NTNU | Norwegian University of Science and Technology

Compiler Construction

Solutions to the 2020 exam

Michael Engel

1 Regular Languages

1.1 Write a regular expression for all strings of a’s and b’s which contain the substring abba

(a|b)*abba(a|b)*

1.2 Write a regular expression for all strings of x’s and y’s where every y is immediately followed by at least 3 x’s

(x | (yxxx))*

1 Regular Languages

1.3 Write a regular expression for all strings of p's and q's which contain an odd number of q's

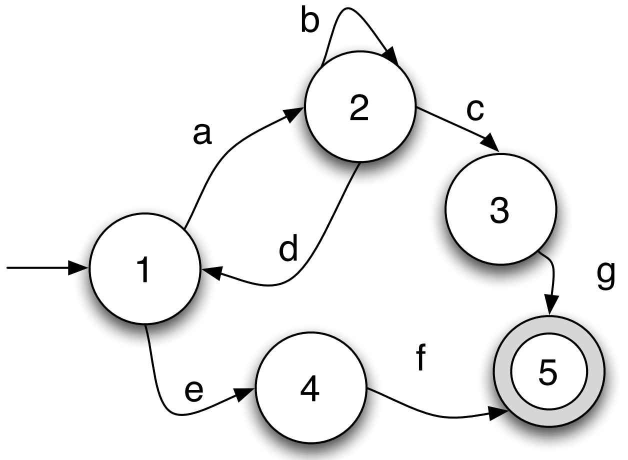
p* q ((q p* q) | p)*

1.4 A finite language is a language with a finite number of strings. For example, the language with only the strings a, ba, and bba is finite, while the languages in question 1.1, 1.2, and 1.3 above are not finite. Are all finite languages regular? If so, explain why. If not, give an example of a finite language which is not regular.

Yes, all finite languages are regular. Imagine a finite language comprised of strings s_1, s_2, …, s_n. We can simply write a regular expression to describe it by ORing together all possible cases (s_1) | (s_2) | ... | (s_n).

Note that this *only* works for finite languages, because there have a finite number of cases to OR together.

2 NFAs and DFAs



2.1. Convert the NFA given in fig. 1 to a DFA

Whoops. This was the PDF version of a draft of the 2020 exam and it contained the wrong automaton. This is a DFA already (which is also a valid NFA).

2.2 Give the corresponding regular expression.

(ab*d)* ((ab*cg) | (ef))

3 Compiler Structure

For each of the rules of a programming language below, specify which phase of the compiler should verify that the program adheres to that rule…

3.1 A function call has the correct number of arguments

3.2 Digits [0-9] may appear in identifiers, but not as the first character (e.g., a123 and pi314 are valid identifiers,
but 2big is not).

3.3 Every variable must be declared before it is used in the program (as in C).

3.4 Assignments such as a=42; must end with a semicolon (;).

[see example exam solutions question 1]

4 LL Parsing

```
S → S a S b
  | c
  | Q q
Q → Q m
  | ε
```

4.1 Which non-terminals (if any) can derive the empty string?

Q (every production for S involved at least one non-terminal)

4.2 What are the FIRST sets of Q and S?

FIRST[S] = {m,q,c}   FIRST[Q] = {m}

4.3 What are the FOLLOW sets of Q and S?

FOLLOW[S] = {a, b}   FOLLOW[Q] = {q, m}

4 LL Parsing

```
S → S a S b
  | c
  | Q q
Q → Q m
  | ε
```

4.4 This grammar can not be parsed by an LL(0) or LL(1) parser. Explain why not.

It contains left recursion both in the first and fourth rule

4.5 Rewrite the grammar so that it accepts the same language, but can be parsed by an LL(1) parser (use ~~by~~ left factoring and eliminate left recursion).

```
S     → c SREST
      | Q q SREST
SREST → a S b SREST
      | ϵ
Q     → q
      | ϵ
```

5 Parse trees and ASTs

```
1  Start  → Expr
2  Expr   → Expr + Term
3         | Expr - Term
4         | Term
5  Term   → Term × Factor
6         | Term ÷ Factor
7         | Factor
8  Factor → "(" Expr ")"
9         | number
10        | ident
```

