ADF 2*x* & PRO 2*x* Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP **SS 21, Übung 5**

Abgabetermin: Mi, 05. 05. 2021

☐ **Gr. 1,** Dr. S. Wagner **Name** Angelos Angelis **Aufwand in h** 10

☒ **Gr. 2,** Dr. D. Auer

☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

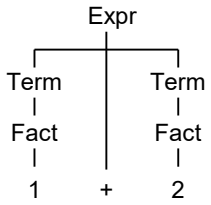
1. Syntaxbäume in kanonischer Form (10 Punkte)

Wie Sie bereits wissen, kann die Syntax einfacher arithmetischer Ausdrücke in Infix-Notation, z. B. (17 + 4) * 21, durch folgende Grammatik beschrieben werden:

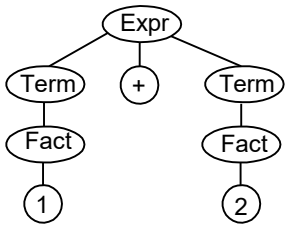
```
Expr  =  Term { '+' Term | '-' Term } .
Term  =  Fact { '*' Fact | '/' Fact } .
Fact  =  number | '(' Expr ')' .
```

Die Struktur von solchen Ausdrücken kann auf Basis obiger Grammatik durch ihren Syntaxbaum dargestellt werden. Folgende Abbildungen zeigen zwei unterschiedliche Darstellungen des Syntaxbaums für den Beispielausdruck 1 + 2:

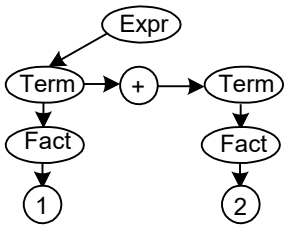
"Konventionelle" Darstellung



Baumdarstellung



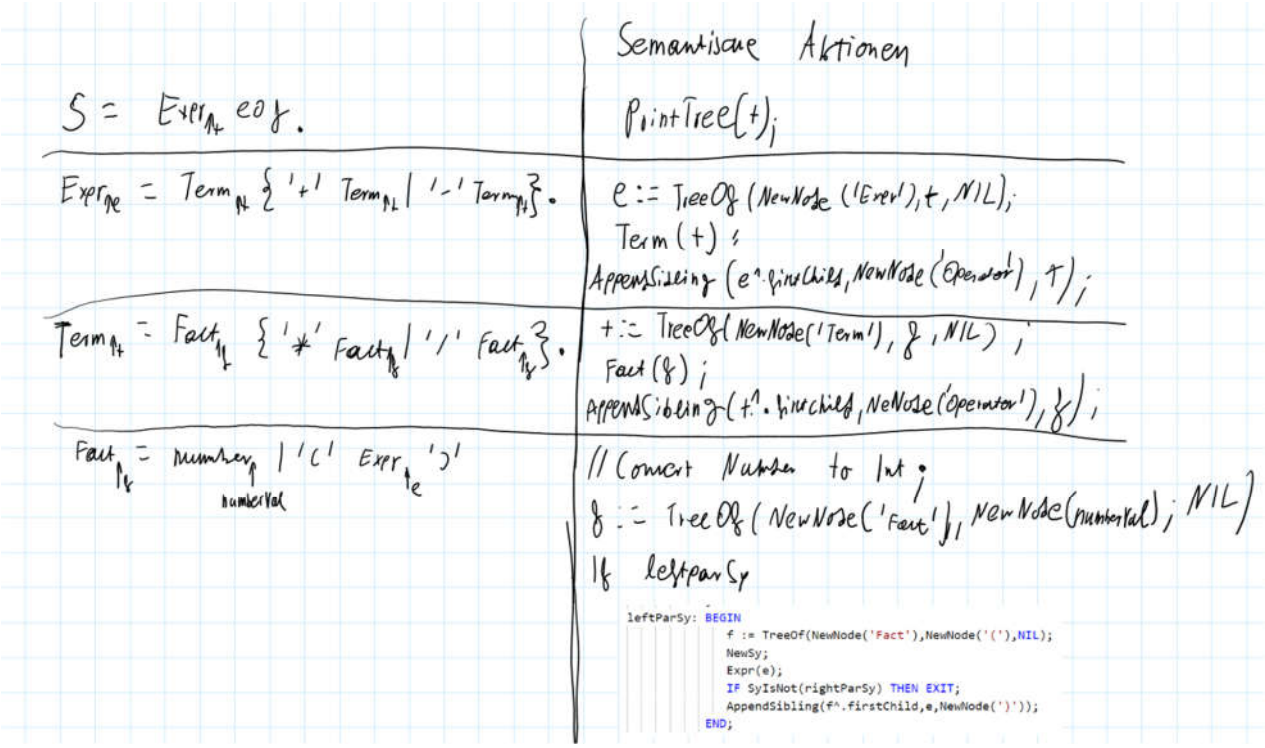
Syntaxbäume sind somit Bäume, deren Knoten beliebig viele Nachfolgeknoten haben können. Will man Syntaxbäume in Form von dynamischen Datenstrukturen abbilden, tritt ein Problem auf: Wie viele Zeiger braucht ein Knoten? Eine einfache Implementierung für solche *allgemeinen Bäume* besteht darin, diese auf einen Spezialfall von *Binärbäumen* zurückzuführen, indem jeder Knoten einen Zeiger auf sein erstes „Kind“ (in der Komponente *firstChild*) und einen Zeiger auf die einfach-verkettete Liste seiner „Geschwister“ (in der Komponente *sibling*) hat. Jeder Knoten kommt dann mit zwei Zeigern aus, unabhängig davon, wie viele Geschwister er hat. Man nennt diese Darstellung *kanonische Form*. Der Syntaxbaum für das obige Beispiel sieht in kanonischer Form wie folgt aus:



