

## CS5363 Final Exam

1. (10pts) Write a regular expression for each of the following languages.
  - (a) The set of floating point numbers over the alphabet  $\{0-9, .\}$ . For example, strings “0.52”, “35.” and “100.63” are in the language, but “1.01.5”, “35” (an integer number with no floating point) are not.  
 Solution:  $([0-9]^*.[0-9]^+)|[0-9]^+.[0-9]^*$
  - (b) The set of strings over  $\{1,0\}$  that contain no substring 101. For example, strings “0100” and “10010” are in the language, but string “0101” is not.  
 Solution:  $(1+00+|0^*)^*(1^*|10)$

2. (12pts) Give a context-free grammar for a small graph description language. The terminals of your grammar include integer numbers (denoted by the token *NUM*), ‘(’, ‘)’, ‘;’ and ‘→’. Each node of the graph is represented by an integer number, and each edge is denoted by a ‘→’ connecting two nodes (eg., “3→14” represents an edge from node 3 to node 14). Each graph description is a sequence of paths (here a single node is considered a special path) that starts with ‘(’, ends with ‘)’, and separated by ‘;’.

The following gives an example of a graph description

$(1 \rightarrow 2 \rightarrow 5 \rightarrow 1; 2 \rightarrow 1)$

which describes a graph with three nodes (1, 2 and 5) and four edges ( $1 \rightarrow 2$ ,  $2 \rightarrow 5$ ,  $5 \rightarrow 1$ , and  $2 \rightarrow 1$ ).

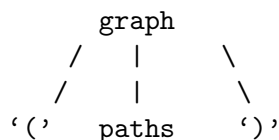
The description “(1; 2)” gives another example, which describes a graph with two nodes (1 and 2) but no edges. Note that “(1;3;)” is not a valid graph description (the second “;” should not be there), neither is the description “(1→)” (missing end point of the edge).

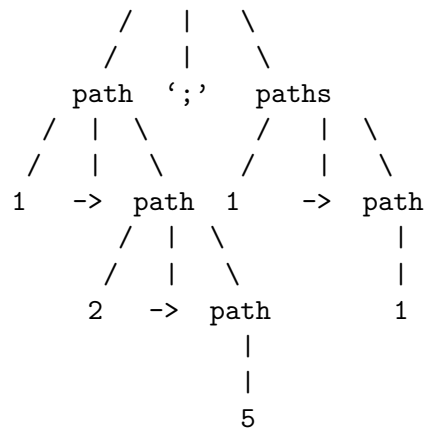
Using your grammar, write a parse tree and an abstract syntax tree for “(1→2→5; 1→1)”.

Solution:

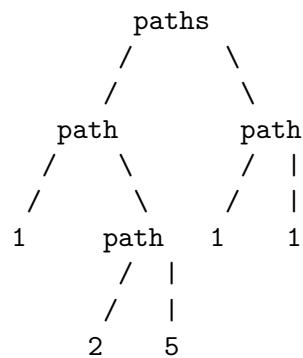
graph ::= ‘(’ paths ‘)’  
 paths ::= path — path ‘;’ paths  
 path ::= NUM — NUM ‘→’ path

Parse tree for the given graph:





Abstract syntax tree:



3. (10pts) For the following Pascal program, draw the set of activation records that are on the stack just prior to the return from function F1. Use access links for non-local data access and use line numbers for return addresses. Draw directed arcs for control and access links. Label the values of local variables and parameters. Label each AR (activation record) with its procedure name.

```

1 program main(input, output);
2   var x : integer;
3   function F1(a : integer ) : integer;
4     begin
5       x := a + 3;
6       F1 = x;
7     end;
8   procedure P;
9     var a: integer;
10    function F2(b : integer) : integer;
11      begin
12        F2 := a + F1(b)
13      end
20  begin
21    a := 5;
  
```

```

22  writeln(F2(a));
23  end;
24  begin
25  P
26  End.