5.1 Draw the parse tree for the input string b×c+3÷c×b.

Rule	Applied on	Result
1. Start ➛ Expr	S	E
2. Expr ➛ Expr + Term	E	E + T
4. Expr ➛ Term	E + T	T + T
5. Term ➛ Term * Factor	T + T	T * F + T
7. Term ➛ Factor	T * F + T	F * F + T
9. Factor ➛ number	F * F + T	i * F + T
10. Factor ➛ ident	i * F + T	i * i + T
5. Term ➛ Term * Factor	i * i + T	i * i + T * F
6. Term ➛ Term / Factor	i * i + T * F	i * i + T / F * F
7. Term ➛ Factor	i * i + T / F * F	i * i + F / F * F
10. Factor ➛ ident	i * i + F / F * F	i * i + n / F * F
9. Factor ➛ number	i * i + i / F * F	i * i + n / i * F
10. Factor ➛ ident	i * i + i / n * F	i * i + n / i * i

5 Parse trees and ASTs

The following grammar is given:

```
 1 Start  → Expr
 2 Expr   → Expr + Term
 3          | Expr - Term
 4          | Term
 5 Term   → Term × Factor
 6          | Term ÷ Factor
 7          | Factor
 8 Factor → "(" Expr ")"
 9          | number
10          | ident
```

5.1 Draw the parse tree for the input string
b×c+3÷c×b.

$$\begin{tikzcd}
 & & & & \text{Start} \arrow[d] & & & \\
 & & & & \text{Expr} \arrow[dll] \arrow[d] \arrow[drr] & & & \\
 & & \text{Expr} \arrow[dl] & & + & & \text{Term} \arrow[dl] \arrow[d] \arrow[dr] & \\
 & \text{Term} \arrow[dl] \arrow[d] \arrow[dr] & & & & \text{Term} \arrow[dl] \arrow[d] \arrow[dr] & \times & \text{Factor} \arrow[d] \\
\text{Term} \arrow[d] & \times & \text{Factor} \arrow[d] & & \text{Term} \arrow[d] & \div & \text{Factor} \arrow[d] & b \\
\text{Factor} \arrow[d] & & c & & \text{Factor} \arrow[d] & & c & \\
b & & & & 3 & & &
\end{tikzcd}$$

5 Parse trees and ASTs

```
 1 Start  → Expr
 2 Expr   → Expr + Term
 3        | Expr - Term
 4        | Term
 5 Term   → Term × Factor
 6        | Term ÷ Factor
 7        | Factor
 8 Factor → "(" Expr ")"
 9        | number
10        | ident
```

The following grammar is given:

5.2 Draw the reduced parse tree to obtain the abstract syntax tree (AST).

$$\begin{tikzcd}
& & & + \arrow[dll] \arrow[drr] & & & \\
& \times \arrow[dl] \arrow[dr] & & & & \times \arrow[dl] \arrow[dr] & \\
b & & c & & \div \arrow[dl] \arrow[dr] & & b \\
& & & 3 & & c &
\end{tikzcd}$$