Entwickeln Sie aus der oben angegebenen Grammatik eine attributierte Grammatik (ATG), die für arithmetische Ausdrücke den Syntaxbaum in kanonischer Form aufbaut und implementieren Sie diese. Verwenden Sie dazu folgende Deklarationen:

```
TYPE
  NodePtr = ^Node;
  Node = RECORD
    firstChild, sibling: NodePtr;
    val: STRING; (* nonterminal, operator or operand as string *)
  END; (* Node *)
  TreePtr = NodePtr;
```

Lösungsidee:
Attributierte Gramatik:



Meine Idee die Aufgabe zu lösen wäre, dass jeweils jedes Mal wo eine Expression, ein Term oder ein Faktor erkannt wird eine neue Baumstruktur mittels TreeOf erstellt wird und dann letzten Endes alles verbunden wird. Abgesehen davon sind Operanden immer Siblings von der zugehörigen „Grammatik“(+- zu Term; */ zu Faktor). Daher muss jedes Mal wo ein Operand erkannt wird als erstes ein neues Zeichen gelesen werden. Zweitens muss dann bei diesem Zeichen überprüft werden ob es eine Zahl ist oder gar eine neue Expression. Den Datentyp habe ich mittels einer Unit implementiert. Hierbei habe ich die Unit vom ersten Semester genommen und umgeschrieben. Die wichtigste Änderung jedoch ist die AppendSibling Prozedur. Diese nimmt einen Knoten im Baum und fügt zwei neue Knoten hinten als Siblings an.

Testfälle:
Ich habe es nicht geschafft eine Sinnvolle Ausgabe zu Programmieren weshalb es schwer ist zu Überprüfen ob das Programm funktioniert. Jedoch bin ich selber den Code im Kopf des Öfteren durchgegangen und bin zum Punkt gekommen ,dass das Programm Funktionsfähig ist.

