Aufgabe 2 - Optimierender MidiPascal-Compiler

**Lösungsidee:**  
Für das Aufbauen des Binärbaumes entferne ich die semantischen Aktionen die in den Prozeduren Expr, Term und Fact vorkommen und ersetzte sie mit den Aufbau Aktionen des Baumes ähnlich zur letzten Hausübung und führe die Prozedur EmitCodeForExprTree nach jedem Aufruf der Prozedur Expr (außerhalb von Fact) auf. Dort optimier ich den Baum, indem ich in post-order bei jedem Operator überprüfe, ob eine Optimierung mit dem linken und rechten wert möglich ist. Falls ja ändere ich diesen Knoten so ab, das er optimiert ist, bis man alle Knoten abgearbeitet hat.   
Fürs Ausgeben der Codesequenz gehe ich erneut nach optimieren des Baums rekursiv in post-order durch (Grund: durch die Optimierung davor kann es passieren das Knoten gelöscht/geändert werden und es deshalb schwer ist das Ausgeben im gleichen rekursiven Durchgang wie das Optimieren zu lösen) und speichere die jeweiligen Byte Operationen.

**Zeitaufwand:** 2,5h

**Code (Ausschnitte):**

**Teilaufgabe A:**

type

NodePtr = ^Node;

  Node = record

    left, right: NodePtr;

    val: string;

    valInt: integer;

    isOperator: boolean;

    isIdent: boolean;

  end;

  ExprTreePtr = NodePtr;

function NewNode: NodePtr;

var

  n: NodePtr;

begin

  New(n);

  n^.left := nil;

  n^.right := nil;

  n^.isOperator := false;

  n^.isIdent := false;

  NewNode := n;

end;

function CreateOperatorNode(left, right: NodePtr; opVal: string): NodePtr;

var

  n: NodePtr;

begin

  n := NewNode;

  n^.val := opVal;

  n^.isOperator := true;

  n^.left := left;

  n^.right := right;

  CreateOperatorNode := n;

end;

procedure Expr(var e: NodePtr);

var

  right: NodePtr;

begin

  Term(e); if not success then exit;

  while(sy = plusSy) or (sy = minusSy) do

    case sy of

      plusSy:

      begin

        NewSy;

        (\*sem\*) right := NewNode; (\*endsem\*)

        Term(right); if not success then exit;

        (\*sem\*) e := CreateOperatorNode(e, right, '+'); (\*endsem\*)

      end;

      minusSy:

      begin

        NewSy;

        (\*sem\*) right := NewNode; (\*endsem\*)

        Term(right); if not success then exit;

        (\*sem\*) e := CreateOperatorNode(e, right, '-'); (\*endsem\*)

      end;

    end;

end;

procedure Term(var t: NodePtr);

var

  right: NodePtr;

begin

  Fact(t); if not success then exit;

  while(sy = timesSy) or (sy = divSy) do

    case sy of

      timesSy:

      begin

        NewSy;

        (\*sem\*) right := NewNode; (\*endsem\*)

        Fact(right); if not success then exit;

        (\*sem\*) t := CreateOperatorNode(t, right, '\*'); (\*endsem\*)

      end;

      divSy:

      begin

        NewSy;

        (\*sem\*) right := NewNode; (\*endsem\*)

        Fact(right); if not success then exit;

        (\*sem\*) t := CreateOperatorNode(t, right, '/'); (\*endsem\*)

      end;

    end;

end;

procedure Fact(var f: NodePtr);

begin

  case sy of

    identSy:

    begin

      (\*sem\*) f^.val := identStr; f^.isIdent := true; f^.valInt := AddrOf(identStr); (\*endsem\*)

      NewSy;

    end;

    numberSy:

    begin

      (\*sem\*) f^.val := 'const'; f^.valInt := numberVal; (\*endsem\*)

      NewSy;

    end;

    leftParSy:

    begin

      NewSy;

      Expr(f); if not success then exit;

      if SyIsNot(rightParSy) then exit;

      NewSy;

    end;

  else

    success := false;

  end;

end;

**Teilaufgabe B:**

procedure DisposeExprTree(t: ExprTreePtr);

begin

  if t <> nil then

  begin

    DisposeExprTree(t^.left);

    DisposeExprTree(t^.right);

    Dispose(t);

  end;

end;

procedure RecursiveEmit(t: ExprTreePtr);

begin

  if(t = nil) then Exit;

  RecursiveEmit(t^.left);

  RecursiveEmit(t^.right);

  if t^.isOperator then

  begin

    // emit operator operation

    if (t^.val = '+') then Emit1(AddOpc)

    else if (t^.val = '-') then Emit1(SubOpc)

    else if (t^.val = '\*') then Emit1(MulOpc)

    else if (t^.val = '/') then Emit1(DivOpc);

  end

  else if t^.isIdent then

    // emit ident value

    Emit2(LoadValOpc, t^.valInt)

  else

    // emit const value

    Emit2(LoadConstOpc, t^.valInt);

end;

procedure EmitCodeForExprTree(t: ExprTreePtr);

begin

  RecursiveEmit(t);

  DisposeExprTree(t);

end;

