Final Practice Questions
CS 473, Spring 2020

1 Lexing

a. What kinds of languages are regular expressions unable to recognize? Give a general principle and a
specific example.

Regular expressions don’t have unbounded memory, so they can’t recognize languages that require
storing arbitrarily large numbers, like nesting depth for matching parentheses.

b. Write regular expressions for each of the following languages, or indicate that no such regular expression
exists.

i) Strings over the alphabet \{x, y, z\} in which no x’s appear after the first y (if one exists).
\([xz]^*(yz^*)^*\)?

ii) Binary numbers that are multiples of 4.
\([01]^*00\)

iii) Identifiers that start with a lowercase letter and then have 0 or more alphanumeric characters,
ending in a number.
\([a-z][a-zA-Z0-9]^*[0-9]\)

iv) Strings over the alphabet \{a, b\} that have exactly as many a’s as b’s.
none exists

2 Parsing

a. Write a context-free grammar for each of the following languages.

i) Strings over the alphabet \{a, b\} that have exactly as many a’s as b’s.

\[ S ::= aSbS \mid bSaS \mid \epsilon \]

ii) Arithmetic expressions involving addition and subtraction, with no particular associativity or
precedence.

\[ S ::= \text{number} \mid S + S \mid S - S \]
iii) The same language, but where addition and subtraction are left-associative, and addition has higher precedence than subtraction.

\[
S ::= S - T | T \\
T ::= T + U | U \\
U ::= \text{number}
\]

iv) L-values, where an l-value is either the terminal symbol \text{ID}, \text{l.ID} where \text{l} is an l-value, or \text{l}[n] where \text{l} is an l-value and \text{n} is a number (you should define numbers as another nonterminal in your grammar).

\[
N ::= [0-9] | [0-9]N \\
S ::= \text{ID} | S.\text{ID} | S[N]
\]

b. Consider the following excerpt of a yacc grammar:

\[
\text{exp:} \\
\text{ID} \\
| \text{NUM} \\
| \text{exp PLUS exp} \\
| \text{exp TIMES exp} \\
| \text{REF exp} \quad \text{// memory access}
\]

\[
\text{stm:} \\
\text{ID EQ exp} \quad \text{// assignment} \\
| \text{REF exp EQ exp} \quad \text{// store} \\
| \text{stm SEMI stm} \quad \text{// sequence} \\
| \text{IF exp LBRACE stm RBRACE ELSE LBRACE stm RBRACE} \\
| \text{IF exp LBRACE stm RBRACE}
\]

For each production, write an action that produces an appropriate AST node for that production. Assume the existence of appropriate node constructors for each type of node.

\[
\text{exp:} \\
\text{ID} \quad \{ \text{return A_Ident($1); } \} \\
| \text{NUM} \quad \{ \text{return A_Num($1); } \} \\
| \text{exp PLUS exp} \quad \{ \text{return A_BinOp(A_plus, $1, $3); } \} \\
| \text{exp TIMES exp} \quad \{ \text{return A_BinOp(A_mul, $1, $3); } \} \\
| \text{REF exp} \quad \{ \text{return A_Load($2); } \}
\]

\[
\text{stm:} \\
\text{ID EQ exp} \quad \{ \text{return A_Assign($1, $3); } \} \\
| \text{REF exp EQ exp} \quad \{ \text{return A_Store($2, $4); } \} \\
| \text{stm SEMI stm} \quad \{ \text{return A_Seq($1, $3); } \} \\
| \text{IF exp LBRACE stm RBRACE ELSE LBRACE stm RBRACE} \quad \{ \text{return A_If($2, $4, $8); } \} \\
| \text{IF exp LBRACE stm RBRACE} \quad \{ \text{return A_If($2, $4, \text{NULL}); } \}
\]
3 Semantic Analysis

Consider the following program:

```c
int x = 0;
int *y = malloc(sizeof(int));
while(x < 10){
    int z = 0;
    int y = x;
    while(y < 10){
        z = z + y;
        y = y + 1;
    }
    *y = x;
}
*y = x;
```

a. Write an environment/symbol table showing the type of each variable that is in scope at line 4.

{x : int, y : int*}

b. Write an environment/symbol table showing the type of each variable that is in scope at line 6.

{x : int, y : int, z : int}

c. Write an environment/symbol table showing the type of each variable that is in scope at line 11.

{x : int, y : int*}

4 Intermediate Representations

a. Name one difference between abstract syntax trees and intermediate representation trees.