Testfall1)

```
expr > 2 + 3
Expr
Term
Fact
3
+
Term
Fact
2
successfull
```

Testfall2)

```
expr > 2 * ( 3 + 4)
Expr
Term
Fact
)
Expr
Term
Fact
4
+
Term
Fact
3
(
*
Fact
2
successfull
```

```
PROGRAM ExprPrg;

  USES ModTree;

  CONST
    eofCh = Chr(0);
    spaceCh = ' ';

  TYPE
    Symbol = (errorSy,eofSy,
              plusSy, minusSy, timesSy, divSy,
              leftParSy, rightParSy,
              number);

    VAR
      line: STRING; //input = arithmetic expr
      ch: CHAR; // current char
      cnr: INTEGER; // char column number
      sy: Symbol; // current Symbol
      numberVal: INTEGER; // holds number value if sy = number
      success: BOOLEAN; // parsing successfull

  PROCEDURE NewCh;
  BEGIN (* NewCh *)
    IF (cnr < Length(line)) THEN BEGIN
      cnr := cnr + 1;
      ch := line[cnr];
    END ELSE BEGIN
      ch := eofCh;
    END;
  END; (* NewCh *)

  PROCEDURE NewSy;
  BEGIN (* NewSy *)
    WHILE (ch = spaceCh) OR (ch = Chr(9)) DO BEGIN
      NewCh;
    END; (* WHILE *)
    CASE ch OF
      eofCh:BEGIN
            sy :=eofSy;
          END;
      '+':BEGIN
            sy := plusSy; NewCh;
```

```
        END;
    '-':BEGIN
          sy := minusSy; NewCh;
        END;
    '*':BEGIN
          sy := timesSy; NewCh;
        END;
    '/':BEGIN
          sy := divSy; NewCh;
        END;
    '(':BEGIN
          sy := leftParSy; NewCh;
        END;
    ')':BEGIN
          sy := rightParSy; NewCh;
        END;
    '0'..'9': BEGIN
                sy:= number;
                numberVal := 0;
                REPEAT
                  numberVal := numberVal * 10 + (Ord(ch) - Ord('0'));
                  NewCh;
                UNTIL (ch < '0') OR (ch > '9'); (* REPEAT *)
              END;
    ELSE BEGIN
      sy := errorSy;
    END;
  END; (* CASE *)
END; (* NewSy *)

//PARSER

FUNCTION SyIsNot(expectedSy: Symbol): BOOLEAN;
BEGIN (* SyIsNot *)
  success := (success) AND (expectedSy = sy);
  SyIsNot := NOT success;
END; (* SyIsNot *)

PROCEDURE S; FORWARD;
PROCEDURE Expr(VAR e: NodePtr) FORWARD;
PROCEDURE Term(VAR t: NodePtr) FORWARD;
PROCEDURE Fact(VAR f: NodePtr) FORWARD;

PROCEDURE S;
 VAR
```

```
  e: TreePtr;
BEGIN (* S *)
  e := InitTree;
  Expr(e); IF NOT success THEN EXIT;
  NewSy;
  PrintTree(e); // SEM
  DisposeTree(e);
  IF (SyIsNot(eofSy)) THEN EXIT;
END; (* S *)

PROCEDURE Expr(VAR e: NodePtr); // Expr = Term { "+" Term | "-" Term } .
  VAR
    t: NodePtr;
BEGIN (* Expr *)
  Term(t); IF NOT success THEN Exit;
  e := TreeOf(NewNode('Expr'), t, NIL);
  WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
    CASE sy OF
      plusSy: BEGIN
                NewSy;
                Term(t); IF NOT success THEN Exit;
                AddSiblings(e^.firstChild,NewNode('+'),t); //SEM
              END;
      minusSy: BEGIN
                NewSy;
                Term(t); IF NOT success THEN Exit;
                AddSiblings(e^.firstChild,NewNode('+'),t); //SEM
               END;
    END;
  END; (* WHILE *)

END; (* Expr *)

PROCEDURE Term(VAR t: NodePtr); // TERM = Fact {* Fact | / FAct}
  VAR f: NodePtr;
BEGIN (* Tert *)
  Fact(f); IF NOT success THEN Exit;
  t := TreeOf(NewNode('Term'), f, NIL);
  WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
    CASE sy OF
      timesSy: BEGIN
                NewSy;
                Fact(f); IF NOT success THEN Exit;
                AddSiblings(t^.firstChild,NewNode('*'),f); //SEM
              END;
```

```
        divSy: BEGIN
                 NewSy;
                 Fact(f); IF NOT success THEN Exit;
                 AddSiblings(t^.firstChild,NewNode('*'),f); //SEM
               END;
      END;
    END; (* WHILE *)

  END; (* Tert *)

  PROCEDURE Fact(VAR f: NodePtr); // Fact = number | (Expr).
    VAR
      e: NodePtr;
      numberStr: STRING;
  BEGIN (* Fact *)
    CASE sy OF
      number: BEGIN
                Str(numberVal,numberStr);
                f := TreeOf(NewNode('Fact'),NewNode(numberStr),NIL); //SEM
                NewSy;
              END;
      leftParSy: BEGIN
                   f := TreeOf(NewNode('Fact'),NewNode('('),NIL);
                   NewSy;
                   Expr(e);
                   IF SyIsNot(rightParSy) THEN EXIT;
                   AddSiblings(f^.firstChild,e,NewNode(')'));
                 END;
      ELSE BEGIN
        success := FALSE;
      END;
    END;
  END; (* Fact *)
  //PARSER

BEGIN (* ExprPrg *)
  Write('expr > ');
  ReadLn(line);
  cnr := 0;
  NewCh;
  NewSy;
  success := TRUE;
  S;
  IF (success) THEN BEGIN
    WriteLn('successfull')
```

