Humboldt-Universität zu Berlin Institut für Informatik

Codeoptimierung **SS 00**

Codeoptimierung – Übungsblatt 4
peep – Gruppenarbeit (max. 3 Personen)

Abgabe: 6.6.00 vor 9:30h

(10 Punkte) *peep* ist ein Guckloch-Optimierer. Er liest Code von *stdin* und schreibt optimierten Code nach *stdout*. *peep* liest Regeln zur Guckloch-Optimierung aus einer Datei, deren Name als Parameter beim Aufruf angegeben wird. Diese Datei hat folgenden Aufbau:

```
<Muster Eingabe Zeile 1>
<Muster Eingabe Zeile 2>
...
<Muster Eingabe Zeile n>
=
<Muster Ausgabe Zeile 1>
<Muster Ausgabe Zeile 2>
...
<Muster Ausgabe Zeile m>
<Leerzeile>
```

Die Datei kann beliebig viele Regeln (I/O-Muster) enthalten, jeweils durch eine Leerzeile getrennt. Bei der Mustererkennung werden *stdin* und Zeilenmuster miteinander verglichen. Ein Muster kann neben Literalen auch beliebige Ausdrücke enthalten, die mit *$n* spezifiziert werden ($n = 0..9$, jeweils max. 80 Zeichen lang). Ein solcher Ausdruck ist durch das nächste Literal, ein Space, Tab oder Newline begrenzt. (Space, Tab und Newline trennen in der Eingabe und in den Mustern die Tokens. Die Anzahl der Spaces, Tabs und Newlines ist dabei nicht signifikant.) Wenn *stdin* mit einem Eingabemuster übereinstimmt, wird eine Ersetzung gemäß des Ausgabemusters vorgenommen. Andernfalls wird die nächste Zeile von *stdin* gelesen. Beispiel:

```
Regel:                   Eingabe/Ausgabe:
sethi %hi($1),$2         sethi %hi(_printf),%o1
add $2,%lo($1),$3        add %o1,%lo(_printf),%o1
call $3,$5               call %o1,1
=
call $1,$5               call _printf,1

mul $1                   mul %o0,10,%o0
=
smul $1                  smul %o0,10,%o0
```

Damit die Effekte der Guckloch-Optimierung propagieren können, werden Eingabemuster und *stdin* von unten nach oben miteinander verglichen, d.h. Zeile n *stdin* mit Musterzeile n, dann $n-1$ etc. Die Anwendung der Regeln auf Zeile n *stdin* erfolgt dabei solange (auch wiederholt), bis keine Regel mehr anwendbar ist (Fixpunkt-Algorithmus).

Zur Bearbeitung der Aufgabe loggen Sie sich bitte auf einer Sun SPARC ein und geben (in Ihrem Home-Verzeichnis) die folgenden Befehle ein:

```
mkdir ˜/codeopt
mkdir ˜/codeopt/ub4
cd ˜/codeopt/ub4
cp ˜mueller/classes/codeopt/ub4/* .
```

Erweitern Sie dann die Dateien *peep.c* (das C Programm) und *sparc.pat* (die Regeldatei mit I/O-Mustern, die 2 alte/vorgegebene und 8 neue Regeln enthalten soll). Testen Sie das Programm, z.B.:

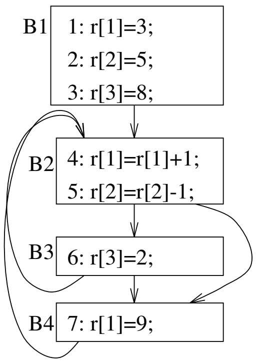
```
ub1.cc ˜mueller/test/src/cal.c   # erzeugt Assembler-Code "cal.s"
peep sparc.pat < cal.s > cal.S   # erzeugt optimierten Assembler-
Code "cal.S"
ub1.ld cal.S                     # erzeugt ausf{\"u}hrbares Pro-
gramm "cal.out"
cal.out 1995                     # Programm ausf{\"u}hren
```

Die Abgabe umfaßt die Dateien *peep.c* sowie *sparc.pat* und erfolgt per Upload durch http://eiche.informatik.hu-berlin.de:8081/codeopt bis zum oben genannten Datum.

Algorithmus zur Bestimmung der aktiven Variablen

```
Eingabe: def[] und use[] f{\"u}r jeden Block
Ausgabe: in[] und out[] f{\"u}r jeden Block

for each Block X do
  in[X] := 0;
while change of any in[] do
  for each Block X do
    out[X] := 0;
    for each S in Succs(X) do
      out[X] := out[X] + in[S];
    in[X] := use[X] + (out[X] - def[X]);
```

Beispiel:



def[B1]={r[1],r[2],r[3]}
use[B1]={}

def[B2]={}
use[B2]={r[1],r[2]}

def[B3]={r[3]}
use[B3]={}
def[B4]={r[1]}
use[B4]={}

Initialisierung:	1. Pass:	2. Pass:
out[B1]={}	{}	{r[1],r[2]}
in [B1]={}	{}	{}
out[B2]={}	{}	{r[1],r[2]}
in [B2]={}	{r[1],r[2]}	{r[1],r[2]}
out[B3]={}	{r[1],r[2]}	{r[1],r[2]}
in [B3]={}	{r[1],r[2]}	{r[1],r[2]}
out[B4]={}	{r[1],r[2]}	{r[1],r[2]}
in [B4]={}	{r[2]}	{r[2]}

