

☒ 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
<code>VAR x: INTEGER;</code>	<code>VAR x: INTEGER;</code>
<code>IF x THEN ...</code>	<code>IF x <> 0 THEN ...</code>
<code>WHILE x DO ...</code>	<code>WHILE x <> 0 DO ...</code>

Mit diesen Spracherweiterungen könnte man dann z. B. ein MidiPascal-Programm schreiben, das für eine eingebene Zahl n die Fakultät $f = n!$ iterativ berechnet und diese ausgibt. Siehe Quelltextstück rechts.

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

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

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

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

MidiPascal	Bytecode (mit fiktiven Adressen)
<pre>IF x THEN BEGIN then stats END; ...</pre>	<pre>1 LoadVal x 4 JmpZ 99 ... code for then stats 99 ...</pre>
<pre>IF x THEN BEGIN then stats</pre>	<pre>1 LoadVal x 4 JmpZ 66 ... code for then stats</pre>

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-, Prä- oder Postfix-Notation) zu übersetzen.

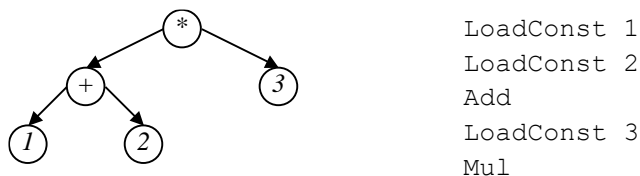
Die Repräsentation von arithmetischen Ausdrücken in Form von Binärbäumen bietet aber auch die Möglichkeit, einfache Optimierungen in den MidiPascal-Compiler einzubauen.

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

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

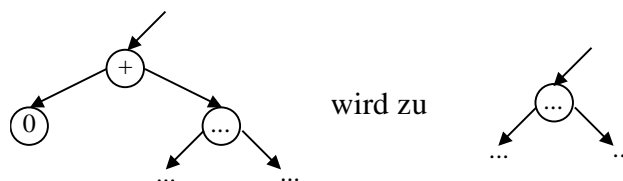
die aus dem Binärbaum in einem Postorder-Durchlauf Bytecode für die Berechnung des Ausdrucks durch die virtuelle MiniPascal-Maschine erzeugt.

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

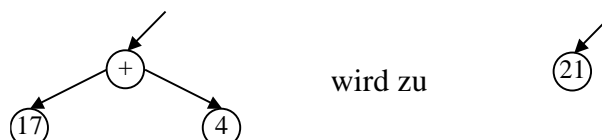


Damit können Sie Ihren Compiler zwar schon testen – aber von Optimierung ist noch keine Rede. Die erzeugten Binärbäume eignen sich aber dazu, einfache Optimierungen an Ausdrücken vorzunehmen, die z. B. in modernen Compilern eingesetzt werden: die Binärbäume werden transformiert und erst die sich daraus 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.: $\dots + 17 + 4 + \dots$ wird zu $\dots + 21 + \dots$



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

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

Aufgabe 1 - MidiPascal

Lösungsidee:

Man nimmt den MiniPascal Compiler der Übung und fügt beim lexikalischen Analysator die neuen 5 Symbole zum Enum hinzu und beim Scanner beim Erkennen eines Idents die Zusatzoption der 5 neuen Symbole. Beim Parser in der Statement Prozedur fügt man beim Case wieder die Zusatzoptionen der neuen Symbole hinzu mit deren Grammatik und Semantischen Aktionen, die in der Angabe gegeben sind.