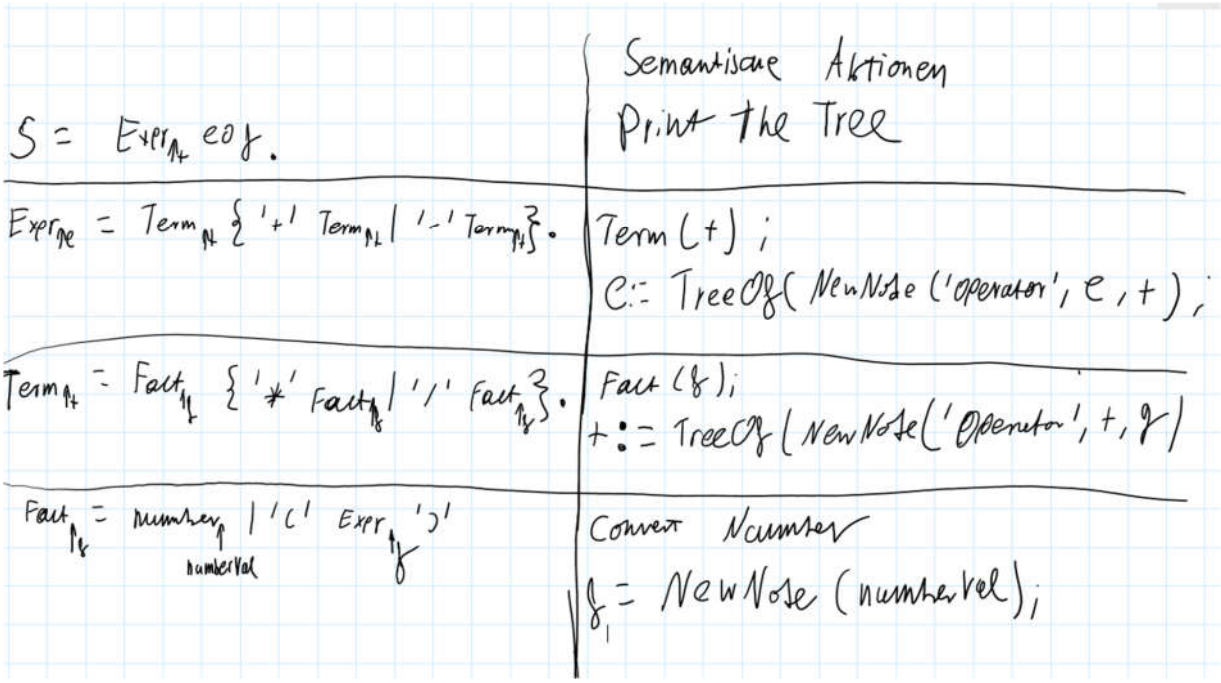
```
  END ELSE BEGIN
    WriteLn('syntax error in: ', cnr);
  END;
END. (* ExprPrg *)


UNIT ModTree;

INTERFACE
  TYPE
    NodePtr = ^Node;
    Node = RECORD
      firstChild, sibling: NodePtr;
      val: STRING;
    END; (* Node *)
    TreePtr = NodePtr;

  FUNCTION InitTree: TreePtr;
  PROCEDURE DisposeTree(VAR t: TreePtr);
  FUNCTION NewNode(value: STRING): NodePtr;
  FUNCTION TreeOf(root: NodePtr; left, right: NodePtr): NodePtr;
  PROCEDURE PrintTree(t: TreePtr);
  PROCEDURE AddSiblings(VAR n: NodePtr; s1, s2: NodePtr);

IMPLEMENTATION

  FUNCTION InitTree: TreePtr;
    VAR t: TreePtr;
  BEGIN (* InitTree *)
    t := NIL;
    InitTree := t;
  END; (* InitTree *)

  PROCEDURE DisposeTree(VAR t: TreePtr);
  BEGIN (* DisposeTree *)
    IF (t <> NIL) THEN BEGIN
      DisposeTree(t^.firstChild);
      DisposeTree(t^.sibling);
      Dispose(t);
      t := NIL;
    END; (* IF *)
  END; (* DisposeTree *)

  FUNCTION NewNode(value: STRING): NodePtr;
    VAR
```

```
    n: NodePtr;
  BEGIN (* NewNode *)
    New(n);
    n^.val := value;
    n^.firstChild := NIL;
    n^.sibling := NIL;
    NewNode := n;
  END; (* NewNode *)

  FUNCTION TreeOf(root: NodePtr; left, right: NodePtr): NodePtr;
  BEGIN (* TreeOf *)
    root^.firstChild := left;
    root^.sibling := right;
    TreeOf := root;
  END; (* TreeOf *)

  PROCEDURE PrintTree(t: TreePtr);
  BEGIN (* PrintTree *)
    IF (t <> NIL) THEN BEGIN
      PrintTree(t^.sibling);
      WriteLn(t^.val);
      PrintTree(t^.firstChild);
    END; (* IF *)
  END; (* PrintTree *)

  PROCEDURE AddSiblings(VAR n: NodePtr; s1, s2: NodePtr);
    VAR
      lastSibling: NodePtr;
  BEGIN (* AddSiblings *)
    lastSibling := n;
    WHILE (lastSibling^.sibling <> NIL) DO BEGIN
      lastSibling := lastSibling^.sibling;
    END; (* WHILE *)
    lastSibling^.sibling := s1;
    s1^.sibling := s2;
  END; (* AddSiblings *)

BEGIN (* ModTree *)

END. (* ModTree *)
```

Aufgabe 2

Lösungsidee:

Attributierte Grammatik:



Die Aufgabe ist relative ähnlich zur ersten Aufgabe nur einfacher, da man sich nicht um die Siblings kümmern muss und da man dieses Mal einen Binärbaum erstellen muss. Ebenfalls wird bei der Aufgabe ein Baum mittels TreeOf erstellt erst wenn ein 2 Zahlen und ein Operand erkannt wurde wobei jedes Mal der Operand der Wurzelknoten ist. Der Datentyp ist ebenfalls eine abgeänderte Unit vom ersten Semester. Daher war auch die Ausgabe in den verschiedensten Varianten zu programmieren kein großer Aufwand. Als letztes muss der konstruierte Baum bzw. die Inhalte davon berechnet werden. Dies habe ich mittels Rekursion implementiert. Hierbei wird jedes Mal Überprüft ob das Aktuelle Zeichen ein Operand ist. Falls ja wird die Entsprechende Operation mit den zwei Unterknoten ausgeführt und dann das Ergebnis ausgegeben.

Anmerkung: Ich weiß nicht wie man in dem Fall die Klammer im Baum einbauen und dann Ausgeben soll deshalb hab ich das Ausgelassen.

Testfälle:

Testfall1)

```
expr > 7 + 3
In-Order:
   7   +   3
Pre-Order:
   +   7   3
Post-Order:
   7   3   +
Value of: 10
successfull
```