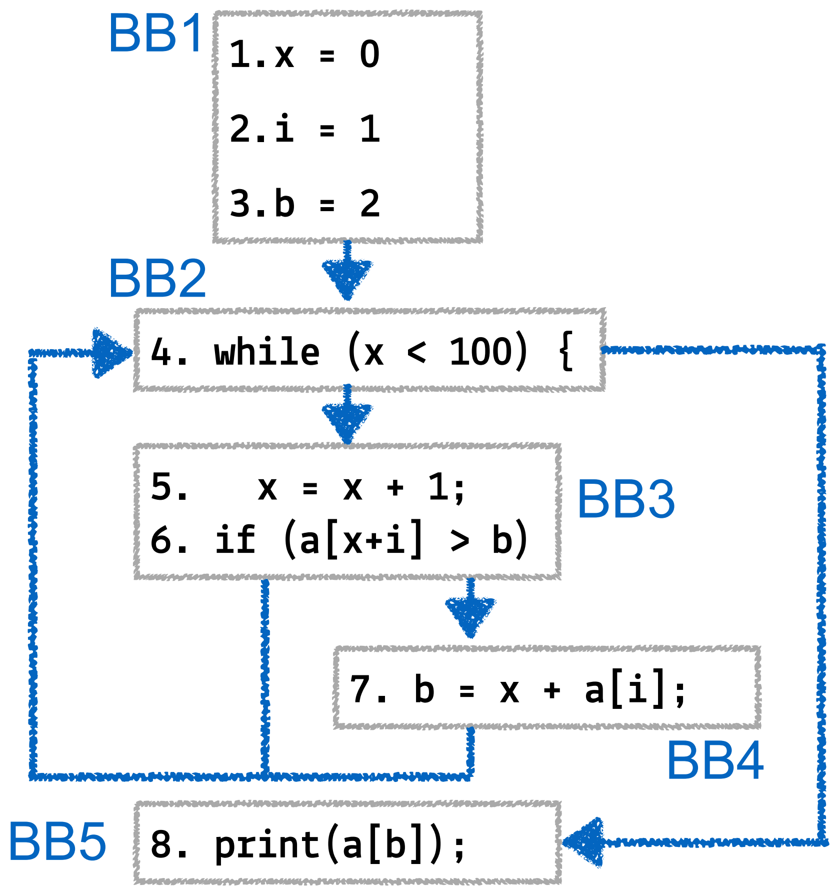
5.3 Derive a directed acyclic graph (DAG) from the AST that avoids code duplications.

(see 5.3 in example exam solutions)

6 Control Flow Graphs

```
BB1  1 x = 0;
     2 i = 1;
     3 b = 2;
BB2  4 while (x < 100) {
BB3  5   x = x + 1;
     6   if (a[x+i] > b)
BB4  7     b = x + a[i];
     8 }
BB5  9 print(a[b]);
```

6.1 Draw the control flow graph (CFG) for this piece of code



```
BB1:
1.x = 0
2.i = 1
3.b = 2

BB2:
4. while (x < 100) {

BB3:
5.    x = x + 1;
6. if (a[x+i] > b)

BB4:
7. b = x + a[i];

BB5:
8. print(a[b]);
```

Split the code into basic blocks (BB)

BB = linear code sequence

Jump at the end

Jump target at the start

6 Control Flow Graphs

6.2 Draw the dependence graph for the code.

(construction as in 6.2 in the example exam solution)

```
     1 x = 0;
BB1  2 i = 1;
     3 b = 2;
BB2  4 while (x < 100) {
     5   x = x + 1;
BB3  6   if (a[x+i] > b)
BB4  7     b = x + a[i];
     8 }
BB5  9 print(a[b]);
```

7 Three-Address Code

7.1 Translate the do-loop in the code to an equivalent while-loop.

```
x = 0;
i = 1;
do {
   if (a[x+i] > 0)
      i = x + a[i];
   x = x + 1;
} while (x < 100);
print(i);
```

Prepend the loop body before the first iteration of the while-loop:

```
if (a[x+i] > 0)
  i = x + a[i];
x = x + 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
```

You could simplify the code to:

```
i = 1;
if (a[1] > 0)
  i = a[1];
x = 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
print(i);
```

7 Three-Address Code

```
x = 0;
i = 1;
if (a[x+i] > 0)
  i = x + a[i];
x = x + 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
```

7.2 Translate the given code to three-address code (TAC). Show all intermediate steps of the translation.

This requires the translation of the do-loop from 7.1 since we only have a translation for a do-loop (see slide set 13 for the detailed steps).

Our specification of the IR in the course could be more formal…

```
    t0 = 0 // we omit the
    t1 = 1 // assignmts to x,i
    t2 = t0 + t1     // x+i
    t3 = a[t2]       // a[x+i]
    t4 = 0
    if t4>=t3 goto L1
    t5 = a[t1]       // a[i]
    t6 = t0 + t5     // i=x+a[i]
L1: t7 = t0 + 1        // x=x+1
L2: t8 = 100
    if t7 < t8 goto L4 // while(x<100)
    t9 = t7 + t1    // x+i
    t10 = a[t9]     // a[x+i]
    t11 = 0
    if t10>=t11 goto L3
    t12 = a[t1]     // a[i]
    t13 = t7 + t12  // i=x+a[i]
L3: t14 = t7 + 1    // x=x+1
    goto L2
L4: …               // WAIT?!?
```

7 Three-Address Code

```
x = 0;
i = 1;
if (a[x+i] > 0)
  i = x + a[i];
x = x + 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
```

The Phi-functions strike again!

When entering the while-loop for the first time, the value of x is stored in t7, for subsequent iterations in t14
(for i: t1 and t13)

How do we implement Phi-functions in TAC?

```
 t0 = 0 // we omit the
 t1 = 1 // assignmts to x,i
 t2 = t0 + t1     // x+i
 t3 = a[t2]       // a[x+i]
 t4 = 0
 if t4>=t3 goto L1
 t5 = a[t1]       // a[i]
 t6 = t0 + t5     // i=x+a[i]
L1: t7 = t0 + 1   // x=x+1
L2: t8 = 100
    if t7 < t8 goto L4 // while(x<100)
    t9 = t7 + t1   // x+i
    t10 = a[t9]    // a[x+i]
    t11 = 0
    if t10>=t11 goto L3
    t12 = a[t1]     // a[i]
    t13 = t7 + t12  // i=x+a[i]
L3: t14 = t7 + 1    // x=x+1
    goto L2
L4: …               // WAIT?!?
```

x

i

7 Three-Address Code

```
x = 0;
i = 1;
if (a[x+i] > 0)
  i = x + a[i];
x = x + 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
```

How do we implement Phi-functions in TAC?

Add additional unique temporal variables p0, p1 and copy the respective value *before entering* or *leaving* a loop iteration or if/else part!

Then use the unique variables inside the loop.

This violates a possible SSA assumption!

```
    t0 = 0 // we omit the
    t1 = 1 // assignmts to x,i
    t2 = t0 + t1     // x+i
    t3 = a[t2]       // a[x+i]
    t4 = 0
    if t4>=t3 goto L1
    t5 = a[t1]       // a[i]
    t6 = t0 + t5     // i=x+a[i]
L1: t7 = t0 + 1      // x=x+1
    p0 = t1          // store i
    p1 = t7          // store x
L2: t8 = 100
    if p1 < t8 goto L4 // while(x<100)
    t9 = p1 + p0 // x+i
    t10 = a[t9]    // a[x+i]
    t11 = 0
    if t10>=t11 goto L3
    t12 = a[p0]      // a[i]
    t13 = p0 + t12 // i=x+a[i]
    p0  = t13
L3: t14 = p1 + 1     // x=x+1
    p1  = t14
    goto L2
L4: …
```

7 Three-Address Code

7.3 When translating switch statements to TAC,
you can use cascaded gotos or a jump table.
Shortly discuss the advantages and differences of
each approach.

This is a tradeoff between runtime and memory consumption (see slide set 13 slide 26/27 for the two implementations).

Cascaded gotos require a higher runtime, depending on the number of switch labels you have to go through to find the one for the specific case.

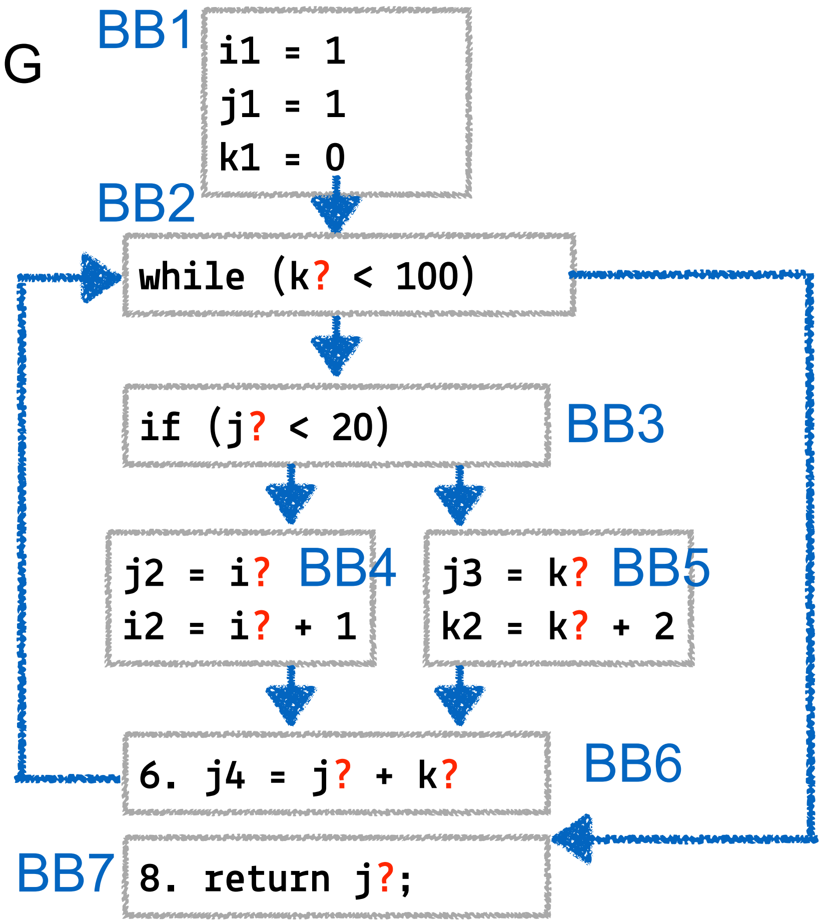
Tables require more memory in general, especially for switch statements with a sparse label space, e.g. 1, 100, 10000… (all table entries not related to a label have to link to either error handling code or a default label, depending on the semantics of your language)

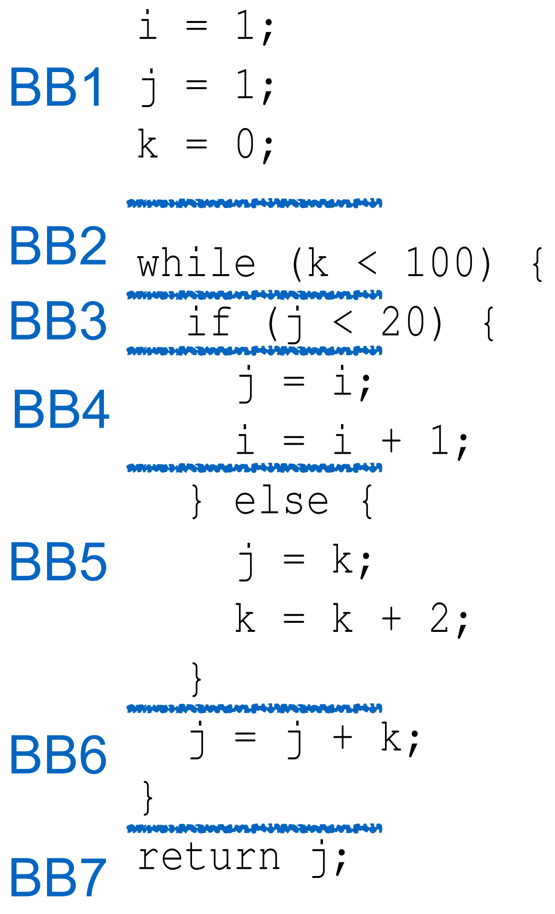
8 Static Single Assignment Form

8.1 Give the result of splitting the code into basic blocks. (see BBs on the right)

8.2 Draw the CFG using unique variable names for each assignment as used in the SSA form.

For variables with ?, we need a Phi-function!



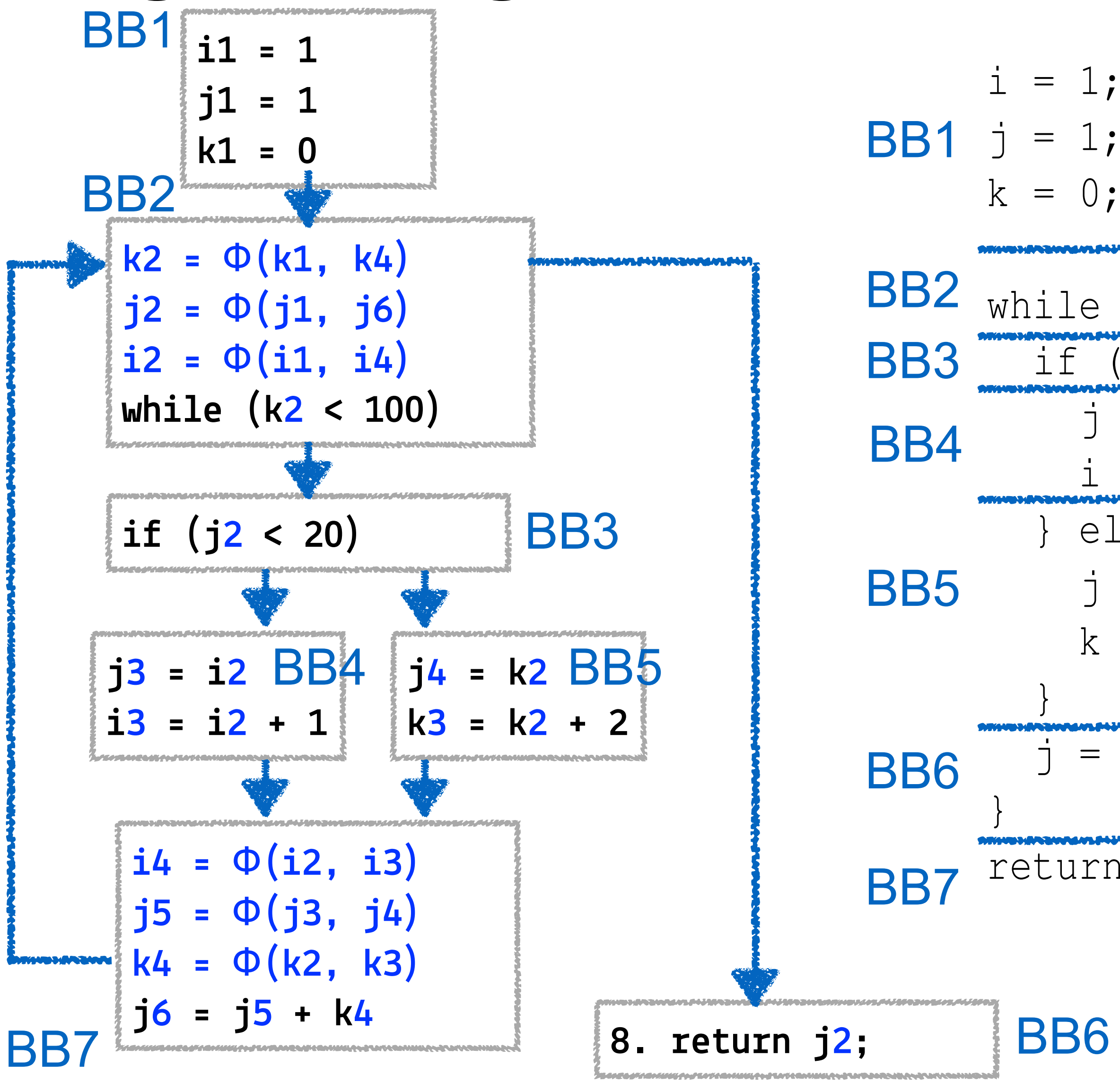


```
BB1  i = 1;
     j = 1;
     k = 0;

BB2  while (k < 100) {
BB3    if (j < 20) {
BB4      j = i;
         i = i + 1;
       } else {
BB5      j = k;
         k = k + 2;
       }
BB6    j = j + k;
     }
BB7  return j;
```

8 Static Single Assignment Form

8.3 Insert the required Phi (Φ) functions into the CFG



BB1	`i = 1;` `j = 1;` `k = 0;`
BB2	`while (k < 100) {`
BB3	`if (j < 20) {`
BB4	`j = i;` `i = i + 1;`
BB5	`} else {` `j = k;` `k = k + 2;` `}`
BB6	`j = j + k;` `}`
BB7	`return j;`

9 Simple Optimizations

9.1 Apply constant folding and constant propagation and give the result

```
int foo(n) {
  int x, y, z, a, b, c;
  x=23;
  y=42;
  z=69;

  a = x;
  a = a + 4*(x+y)*n;
  c = n;
  if (a > 10) {
    b = a * 1 + x * ( a + 0 + a);
    c = x+z;
  }
  a = b * 3;
}
```

First: constant propagation

```
a = 23;
a = 23 + 4 * (23+42) * n;
c = n;
if (a > 10) {
   b = a * 1 + 23*(a+0+a);
   c = 23 + 69;
}
a = b * 3;
```

Then: constant folding

<<< Note that we did not perform dead code elimination

```
a = 23;
a = 23 + 260 * n;
c = n;
if (a > 10) {
   b = a * 1 + 23*(a+0+a);
   c = 92;
}
a = b * 3;
```

9 Simple Optimizations

9.2 Apply algebraic simplification to the result of question 9.1 and give the result

```
int foo(n) {
  int x, y, z, a, b, c;
  x=23;
  y=42;
  z=69;

  a = x;
  a = a + 4*(x+y)*n;
  c = n;
  if (a > 10) {
    b = a * 1 + x * ( a + 0 + a);
    c = x+z;
  }
  a = b * 3;
}
```

After constant folding (from 9.1):

```
a = 23;
a = 23 + 260 * n;
c = n;
if (a > 10) {
  b = a * 1 + 23*(a+0+a);
  c = 92;
}
a = b * 3;
```

After algebraic simplification:

```
a = 23;
a = 23 + 260 * n;
c = n;
if (a > 10) {
  b = a + 23*(a+a);   <<< a*1 and a+0 each replaced by a
  c = 92;
}
a = b * 3;
```

9 Simple Optimizations

9.3 Apply strength reduction to the result of question 9.2 and give the result

```
int foo(n) {
  int x, y, z, a, b, c;
  x=23;
  y=42;
  z=69;

  a = x;
  a = a + 4*(x+y)*n;
  c = n;
  if (a > 10) {
    b = a * 1 + x * ( a + 0 + a);
    c = x+z;
  }
  a = b * 3;
}
```

After algebraic simplification (from 9.2):

```
a = 23;
a = 23 + 260 * n;
c = n;
if (a > 10) {
  b = a + 23*(a+a);
  c = 92;
}
a = b * 3;
```

After strength reduction:

```
a = 23;
a = 23 + 260 * n;
c = n;
if (a > 10) {
  b = 47 * a;
  c = 92;
}
a = b * 3;
```

<<< 23*(a+a) = 46*a; a + 23*(a+a) = 47*a

10 lex

Extend this code to create a scanner that counts the number of (English) vowels (i.e. aeiouAEIOU) and outputs it. Use whitespace as separators between numbers and other text.

```
%{
#include <stdio.h>
enum { END = 256 };
int count;
%}
%%
end             { return END; } // end must come first!
[aeiouAEIOU]    { count++; }
.               { printf("%c", yytext[0]); }
%%
int main(void) {
  int token;
  count = 0;

  while (1) {
    token = yylex();
    if (token == END)
      break;
  }
  printf("\n%d vowels\n", count);
  printf("\nBye!\n");
  return 0;
}
```

```
%{
#include <stdio.h>
enum { END = 256 };
%}
%%
end   { return (END); }
.     { printf("%c", yytext[0]); }
%%
int main(void) {
  int token;

  while (1) {
    token = yylex();

    if (token == END)
      break;
  }
  printf("\nBye!\n");
  return 0;
}
```