to declare. *)
12 (* OUT ok: True if declaration was successful, false if variable has
   ↪  already been declared before. *)
13 PROCEDURE DeclareVar(NAME: STRING; VAR ok: BOOLEAN);
14
15 (* Checks if variable is declared. *)
16 (* IN name: Name of variable to check. *)
17 (* RETURNS: True if variable is declared. *)
18 FUNCTION IsDeclared(NAME: STRING): BOOLEAN;
19
20 (* Gets address of variable. *)
21 (* IN name: Name of variable to get address of. *)
22 (* RETURNS: Address of variable. *)
23 FUNCTION AddressOf(NAME: STRING): INTEGER;
24
25 IMPLEMENTATION
26
27 TYPE
28     Variable = ^VariableRec;
29     VariableRec = RECORD
30         name: STRING;
31         address: INTEGER;
32         next: Variable;
33     END;
```

```
34      VariableList = Variable;
35
36  VAR
37      variables: VariableList;
38      nextAddress: INTEGER;
39
40  FUNCTION LookUp(NAME: STRING): INTEGER;
41  VAR
42      found: BOOLEAN;
43      v: Variable;
44  BEGIN
45      LookUp := 0;
46      found := FALSE;
47      v := variables;
48      WHILE (NOT found) AND (v <> NIL) DO BEGIN
49          IF v^.name = name THEN BEGIN
50              LookUp := v^.address;
51              found := TRUE;
52          END;
53          v := v^.next;
54      END;
55  END;
56
57  PROCEDURE ResetSymbolTable;
58  VAR
59      next: Variable;
60  BEGIN
61      WHILE variables <> NIL DO BEGIN
62          next := variables^.next;
63          DISPOSE(variables);
64          variables := next;
65      END;
66      nextAddress := 1;
67  END;
68
69  PROCEDURE DeclareVar(NAME: STRING; VAR ok: BOOLEAN);
70  VAR
71      v: Variable;
72  BEGIN
73      IF IsDeclared(name) THEN
74          ok := FALSE
75      ELSE BEGIN
76          NEW(v);
77          v^.name := name;
78          v^.address := nextAddress;
79          v^.next := variables;
80          variables := v;
```

```
81          Inc(nextAddress);
82          ok := TRUE;
83      END;
84  END;
85
86  FUNCTION IsDeclared(NAME: STRING): BOOLEAN;
87  BEGIN
88      IsDeclared := LookUp(name) <> 0;
89  END;
90
91  FUNCTION AddressOf(NAME: STRING): INTEGER;
92  BEGIN
93      AddressOf := LookUp(name);
94  END;
95
96  BEGIN
97      variables := NIL;
98      ResetSymbolTable;
99  END.
```

1.2.8 MPC.pas

```
1   PROGRAM MPC;
2
3   USES
4   MpParser, CodeDef;
5
6   PROCEDURE ShowHelp;
7   BEGIN (* ShowHelp *)
8     WriteLn('Usage: MPC inputFile [outputFile]');
9     WriteLn('   inputFile: the file to be compiled.');
10    WriteLn('   outputFile: the file where the compiled code will be
      ↪  stored. Default inputFile + ''c''.');
11    WriteLn('   --help: display this help and exit.');
12  END; (* ShowHelp *)
13
14  PROCEDURE ProcessParameters(VAR inputFile: TEXT; VAR outputFile: FILE);
15  VAR
16    inputFileName, outputFileName: STRING;
17  BEGIN (* ProcessParameters *)
18    IF (ParamCount < 1) OR (ParamCount > 2) OR (ParamStr(1) = '--help')
      ↪  THEN
19      BEGIN (* IF *)
20        ShowHelp();
21        HALT(1);
22      END; (* IF *)
23
24    inputFileName := ParamStr(1);
```