Zeitaufwand: ~15min

Code (Ausschnitte):

```
unit MPLex;

interface

type
  Symbol = (
    emptySy, eofSy, errSy,
    numberSy, identSy,
    semicolonSy, colonSy, commaSy, periodSy, assignSy,
    plusSy, minusSy, timesSy, divSy,
    leftParSy, rightParSy,
    programSy,
    varSy, integerSy,
    ifSy, elseSy, thenSy, whileSy, doSy,
    readSy, writeSy,
    beginSy, endSy
  );

...

procedure NewSy;
begin
  sy := emptySy;
  repeat
    while((ch = ' ') or (ch = tabCh)) do NewCh;

    syLnr := chLnr;
    syCnr := chCnr;

    case ch of
      eofCh: sy := eofSy;
      '+':
```

```

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;
';':
begin sy := semicolonSy; NewCh; end;
':':
begin
    sy := colonSy; NewCh;
    if(ch = '=') then
        begin
            sy := assignSy; NewCh;
        end;
    end;
end;
'.':
begin sy := periodSy; NewCh; end;
',':
begin sy := commaSy; NewCh; end;
'0'..'9':
begin
    sy := numberSy;
    numberval := 0;
    while((ch >= '0') and (ch <= '9')) do
        begin
            numberval := numberVal * 10 + Ord(ch) - Ord('0');
            NewCh;
        end;
    end;
end;
'a'..'z', 'A'..'Z', '_':
begin
    identStr := '';
    while((ch in ['a'..'z', 'A'..'Z', '_', '0'..'9'])) do
        begin
            identStr := identStr + UpCase(ch);
            NewCh;
        end;
    end;
    if(identStr = 'PROGRAM') then
        sy := programSy
    else if(identStr = 'VAR') then
        sy := varSy
    else if(identStr = 'READ') then

```

```

        sy := readSy
    else if(identStr = 'WRITE') then
        sy := writeSy
    else if(identStr = 'BEGIN') then
        sy := beginSy
    else if(identStr = 'END') then
        sy := endSy
    else if(identStr = 'INTEGER') then
        sy := integerSy
    else if(identStr = 'IF') then
        sy := ifSy
    else if(identStr = 'ELSE') then
        sy := elseSy
    else if(identStr = 'THEN') then
        sy := thenSy
    else if(identStr = 'WHILE') then
        sy := whileSy
    else if(identStr = 'DO') then
        sy := doSy
    else sy := identSy;
end;

```

```

    else sy := errSy;
end;

```

```

until(sy <> emptySy);
end;

```

```

unit MPC_SS;

```

```

...

```

```

procedure Stat;
var
    destId: string;
    addr, addr1, addr2: integer;
begin
    case sy of
        identSy:
            begin
                (*sem*)
                destId := identStr;
                if (not IsDecl(destId)) then SemErr('variable not declared') else
Emit2(LoadAddrOpc, AddrOf(destId));
                (*endsem*)
                NewSy;
                if (SyIsNot(assignSy)) then Exit;
                NewSy;
            end;
    end;
end;

```

```

    Expr; if (not success) then Exit;
    (*sem*)
    if (IsDecl(destId)) then Emit1(StoreOpc);
    (*endsem*)
end;
readSy:
begin
    NewSy;
    if (SyIsNot(leftParSy)) then Exit;
    NewSy;
    if (SyIsNot(identSy)) then Exit;
    (*sem*)
    if (not IsDecl(identStr)) then SemErr('variable not declared') else
Emit2(ReadOpc, AddrOf(identStr));
    (*endsem*)
    NewSy;
    if (SyIsNot(rightParSy)) then Exit;
    NewSy;
end;
writeSy:
begin
    NewSy;
    if (SyIsNot(leftParSy)) then Exit;
    NewSy;
    Expr; if (not success) then Exit;
    (*sem*) Emit1(WriteOpc); (*endsem*)
    if (SyIsNot(rightParSy)) then Exit;
    NewSy;
end;
beginSy:
begin
    NewSy;
    StatSeq; if not success then exit;
    if SyIsNot(endSy) then exit;
    NewSy;
end;
ifSy:
begin
    NewSy;
    if SyIsNot(identSy) then exit;
    (*sem*)
    if not IsDecl(identStr) then SemErr('variable not declared');
    Emit2(LoadValOpc, AddrOf(identStr));
    Emit2(JmpZOpc, 0); (*0 as dummy address*)
    addr := CurAddr - 2;
    (*endsem*)
    NewSy;
    if SyIsNot(thenSy) then exit;

```



```

NewSy;
Stat; if not success then exit;
while (sy = elseSy) do
begin
    (*sem*)
    Emit2(JmpOpc, 0); (*0 as dummy address*)
    FixUp(addr, CurAddr);
    addr := CurAddr - 2;
    (*endsem*)
    NewSy;
    Stat; if not success then exit;
end;
(*sem*) FixUp(addr, CurAddr); (*endsem*)
end;
whileSy:
begin
    NewSy;
    if SyIsNot(identSy) then exit;
    (*sem*)
    if not IsDecl(identStr) then SemErr('variable not declared');
    addr1 := CurAddr;
    Emit2(LoadValOpc, AddrOf(identStr));
    Emit2(JmpZOpc, 0); (*0 as dummy address*)
    addr2 := CurAddr - 2;
    (*endsem*)
    NewSy;
    if SyIsNot(doSy) then exit;
    NewSy;
    Stat; if not success then exit;
    (*sem*) Emit2(JmpOpc, addr1); FixUp(addr2, CurAddr); (*endsem*)
end;
end;
end;

```

Test:

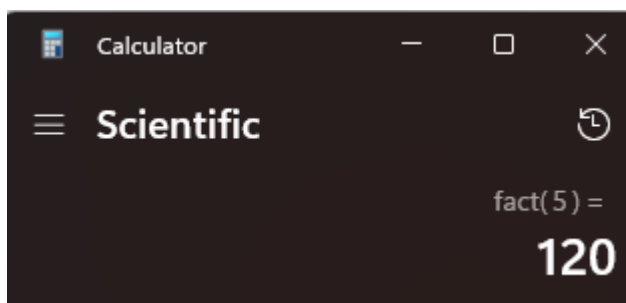
Midipascal Test Programm, dass die Fakultät einer eingegebenen Zahl berechnet und ausgibt.

```
FAKU.mp U ×
UE6 > hu > FAKU.mp
1  PROGRAM SVP;
2  |  VAR
3  |  |  f, n: INTEGER;
4  BEGIN
5  |  Read(n);
6  |  f := n; n := n - 1;
7  |  WHILE n DO BEGIN
8  |  |  f := n * f;
9  |  |  n := n - 1;
10 |  END;
11 |  WRITE(f);
12 END.

> MiniPascal source file > faku.mp
Parsing started ...
file compiled successfully

code interpretation started ...
var@1 > 5
120
... code interpretation ended
```

Die Fakultät von 5 ist 120.



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 ersetze 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 optimiere 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, dass 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 dass 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;
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;

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;
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 OptimizeExprTree(var t: ExprTreePtr);
var
    dummy: NodePtr;
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;
        dummy := t^.left;
        Dispose(t^.right);
        Dispose(t);
        t := dummy;
    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
```

```

        t^.right^.valInt := t^.right^.valInt * -1;
    t := t^.right;
end else
if not t^.right^.isOperator and not t^.right^.isIdent
    and (t^.right^.valInt = 0) then
    t := t^.left;
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 and (t^.left^.valInt = 1) then
        t := t^.right
    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^.right^.valInt = 1) then t := t^.left;
        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

```



```

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;
end;

```

Aufgabe c:

- ```

MiniPascal source file > test.mp
Parsing started ...

```
- $x := 0 + x$ ; Ergebnis:  $x$
  - $x := 0 + 3$ ; Ergebnis:  $\text{const: } 3$
  - $x := 3 + 0$ ; Ergebnis:  $\text{const: } 3$
  - $x := 3 - 0$ ; Ergebnis:  $\text{const: } 3$
  - $x := 3 * 1$ ; Ergebnis:  $\text{const: } 3$
  - $x := 1 * 3$ ; Ergebnis:  $\text{const: } 3$
  - $x := 3 / 1$ ; Ergebnis:  $\text{const: } 3$

#### Aufgabe d:

```
UE6 > hu > test.mp
1 PROGRAM TEST;
2 | VAR
3 | | x: INTEGER;
4 BEGIN
5 | x := (20+10) / (2 * 5);
6 |
7 | write(x);
8 END.
```

#### Ergebnis:

```
MiniPascal source file > test.mp
Parsing started ...
const: 3

X

file compiled successfully

code interpretation started ...
3
... code interpretation ended

Heap dump by heaptrc unit of C:\Repos\2023SS_ADF\UE6\hu\mpc.exe
9 memory blocks allocated : 2164/2200
9 memory blocks freed : 2164/2200
0 unfreed memory blocks : 0
True heap size : 98304 (96 used in System startup)
True free heap : 98208
```