Testfall2)

```
expr > 7 * ( 3 +4)
In-Order:
   7   *   3   +   4
Pre-Order:
   *   7   +   3   4
Post-Order:
   7   3   4   +   *
Value of: 49
successfull
```

```
PROGRAM ATGUe2;

  USES ModTreeA2;

  CONST
    eofCh = Chr(0);
    spaceCh = ' ';

  TYPE
    Symbol = (errorSy,eofSy,
              plusSy, minusSy, timesSy, divSy,
              leftParSy, rightParSy,
```

```
            number);

  VAR
    line: STRING; //input = arithmetic expr
    ch: CHAR; // current char
    cnr: INTEGER; // char column number
    sy: Symbol; // current Symbol
    numberVal: INTEGER; // holds number value if sy = number
    success: BOOLEAN; // parsing successfull

PROCEDURE NewCh;
BEGIN (* NewCh *)
  IF (cnr < Length(line)) THEN BEGIN
    cnr := cnr + 1;
    ch := line[cnr];
  END ELSE BEGIN
    ch := eofCh;
  END;
END; (* NewCh *)

PROCEDURE NewSy;
BEGIN (* NewSy *)
  WHILE (ch = spaceCh) OR (ch = Chr(9)) DO BEGIN
    NewCh;
  END; (* WHILE *)
  CASE ch OF
    eofCh:BEGIN
          sy :=eofSy;
        END;
    '+':BEGIN
          sy := plusSy; NewCh;
        END;
    '-':BEGIN
          sy := minusSy; NewCh;
        END;
    '*':BEGIN
          sy := timesSy; NewCh;
        END;
    '/':BEGIN
          sy := divSy; NewCh;
        END;
    '(':BEGIN
          sy := leftParSy; NewCh;
        END;
    ')':BEGIN
```

```
          sy := rightParSy; NewCh;
        END;
    '0'..'9': BEGIN
                sy:= number;
                numberVal := 0;
                REPEAT
                  numberVal := numberVal * 10 + (Ord(ch) - Ord('0'));
                  NewCh;
                UNTIL (ch < '0') OR (ch > '9'); (* REPEAT *)
              END;
    ELSE BEGIN
      sy := errorSy;
    END;
  END; (* CASE *)
END; (* NewSy *)

//PARSER

FUNCTION SyIsNot(expectedSy: Symbol): BOOLEAN;
BEGIN (* SyIsNot *)
  success := (success) AND (expectedSy = sy);
  SyIsNot := NOT success;
END; (* SyIsNot *)

PROCEDURE Expr(VAR e: NodePtr); FORWARD;
PROCEDURE Term(VAR t: NodePtr); FORWARD;
PROCEDURE Fact(VAR f: NodePtr); FORWARD;

PROCEDURE InitParser;
BEGIN (* InitParser *)
  success := TRUE;
  NewSy;
END; (* InitParser *)

PROCEDURE S;
  VAR
    t: TreePtr;
BEGIN (* S *)
  Expr(t); IF NOT success THEN EXIT;
  IF sy <> eofSy THEN BEGIN success := FALSE; EXIT END;
  NewSy;
  (* SEM *)
  WriteLn('In-Order:');PrintInOrder(t);WriteLn;
  WriteLn('Pre-Order:');PrintPreOrder(t);WriteLn;
  WriteLn('Post-Order:');PrintPostOrder(t);WriteLn;
```

```
    WriteLn('Value of: ', ValueOf(t));
    DisposeTree(t);
    (* ENDSEM *)
  END; (* S *)

  (* Expr -> Term { '+' Term | '-' Term }. *)
  PROCEDURE Expr(VAR e: NodePtr);
    VAR
      t: NodePtr;
  BEGIN (* Expr *)
    Term(e); IF NOT success THEN EXIT;
    WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
      CASE sy OF
        plusSy: BEGIN
          NewSy; (* skip + *)
          Term(t); IF NOT success THEN EXIT;
          (* SEM *)
          e := TreeOf(NewNode('+'), e, t);
          (* ENDSEM *)
        END;
        minusSy: BEGIN
          NewSy; (* skip - *)
          Term(t); If NOT success THEN EXIT;
          (* SEM *)
          e := TreeOf(NewNode('-'), e, t);
          (* ENDSEM *)
        END;
      END; (* CASE *)
    END; (* WHILE *)
  END; (* Expr *)

  (* Term -> Fact { '*' Fact | '/' Fact }. *)
  PROCEDURE Term(VAR t: NodePtr);
    VAR
      f: NodePtr;
  BEGIN (* Term *)
    Fact(t); IF NOT success THEN EXIT;
    WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
      CASE sy OF
        timesSy: BEGIN
          NewSy; (* skip * *)
          Fact(f); IF NOT success THEN EXIT;
          (* SEM *)
          t := TreeOf(NewNode('*'), t, f);
          (* ENDSEM *)
```

```
      END;
      divSy: BEGIN