```
25
26    IF (ParamCount = 1) THEN
27      BEGIN (* IF *)
28        outputFileName := inputFileName + 'c';
29      END (* IF *)
30    ELSE
31      BEGIN (* ELSE *)
32        outputFileName := ParamStr(2);
33      END; (* ELSE *)
34
35    Assign(inputFile, inputFileName);
36    {$I-}
37    Reset(inputFile);
38    {$I+}
39    errorCode := IOResult;
40
41    IF (errorCode <> 0) THEN
42      BEGIN (* IF *)
43        WriteLn('Error opening file ', inputFileName, '. Error code: ',
          ↪  errorCode);
44        HALT(1);
45      END; (* IF *)
46
47    Assign(outputFile, outputFileName);
48    {$I-}
49    Rewrite(outputFile);
50    {$I+}
51    errorCode := IOResult;
52
53    IF (errorCode <> 0) THEN
54      BEGIN (* IF *)
55        WriteLn('Error opening file ', outputFileName, '. Error code: ',
          ↪  errorCode);
56        Close(inputFile);
57        HALT(1);
58      END; (* IF *)
59  END; (* ProcessParameters *)
60
61  VAR
62    ok: BOOLEAN;
63    errorCol, errorLine: INTEGER;
64    errMessage: STRING;
65    inputFile: TEXT;
66    outputFile: FILE;
67  BEGIN (* MPC *)
68    ProcessParameters(inputFile, outputFile);
69
```

```
70    Parse(inputFile, ok, errorLine, errorCol, errMessage, outputFile);
71    Close(inputFile);
72    Close(outputFile);
73
74    IF (ok) THEN
75      BEGIN (* IF *)
76        WriteLn('Compilation completed.')
77      END (* IF *)
78    ELSE
79      IF (errMessage <> '') THEN
80        BEGIN (* ELSE IF *)
81          WriteLn('Semantic Error: ', errMessage, ' at line ', errorLine, '
              ↪  column ', errorCol);
82        END (* ELSE IF *)
83    ELSE
84      BEGIN (* ELSE *)
85        WriteLn('Syntax error at line ', errorLine, ' column ', errorCol);
86      END; (* ELSE *)
87  END. (* MPC *)
```

1.3 Tests

1.3.1 Testskript

```
echo "Invalid Input File Test:"
../bin/MPC /invalidFile.mp
echo

echo "Invalid Output File Test:"
../bin/MPC SVP.mp /invalidFile.mp
echo

echo "Syntax Error Test:"
../bin/MPC SyntaxError.mp
echo

echo "Division by Zero Test:"
../bin/MPC DivisionByZero.mp
echo

echo "SVP Test:"
../bin/MPC SVP.mp
echo "Execution:"
MPVM SVP.mpc < OneTwo
echo

echo "Output File Test:"
../bin/MPC SVP.mp OutputFile.mpc
echo "Execution:"
MPVM OutputFile.mpc < OneTwo
echo

echo "Oneliner Test:"
../bin/MPC Oneliner.mp
echo "Execution:"
MPVM Oneliner.mpc < OneTwo
echo

echo "Factorial Test:"
../bin/MPC Factorial.mp
echo "Execution:"
MPVM Factorial.mpc < Five
echo

echo "Expression Test:"
../bin/MPC Expressions.mp
echo "Execution:"
MPVM Expressions.mpc < OneTwo
```

1.3.2 OneTwo

Listing 1: OneTwo

```
1
2
```

1.3.3 Five

Listing 2: Five

```
5
```

1.3.4 SyntaxError.mp

Listing 3: SyntaxError.mp

```
PROGRAM SVP;

VAR
   a, b, cs: INTEGER;
BEGIN
   READ(a);
   READ(b)
   cs := (a * a) + (b * b);
   WRITE(cs + 0);
END.
```

1.3.5 DivisionByZero.mp

Listing 4: DivisionByZero.mp

```
PROGRAM ExressionTest;

BEGIN
   Write( (4 + 5 + 2)/ (1 * 3 * 15 - 5 * 9));
END.
```

1.3.6 SVP.mp

Listing 5: SVP.mp