ASTs have structured control flow nodes (While, If), while IR trees have jumps and labels. (Also: ASTs have field and array accesses while IR trees have pointers and memory accesses, ASTs have type information and IR trees don’t, etc.)

b. Translate each of the following Tiger programs into IR trees. Assume that all variables are temporaries unless otherwise specified.

i) a.field3[2] := 1, where a’s type is

```c
{ field1 : int, field2 : string, field3 : array of int }
```

```
Move

    Mem
    |   Const
    |   1
    Plus
    Mem
    |   Const
    |   2 * W
    Plus
    Temp
    |   Const
    |   2 * W
```

a = 2 * W
ii) if \( b = 0 \) then \( f(0, "a") \) else \( f(1, "b") \), where \( b \) is the second variable in the current stack frame

iii) \( c := \text{myrec} \{ \text{field1} = 3, \text{field2} = "Hi", \text{field3} = \text{myarr} \[ n \] \text{of} \ 0 \} \) where \( \text{myrec} \) is the type from i) and \( \text{myarr} \) is defined as type \( \text{myarr} = \text{array of} \ \text{int} \)

---

```
ii) if b = 0 then f(0, "a") else f(1, "b"), where b is the second variable in the current stack frame

iii) c := myrec { field1 = 3, field2 = "Hi", field3 = myarr [ n ] of 0 } where myrec is the type from i) and myarr is defined as type myarr = array of int
```
c. Suppose we wanted to add a repeat-until loop to Tiger, so that \texttt{repeat e until c} executes the expression \texttt{e}, checks whether the condition \texttt{c} is true, and if not repeats the process.

i) Implement the function \texttt{Tr\_repeat} that translates an AST of the form \texttt{repeat e until c} into an IR tree, without calling any other translation functions. Use \texttt{Temp\_newlabel} to create new labels, and \texttt{T\_\langle kind\rangle} to create a new node of kind \texttt{kind}.

\begin{verbatim}
T_exp Tr_repeat(T_exp e, T_exp c){
    Temp_label start = Temp_newlabel();
    Temp_label done = Temp_newlabel();
    return T_Seq(T_Label(start),
                 T_Seq(T_Exp(e),
                       T_Seq(T_Cjump(T_Eq, c, T_Const(0), start, done),
                             T_Label(done))));
}
\end{verbatim}

ii) Give a different implementation of \texttt{Tr\_repeat} that makes an appropriate call to \texttt{Tr\_while} instead of generating new labels and jumps. You may need to add statements onto the beginning or end of the return value of \texttt{Tr\_while}.

\begin{verbatim}
T_exp Tr_repeat(T_exp e, T_exp c){
    return T_Seq(T_Exp(e), Tr_while(Tr_binOp(A_eq, c, T_Const(0)), e));
}
\end{verbatim}
5 Instruction Selection

Suppose we have a target language with the following tiles, where $d$ indicates a destination register, $s$ indicates a source register, and $c$ indicates a constant:

<table>
<thead>
<tr>
<th>instruction</th>
<th>tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi d, s, c</code></td>
<td>Plus Plus Const Temp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add d, s1, s2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lw d, c(s)</code></td>
<td>Mem Mem Mem Mem</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Which of the following are correct translations of the AST to the target language?

Assume that the register $r0$ always contains the value 0.

i) `addi r1, r0, 2`
\[
\text{add } r2, b, r1 \\
\text{lw } r3, 0(r2) \\
\text{addi } r4, a, 1 \\
\text{add } r5, r3, r4
\]

ii) `lw r1, 2(b)`
\[
\text{addi } r2, a, 1 \\
\text{add } r3, r1, r2
\]

iii) `addi r1, b, 2`
\[
\text{addi } r2, a, 1 \\
\text{add } r3, (r1), r2
\]

iv) `addi r1, b, 0`
\[
\text{lw } r2, 2(r1) \\
\text{addi } r3, a, 0 \\
\text{add } r5, r2, r4
\]

i, ii, iv

b. Of the correct translations above, which represent optimal tilings?

ii

6 Dataflow Analysis

The rules for reaching definition analysis are as follows:

If a node $n$ assigns to a variable $a$, then $\text{gen}[n] = \{n\}$ and $\text{kill}[n]$ is the set of all other nodes that define $a$. Otherwise, $\text{gen}[n]$ and $\text{kill}[n]$ are both empty:

\[
\begin{align*}
\text{in}[n] &= \bigcup_{n' \in \text{pred}(n)} \text{out}[n'] \\
\text{out}[n] &= \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

a. Is this a forward or backward dataflow analysis?

Forward. We can tell because according to the first rule, data flows from the out set of a predecessor node to the in set of its successor.
b. Consider the following CFG:

```
1: x = a

2: y = b

3: if (x = 0)

4: y = y + z
5: y = y - z

6: z = y
```

Write the in and out sets for each node after the first iteration of the analysis.

Your answers may vary depending on the order in which you process nodes. Two possibilities are:

- **Case 1:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{1}</td>
<td>{}</td>
<td>{1}</td>
</tr>
<tr>
<td>out[2]</td>
<td>{2}</td>
<td>out[2]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>in[3]</td>
<td>{}</td>
<td>in[3]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>out[3]</td>
<td>{}</td>
<td>out[3]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>in[4]</td>
<td>{}</td>
<td>in[4]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>in[5]</td>
<td>{}</td>
<td>in[5]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>out[5]</td>
<td>{5}</td>
<td>out[5]</td>
<td>{1, 5}</td>
</tr>
<tr>
<td>in[6]</td>
<td>{}</td>
<td>in[6]</td>
<td>{1, 4, 5}</td>
</tr>
<tr>
<td>out[6]</td>
<td>{6}</td>
<td>out[6]</td>
<td>{1, 4, 5, 6}</td>
</tr>
</tbody>
</table>

- **Case 2:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{1}</td>
<td>{}</td>
<td>{1}</td>
</tr>
<tr>
<td>out[2]</td>
<td>{1, 2}</td>
<td>out[2]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>in[3]</td>
<td>{1, 2}</td>
<td>in[3]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>out[3]</td>
<td>{1, 2}</td>
<td>out[3]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>in[4]</td>
<td>{1, 2}</td>
<td>in[4]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>out[4]</td>
<td>{1, 4}</td>
<td>out[4]</td>
<td>{1, 4}</td>
</tr>
<tr>
<td>in[5]</td>
<td>{1, 2}</td>
<td>in[5]</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>out[5]</td>
<td>{1, 5}</td>
<td>out[5]</td>
<td>{1, 5}</td>
</tr>
<tr>
<td>in[6]</td>
<td>{1, 4, 5}</td>
<td>in[6]</td>
<td>{1, 4, 5}</td>
</tr>
<tr>
<td>out[6]</td>
<td>{1, 4, 5, 6}</td>
<td>out[6]</td>
<td>{1, 4, 5, 6}</td>
</tr>
</tbody>
</table>

c. Finish performing the analysis, and write the final in and out sets.

```
in[1] | {}
out[1] | {1}
in[2] | {1}
out[2] | {1, 2}
in[3] | {1, 2}
out[3] | {1, 2}
in[4] | {1, 2}
out[4] | {1, 4}
in[5] | {1, 2}
out[5] | {1, 5}
in[6] | {1, 4, 5}
out[6] | {1, 4, 5, 6}
```
7 Register Allocation

a. Consider the following program, annotated with live variable information:

```
// live: {v, x}
u = v + 1
// live: {u, v, x}
w = u - v
// live: {u, w, x}
x = x + w
// live: {u, w, x}
y = u - w
// live: {x, y}
z = x + y
// live {z}
```

i) Draw the interference graph for the program.

![Interference Graph](image)

ii) What is the smallest number of colors that can be used to color the graph without spilling? Explain why no smaller number of colors will be enough.

3: there are 3 variables live at once, so 2 colors cannot be enough.

iii) Suppose you have machine registers r1, r2, and r3. Write an allocation of variables to registers such that no two adjacent variables have the same register, spilling if necessary.

u: r1
v: r2
w: r2
x: r3
y: r1
z: r1
8 Loop Analysis and Optimization

a. Consider the following control flow graph:

i) What is the immediate dominator of node 8?
   node 3

ii) Draw the dominator tree for the graph.

iii) List each of the natural loops in the graph, by identifying the header node and all the other nodes in the loop.

   Loop 2 → 1: header node 1, other node 2
   Loop 6 → 3: header node 3, other nodes 4, 5, 6
   Loop 8 → 2: header node 2, other nodes 3, 4, 5, 6, 7, 8

b. Consider the following program:

   x = 0
test:
   if (x > 8) goto done
   z = 2 * x
   v = a + 4
   store(0, v + z)
   x = x + 1
   y = b + x
done:

i) Can the instruction v = a + 4 be hoisted outside of the loop without changing the behavior of the program? Why or why not?
   Yes: a is not modified in the loop, so a + 4 will always have the same value, and there is no use of any other definition of v.

ii) What is the basic induction variable of the loop? What are the derived induction variables in its family?
   The basic induction variable is x, and y and z are derived from it.
iii) Strength reduction on a derived induction variable $j$ has the following steps:
   i. Pick a new variable $j'$ and add an instruction $j' = j' + c$ at the end of the loop, where $c$ is
      the amount by which $j$ increases in each iteration.
   ii. Add an instruction $j' = c_0$ before the beginning of the loop, where $c_0$ is the initial value of $j$.
   iii. Replace the right-hand side of the assignment to $j$ with $j'$.

Perform strength reduction on one of the derived induction variables in the program, and show
the resulting program.

```plaintext
x = 0
z' = 0
test:
  if (x > 8) goto done
  z = z'
  v = a + 4
  store(0, v + z)
  x = x + 1
  y = b + x
  z' = z' + 2
done:

or
x = 0
y' = b + 1
test:
  if (x > 8) goto done
  z = 2 * x
  v = a + 4
  store(0, v + z)
  x = x + 1
  y = y'
  y' = y' + 1
done:
```