        NewSy; (* skip / *)
        Fact(f); If NOT success THEN EXIT;
        (* SEM *)
        t := TreeOf(NewNode('/'), t, f);
        (* ENDSEM *)
      END;
    END; (* CASE *)
  END; (* WHILE *)
END; (* Term *)

PROCEDURE Fact(VAR f: NodePtr);
  VAR
    e: NodePtr;
    numberStr: STRING;
BEGIN (* Fact *)
  CASE sy OF
    number: BEGIN
      (* SEM *)
      Str(numberVal, numberStr);
      f := NewNode(numberStr);
      (* END SEM *)
      NewSy;
    END;
    leftParSy: BEGIN
                NewSy; (* skip '(' *)
                Expr(e); IF NOT success THEN EXIT;
                IF (sy <> rightParSy) THEN BEGIN
                  success := FALSE;
                  EXIT;
                END; (* IF *)
                f := e;
                NewSy;
               END;
    ELSE BEGIN
      success := FALSE;
      EXIT;
    END;
  END; (* CASE *)
END; (* Fact *)


//PARSER
```

```
BEGIN (* ATGUe2 *)
  Write('expr > ');
  ReadLn(line);
  cnr := 0;
  NewCh;
  NewSy;
  success := TRUE;
  S;

  IF (success) THEN BEGIN
    WriteLn('successfull')
  END ELSE BEGIN
    WriteLn('syntax error in: ', cnr);
  END;
END. (* ATGUe2 *)


UNIT ModTreeA2;

INTERFACE
  TYPE
    NodePtr = ^Node;
    Node = RECORD
      left, right: NodePtr;
      val: STRING;
    END; (* Node *)
    TreePtr = NodePtr;

  PROCEDURE DisposeTree(VAR t: TreePtr);
  FUNCTION NewNode(value: STRING): NodePtr;
  FUNCTION TreeOf(root: NodePtr; left, right: NodePtr): NodePtr;
  PROCEDURE PrintInOrder(t: TreePtr);
  PROCEDURE PrintPostOrder(t: TreePtr);
  PROCEDURE PrintPreOrder(t: TreePtr);
  FUNCTION ValueOf(t: TreePtr): INTEGER;

IMPLEMENTATION

  PROCEDURE DisposeTree(VAR t: TreePtr);
  BEGIN (* DisposeTree *)
    IF (t <> NIL) THEN BEGIN
      DisposeTree(t^.left);
      DisposeTree(t^.right);
      Dispose(t);
      t := NIL;
```

```
    END; (* IF *)
  END; (* DisposeTree *)

  FUNCTION NewNode(value: STRING): NodePtr;
    VAR
      n: NodePtr;
  BEGIN (* NewNode *)
    New(n);
    n^.val := value;
    n^.left := NIL;
    n^.right := NIL;
    NewNode := n;
  END; (* NewNode *)

  FUNCTION TreeOf(root: NodePtr; left, right: NodePtr): NodePtr;
  BEGIN (* TreeOf *)
    root^.left := left;
    root^.right := right;
    TreeOf := root;
  END; (* TreeOf *)

  PROCEDURE PrintInOrder(t: TreePtr);
  BEGIN (* PrintInOrder *)
    IF (t <> NIL) THEN BEGIN
      PrintInOrder(t^.left);
      Write(t^.val:4);
      PrintInOrder(t^.right);
    END; (* IF *)
  END; (* PrintInOrder *)

  PROCEDURE PrintPostOrder(t: TreePtr);
  BEGIN (* PrintPostOrder *)
    IF (t <> NIL) THEN BEGIN
      PrintPostOrder(t^.left);
      PrintPostOrder(t^.right);
      Write(t^.val:4);
    END; (* IF *)
  END; (* PrintPostOrder *)

  PROCEDURE PrintPreOrder(t: TreePtr);
  BEGIN (* PrintPreOrder *)
    IF (t <> NIL) THEN BEGIN
      Write(t^.val:4);
      PrintPreOrder(t^.left);
      PrintPreOrder(t^.right);
```

```
    END; (* IF *)
  END; (* PrintPreOrder *)

  FUNCTION ValueOf(t: TreePtr): INTEGER;
    VAR
      number,i: INTEGER;
  BEGIN (* ValueOf *)
    IF (t^.val = '*') THEN BEGIN
      ValueOf := ValueOf(t^.left) * ValueOf(t^.right);
    END ELSE IF (t^.val = '+') THEN BEGIN
      ValueOf := ValueOf(t^.left) + ValueOf(t^.right);
    END ELSE IF (t^.val = '-') THEN BEGIN
      ValueOf := ValueOf(t^.left) - ValueOf(t^.right);
    END ELSE IF (t^.val = '/') THEN BEGIN
      ValueOf := ValueOf(t^.left) DIV ValueOf(t^.right);
    END ELSE BEGIN
      FOR i := 1 TO Length(t^.val) DO BEGIN
        number := 0;
        number := (number * 10) + Ord(t^.val[i]) - Ord('0');
        ValueOf := number;
      END; (* FOR *)
    END;
  END; (* ValueOf *)

BEGIN (* ModTreeA2 *)

END. (* ModTreeA2 *)
```