```
PROGRAM SVP;

VAR
   a, b, cs: INTEGER;
BEGIN
   READ(a);
   READ(b);
   cs := (a * a) + (b * b);
   WRITE(cs + 0);
END.
```

1.3.7 Oneliner.mp

Listing 6: Oneliner.mp

```
PROGRAM SVP;VAR a,b,c:INTEGER;BEGIN READ(a);READ(b);c:=(a*a)+(b*b);WRITE(c-
```

1.3.8 Factorial.mp

Listing 7: Factorial.mp

```
PROGRAM SVP;

VAR
  f, n: INTEGER;
BEGIN
  READ(n);
  f := n;
  n := n - 1;
  WHILE n DO
  BEGIN
    f := n * f;
    n := n - 1;
  END;
  WRITE(f);
END.
```

1.3.9 Expressions.mp

Listing 8: Expressions.mp

```
PROGRAM ExressionTest;

var
  a, b: INTEGER;
BEGIN
  Read(a);
  Read(b);
  Write( 0 + (17 + 4) * 1 );
  Write((50 / 2 * ((25 - 14) * (1 + 14)
    + 14 * (1 - 25 + 14)))
    + (20 + 2 * 22 - 23));
  Write((50 / 2 * ((25 - 14) * (1 + 14)
    + 14 * (a - 25 + 14)))
    + (20 + b * 22 - 23));
END.
```

1.4 Testergebnisse

1.4.1 Ausgabe des Testskripts

Listing 9: TestOutput.txt

```
Invalid Input File Test:
Error opening file /invalidFile.mp. Error code: 2

Invalid Output File Test:
Error opening file /invalidFile.mp. Error code: 5

Syntax Error Test:
Syntax error at line 8 column 3

Division by Zero Test:
Unoptimized Expression Tree:
4 5 + 2 + 1 3 * 15 * 5 9 * - /
Optimized Expression Tree:
11 0 /

Semantic Error: Division by zero in expression. at line 4 column 43

SVP Test:
Unoptimized Expression Tree:
a a * b b * +
Optimized Expression Tree:
a a * b b * +

Unoptimized Expression Tree:
cs 0 +
Optimized Expression Tree:
cs

Compilation completed.
Execution:
5

Output File Test:
Unoptimized Expression Tree:
a a * b b * +
Optimized Expression Tree:
a a * b b * +

Unoptimized Expression Tree:
cs 0 +
Optimized Expression Tree:
cs
```

```
Compilation completed.
Execution:
5

Oneliner Test:
Unoptimized Expression Tree:
a a * b b * +
Optimized Expression Tree:
a a * b b * +

Unoptimized Expression Tree:
c 0 +
Optimized Expression Tree:
c

Compilation completed.
Execution:
5

Factorial Test:
Unoptimized Expression Tree:
n
Optimized Expression Tree:
n

Unoptimized Expression Tree:
n 1 -
Optimized Expression Tree:
n 1 -

Unoptimized Expression Tree:
n
Optimized Expression Tree:
n

Unoptimized Expression Tree:
n f *
Optimized Expression Tree:
n f *

Unoptimized Expression Tree:
n 1 -
Optimized Expression Tree:
n 1 -

Unoptimized Expression Tree:
```

```
f
Optimized Expression Tree:
f

Compilation completed.
Execution:
120

Expression Test:
Unoptimized Expression Tree:
0 17 4 + 1 * +
Optimized Expression Tree:
21

Unoptimized Expression Tree:
50 2 / 25 14 - 1 14 + * 14 1 25 - 14 + * + * 20 2 22 * + 23 - +
Optimized Expression Tree:
666

Unoptimized Expression Tree:
50 2 / 25 14 - 1 14 + * 14 a 25 - 14 + * + * 20 b 22 * + 23 - +
Optimized Expression Tree:
25 165 14 a 25 - 14 + * + * 20 b 22 * + 23 - +

Compilation completed.
Execution:
21
666
666
```