**Teilaufgabe C & D:**

procedure DisposeExprTree(t: ExprTreePtr);

begin

  if t <> nil then

  begin

    DisposeExprTree(t^.left);

    DisposeExprTree(t^.right);

    Dispose(t);

  end;

end;

procedure ReplaceWithLeftNode(var t: ExprTreePtr);

var

  dummy: NodePtr;

begin

  dummy := t^.left;

  Dispose(t^.right);

  Dispose(t);

  t := dummy;

end;

procedure ReplaceWithRightNode(var t: ExprTreePtr);

var

  dummy: NodePtr;

begin

  dummy := t^.right;

  Dispose(t^.left);

  Dispose(t);

  t := dummy;

end;

procedure OptimizeExprTree(var t: ExprTreePtr);

begin

  if(t = nil) then Exit;

  OptimizeExprTree(t^.left);

  OptimizeExprTree(t^.right);

  // if left and right node are constant values remove the nodes

  // and make the current node the result with the operation

  if t^.isOperator and (t^.left^.val = 'const')

    and (t^.right^.val = 'const') then

  begin

    if (t^.val = '+') then

      t^.left^.valInt := t^.left^.valInt + t^.right^.valInt

    else if (t^.val = '-') then

      t^.left^.valInt := t^.left^.valInt - t^.right^.valInt

    else if (t^.val = '\*') then

      t^.left^.valInt := t^.left^.valInt \* t^.right^.valInt

    else if (t^.val = '/') then

      t^.left^.valInt := t^.left^.valInt div t^.right^.valInt;

    ReplaceWithLeftNode(t);

  end

  // try to optimize add and sub expressions

  else if(t^.val = '+') or (t^.val = '-') then

  begin

    if not t^.left^.isOperator and not t^.left^.isIdent

      and (t^.left^.valInt = 0) then

    begin

      if (t^.val = '-') then

        // if our expr is 0 - a we want to invert a so we get -a

// redundant because compiler can’t handle signed int anyway

        t^.right^.valInt := t^.right^.valInt \* -1;

      ReplaceWithRightNode(t);

    end else

    if not t^.right^.isOperator and not t^.right^.isIdent

      and (t^.right^.valInt = 0) then

      ReplaceWithLeftNode(t);

  end

  // try to optimize mul and div expressions

  else if(t^.val = '\*') or (t^.val = '/') then

    if (t^.val = '\*') and not t^.left^.isOperator and not t^.left^.isIdent then

    begin

      if (t^.left^.valInt = 0) then

        ReplaceWithLeftNode(t)

      else if (t^.left^.valInt = 1) then

        ReplaceWithRightNode(t);

    end else if not t^.right^.isOperator and not t^.right^.isIdent then

      if (t^.val = '/') and (t^.right^.valInt = 0) then

      begin

        WriteLn('\*\*\* Error: div. by zero');

        HALT;

      end else

      if (t^.val = '\*') and (t^.right^.valInt = 0) then

        ReplaceWithRightNode(t) else

      if (t^.right^.valInt = 1) then

        ReplaceWithLeftNode(t);

end;

procedure RecursiveEmit(t: ExprTreePtr);

begin

  if(t = nil) then Exit;

  RecursiveEmit(t^.left);

  RecursiveEmit(t^.right);

  if t^.isOperator then

  begin

    // emit operator operation

    if (t^.val = '+') then Emit1(AddOpc)

    else if (t^.val = '-') then Emit1(SubOpc)

    else if (t^.val = '\*') then Emit1(MulOpc)

    else if (t^.val = '/') then Emit1(DivOpc);

  end

  else if t^.isIdent then

    // emit ident value

    Emit2(LoadValOpc, t^.valInt)

  else

    // emit const value

    Emit2(LoadConstOpc, t^.valInt);

end;

procedure EmitCodeForExprTree(t: ExprTreePtr);

begin

  OptimizeExprTree(t);

  RecursiveEmit(t);

  // for testing purposes

  // WriteTree(t, 1); writeln;

  DisposeExprTree(t);

end;

**Test:**

Für Testzwecke schrieb ich mir eine Hilfsprozedur, die mir den Baum in die Konsole ausgibt zur Überprüfung, ob er richtig optimiert wurde.

procedure WriteTree(t: ExprTreePtr; indent: Integer);

var

  i: Integer;

begin

  if t <> nil then

  begin

    for i := 1 to indent do

      Write('  ');

    if not t^.isOperator and not t^.isIdent then

writeln(t^.val, ': ', t^.constVal)

else

WriteLn(t^.val);

    WriteTree(t^.left, indent + 1);

    WriteTree(t^.right, indent + 1);

  end;

end;

**Aufgabe c:**

* x := 0 + x; Ergebnis; Ein Bild, das Text, Schrift, weiß, Screenshot enthält.

  Automatisch generierte Beschreibung
* x := 0 + 3; Ergebnis: 
* x := 3 + 0; Ergebnis: 
* x := 3 - 0; Ergebnis: 
* x := 3 \* 1; Ergebnis: 
* x := 1 \* 3; Ergebnis: 
* x := 3 / 1; Ergebnis: 

Aufgabe d:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Ergebnis: Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung