CMPT 379 - Fall 2012 - Final Exam

Fill in your name and student id on your Exam Booklet. Write "Final Exam" next to your name.
Provide answers in the Exam Booklet provided to you. Do not answer the questions on this paper.
When you have finished, return your Exam Booklet along with this question booklet.

(1) **Syntax Directed Translation**

Let the synthesized attribute *val* give the decimal floating point value of the binary number generated by S in the following grammar.

$$\begin{aligned} S &\rightarrow L \,.\, L \mid L \\ L &\rightarrow L\, B \mid B \\ B &\rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

For example, on input `101.101`, the integer part of the number is $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$ and the fractional part is $1 \times \frac{1}{2^1} + 0 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} = \frac{5}{8}$ providing the value of $5\frac{5}{8} = 5.625$ for the synthesized attribute $S.val$.

Consider the following attribute grammar for a syntax-directed definition which determines $S.val$ for all strings in the language.

Rules	Syntax-directed definition
$S \rightarrow L$	\$1.in = (0, 1); # inherited attr *in* is a tuple: (first, second) \$0.val = \$1.val.first; # synthesized attr *val* for L is also a tuple
$S \rightarrow L \,.\, L$	\$1.in = (0, 1); \$3.in = (0, −1); \$0.val = \$1.val.first + \$3.val.first;
$L \rightarrow L\, B$	\$1.in = (\$0.in.first+1 , \$0.in.second); \$2.in = (\$0.in.second < 0) ? −(\$1.val.second+1) : \$0.in.first; \$0.val = (\$1.val.first + \$2.val , \$1.val.second+1);
$L \rightarrow B$	\$1.in = (\$0.in.second < 0) ? −1 : \$0.in.first; \$0.val = (\$1.val, 2);
$B \rightarrow 0$	\$0.val = 0;
$B \rightarrow 1$	\$0.val = $2^{0.in}$;

a. (4pts) Draw the parse tree for the input `101.101` and decorate the nodes with the inherited

and synthesized attributes needed to determine the value of $S.val$.

Answer:

```
(S                    # val = (2^2 + 2^0) + (2^-1 + 2^-3) = 5.625
   (L                 # in = (0, 1);  val = (2^2 + 2^0, 4)
     (L               # in = (1, 1);  val = (2^2, 3)
       (L             # in = (2, 1);  val = (2^2, 2)
         (B 1))       # in = 2;  val = 2^2
       (B 0))         # in = 1;  val = 0
     (B 1))           # in = 0;  val = 2^0
   .
   (L                 # in = (0, -1);  val = (2^-1 + 2^-3, 4)
     (L               # in = (1, -1);  val = (2^-1, 3)
       (L             # in = (2, -1);  val = (2^-1, 2)
         (B 1))       # in = -1;  val = 2^-1
       (B 0))         # in = -2;  val = 0
     (B 1)))          # in = -3;  val = 2^-3
```

b. (7pts) Provide a new attribute grammar where you have eliminated left recursion from the grammar. You *must* eliminate left recursion in the following way: for a left recursive rule schema of the type $A \rightarrow A\alpha \mid \beta$ convert it to $A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \epsilon$.

Answer:

Rules	Syntax-directed definition
$S \rightarrow L$	\$1.in = (0, 1); \$0.val = \$1.val.first;
$S \rightarrow L\ .\ L$	\$1.in = (0, 1); \$3.in = (0, -1); \$0.val = (\$1.val.first + \$3.val.first, 0);
$L \rightarrow B\ L'$	\$1.in = \$0.in.second $*$ \$2.val.second; \$2.in = (\$0.in.first+1, \$0.in.second); \$0.val = (\$2.val.first + \$1.val , \$2.val.second + \$0.in.second);
$L' \rightarrow B\ L'$	\$1.in = \$0.in.second $*$ \$2.val.second; \$2.in = (\$0.in.first+1, \$0.in.second); \$0.val = (\$2.val.first + \$1.val, \$2.val.second + \$0.in.second);
$L' \rightarrow \epsilon$	\$0.val = (\$0.in.second < 0) ? (0, \$0.in.first) : (0,0);
$B \rightarrow 0$	\$0.val = 0;
$B \rightarrow 1$	\$0.val = $2^{0.in}$;

c. (4pts) Draw the parse tree for the input `101.101` and decorate the nodes with the inherited

and synthesized attributes using your new attribute grammar without left recursion.

Answer:

```
(S                    # val = (2^2 + 2^0) + (2^-1 + 2^-3) = 5.625
   (L                 # in = (0, 1);  val = (2^2 + 2^0, 3)
     (B 1)            # in = 2;  val = 2^2
     (L'              # in = (1, 1);  val = (2^0, 2)
        (B 0)         # in = 1;  val = 0
        (L'           # in = (2, 1);  val = (2^0, 1)
           (B 1)      # in = 0;  val = 2^0
           (L' eps)))) # in = (3, 1);  val = (0, 0)
   .
   (L                 # in = (0, -1);  val = (2^-1 + 2^-3, 0)
     (B 1)            # in = -1;  val = 2^-1
     (L'              # in = (1, -1);  val = (2^-3, 1)
        (B 0)         # in = -2;  val = 0
        (L'           # in = (2, -1);  val = (2^-3, 2)
           (B 1)      # in = -3;  val = 2^-3
           (L' eps)))) # in = (3, -1);  val = (0, 3)
```

(2) **Code Generation**

The following attribute grammar implements code-generation for a fragment of a programming language.

Rules	Syntax-directed definition
$P \rightarrow S$	\$1.next = newlabel(); \$0.code = \$1.code + label(\$1.next);
$S \rightarrow$ **assign**	\$0.code = "assign";

$S \rightarrow$ **while** '(' B ')' S	begin = \$5.next = newlabel();
	true = \$3.true = newlabel();
	\$3.false = \$0.next;
	instr = "goto" begin;
	\$0.code = label(begin) + \$3.code + label(true) + \$5.code + instr;

$S \rightarrow S\ S$	firstStmt = \$1.next = newlabel(); \$2.next = \$0.next; \$0.code = \$1.code + label(firstStmt) + \$2.code;
$B \rightarrow$ **true**	?
$B \rightarrow$ **false**	?

We assume that the parser interprets the statement concatenation rule $S \rightarrow S\ S$ unambiguously as being right-associative, and that the while statement takes precedence over statement concatenation.

The operator + simply concatenates three-address instructions and labels. We assume that *newlabel()* creates a new label each time it is called (returning `L1`, `L2`, ...), and that *label(L)* attaches label L to the next three-address instruction that is generated, e.g. *label(L1)* would result in `L1:` generated in the output.

a. (4pts) Add the syntax directed definition for the rules $B \rightarrow$ **true** and $B \rightarrow$ **false** .

Answer:

Rules	Syntax-directed definition
$B \rightarrow$ **true**	$0.code = "goto" $0.true;
$B \rightarrow$ **false**	$0.code = "goto" $0.false;

b. (3pts) Provide the output three-address instructions for the input:

```
while (true) assign assign
```

Answer:

```
L3:           # label(begin)
    goto L4   # 3.code from B -> true
L4:           # label(true)
    assign    # 5.code from S -> assign
    goto L3   # instr
L2:           # 1.code + label(firstStmt)
    assign    # 2.code from S -> assign
L1:           # P -> S
```

c. (5pts) We add a new rule to the grammar: $S \rightarrow$ **do** S **while '('** B **')'** . Provide the syntax directed definition for this new rule.

Answer:

begin = newlabel();
instr = "goto" begin;
check = $2.next = newlabel();
true = $5.true = newlabel();
$5.false = $0.next;
$0.code = label(begin) + $2.code + label(check) + $5.code + label(true) + instr;

d. (3pts) Provide the output three-address instructions for the input:

```
do assign assign while (false)
```

Answer:

```
L2:              # label(begin)
    assign       # 2.code from S -> S S
L4: assign
L3:              # label(check)
    goto L1      # 5.code from B -> false
L5: goto L2      # label(true) + instr
L1:              # P -> S
```

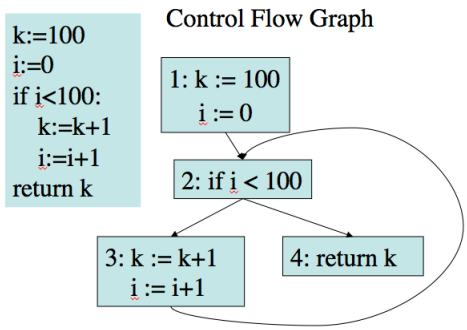
(3) **Static Single Assignment (SSA)**

Examine the following fragment of C++ code:

```
int k=100;
for (int i=0; i<100; i++) k=k+1;
return k;
```

a. (4pts) Draw the control-flow graph for this code. For this part use standard three-address code as your intermediate code and *not* SSA form. You must have exactly 4 basic blocks in your control-flow graph.

Answer:



b. (3pts) Provide the dominance relations for each node in the control-flow graph.

Answer:

$$
\begin{aligned}
D(1) &= \{2, 3, 4\} \\
D(2) &= \{3, 4\} \\
D(3) &= \{\} \\
D(4) &= \{\}
\end{aligned}
$$

c. (3pts) Provide the dominance frontier set for each node in the control-flow graph.

Answer:

$$
\begin{aligned}
DF(1) &= \{\} \\
DF(2) &= \{2\} \\
DF(3) &= \{2\} \\
DF(4) &= \{\}
\end{aligned}
$$

d. (5pts) Provide the SSA form using the dominance frontiers using the iterated dominance frontier algorithm. For each ϕ function provide a reason why it was inserted using the

dominance frontier sets. Do *not* optimize your SSA form intermediate code in any way.

Answer:

- variable i, k in 1. $DF(1) = \{\}$.
- variable i in 2, $DF(2) = \{2\}$, add $\phi(i, i)$ to 2.
- variable i, k in 3, $DF(3) = \{2\}$, add $\phi(k, k)$ to 2, $\phi(i, i)$ is already added.
- variable k in 4, $DF(4) = \{\}$