Static Single Assignment Form (SSA)

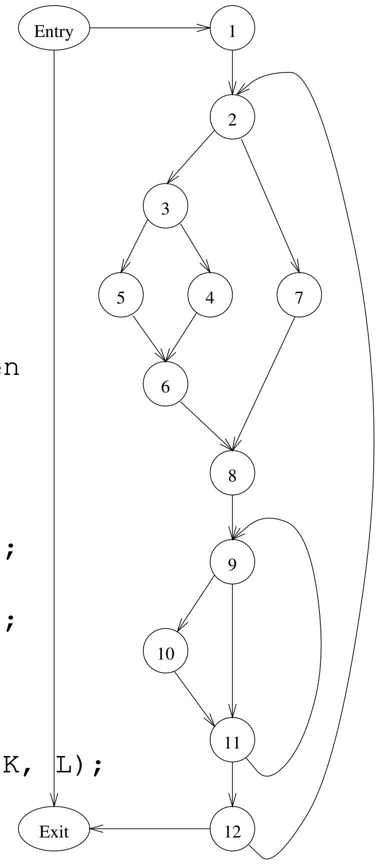
Beispiel:

Code ohne Verzweigung		Code mit Verzweigung	
Code	SSA	Code	SSA
V := 4;	$V_1 := 4;$	if P then	if P then
:= V + 5;	$:= V_1 + 5;$	V := 4;	$V_1 := 4;$
V := 6;	$V_2 := 6;$	else	else
:= V + 7;	$:= V_2 + 7;$	V := 6;	$V_2 := 6;$
			$V_3 := \phi(V_1,V_2)$

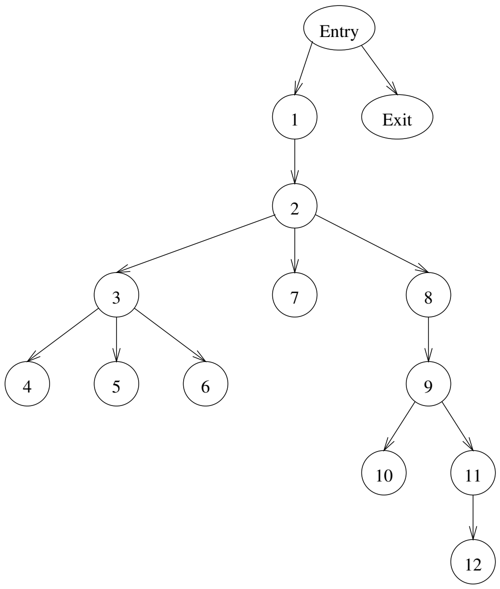
Dominatorbaum: Beispiel und Konstruktionsalgorithmus

```
1: I := 1;
   J := 1;
   K := 1;
   L := 1;
2: repeat




     if (P) then
3:     J := I;
       if (Q) then
4:       L := 2;
5:     else
         L := 3;
6:
       K := K + 1;
7:   else
       K := K + 2;
8:


     print(I, J, K, L);
9:   repeat

       if (R) then
10:      L := L + 4;
11:
     until S;
12:  I := I + 6;
   until T;
```

Kontrollfluss



Dominatorbaum




```
Eingabe: Grundbl{\"o}cke und Kontrollflu{\ss}
Ausgabe: unmittelbarer Dominator idom f{\"u}r jeden Block
i := 1;
for each Block X in a top-down depth-first traversal do
  sdom(X) := X;
  dfs(X) := i;
  i := i + 1;
for each Block X != Root in a bottom-up depth-first traversal do
  for each Y in Preds_cfg(X) do
    U := eval(Y);
    if dfs(sdom(U)) < dfs(sdom(X)) then
      sdom(X) := sdom(U);
    mark X as Done;
    mark X as Bucket;
  for each Block Y do
    if Y marked as Bucket AND sdom(Y) == Parent_dfs(X) then
      unmark Y as Bucket;
      U := eval(Y);
      if dfs(sdom(U)) < dfs(sdom(Y))
        idom(Y) := U;
      else
        idom(Y) := Parent_dfs(X);
```

```
for each Block X != Root in a top-down depth-first traversal do
  if idom(X) != sdom(X) then
    idom(X) := idom(idom(X));

function eval(Y) return Block is
  if not Y marked as Done then
    return Y;
  U := Y;
  Y := Parent_dfs(Y);
  while Y != Null AND Y marked as Done do
    if dfs(sdom(Y)) < dfs(sdom(U)) then
      U := Y;
    Y := Parent_dfs(Y);
  return U;
```

Algorithmus zur Bestimmung der Dominatorgrenzen

```
Eingabe: Dominatorbaum
Ausgabe: Dominatorgrenzen f{\"u}r jeden Block

for each X in a bottom-up traversal of the dominator tree do
  DF(X) := {};
  for each Y in Succs(X) do
    if idom(Y) != X then
      DF(X) := DF(X) + {Y}; /* local */
  for each Z in Children(X) do
    for each Y in DF(Z) do
      if idom(Y) != X then
       DF(X) := DF(X) + {Y}; /* up */
```