Abgabetermin: Mi, 10. 05. 2023

☒ **Gr. 1,** Dr. S. Wagner **Name** Elias Wurm **Aufwand in h** 2h 45min
☐ **Gr. 2,** Dr. D. Auer
☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / 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.

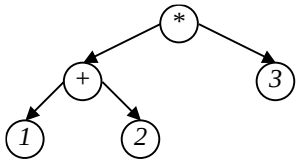
a) Ändern Sie die Erkennungsprozeduren für arithmetische Ausdrücke (*Expr, Term* und *Fact*) im Parser Ihres MidiPascal-Compilers so ab, dass vorerst kein Code mehr für die Ausdrücke erzeugt, sondern ein Binärbaum aufgebaut wird, dessen Knoten Zeichenketten enthalten (die vier Operatoren, die Ziffernfolge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

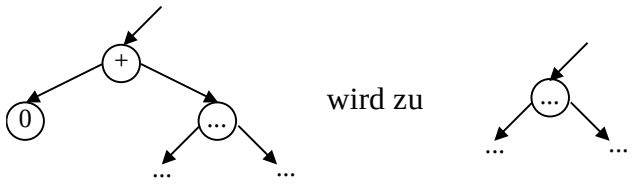
Beispiel: Für den Ausdruck (1 + 2) * 3 soll der links dargestellte Baum aufgebaut werden, und die Prozedur *EmitCodeForExprTree* soll daraus die rechts angegebene Codesequenz erzeugen:



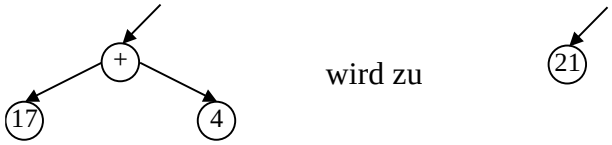

```
LoadConst 1
LoadConst 2
Add
LoadConst 3
Mul
```

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.