```

Solution:

Each activation record has the following fields (ignoring register save area).

```

control link
access link
return addr
return result addr
parameters
local vars

```

The activation records before returning from F1:

```

main:
    CL: 0 <-|<-----|
    AL: 0  |          |
    RA: 0  |          |
    RRA: 0 |          |
    x: 8   |          |
    P:
    CL -----| <-|  |
    AL -----|  |  |
    RA: 26    |  |  |
    RRA: 0    |  |  |
    a : 5     |  |  |
    F2(a):<-|  |  |
    F2:
    CL  ----|---|<-|  |
    AL  ----|---|  |  |
    RA: 22  |  |  |
    RRA: ---|  |  |
    b: 5    |  |  |
    F1(b):8<-|  |  |
    F1:
    CL -----|-----|  |
    AL -----|-----|  |
    RA: 12   |  |  |
    RRA -----|  |  |
    a: 5

```

4. (8pts) Generate three-address ILOC for the following C statements.

```

a = b * c * a + b - (c - 2);
while (b < c || a < c) {
    b = b * b;
    a = a * a;
}

```

Use short-circuit evaluation for translating boolean expressions. Assume we have an infinite number of registers and that all variables can be allocated to registers without a memory address. Therefore no memory load or store operations are necessary. You can use *ra*, *rb*, *rc* to denote the registers for variables *a*, *b*, and *c* respectively, and use *r1*, *r2*, ... to denote registers allocated for temporary values. You additionally can use the following operations in your translation, where *r1*, *r2*, *r3* are registers, *num* is an integer number, and *L1*, *L2* are labels.

```

PLUS r1, r2 $\rightarrow$ r3
MULT r1, r2 $\rightarrow$ r3
SUB r1, r2 $\rightarrow$ r3
SUBI r1, num $\rightarrow$ r3
COMP r1, r2 $\rightarrow$ r3
COMPI r1, num $\rightarrow$ r3
CBR_LT r1 $\rightarrow$ L1, L2
JUMPI L1

```

Solution:

```

    MULT rb, rc => r1
    MULT r1, ra => r2
    PLUS r2, rb => r3
    SUBI rc, 2 => r4
    SUB r3, r4 => ra
4:   COMP rb, rc => r5
    CBR_LT r5 => 1, 2
2:   COMP ra, rc => r6
    CBR_LT r6 => 1, 3
1:   MULT rb, rb => rb
    MULT ra, ra => ra
    JUMPI 4
3:

```

5. (24pts) Suppose the following C program is given.

```

a := b + c
b := a - d
L1:c := c + 1
if (c > d) goto L3
a := a + d
d := d * c

```

```

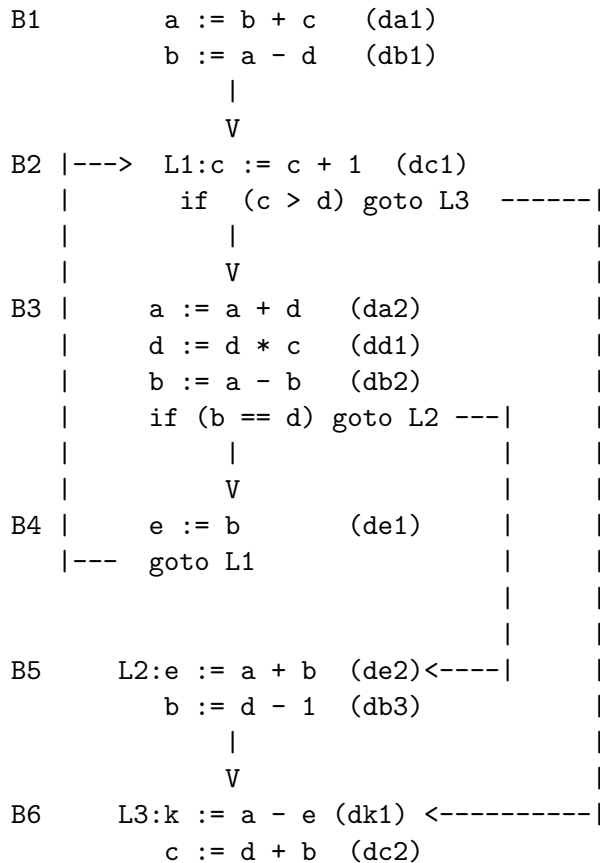
    b := a - b
    if (b == d) goto L2
    e := b
    goto L1
L2:e := a + b
    b := d - 1
L3:k := a - e
    c := d + b

```

- Draw its control flow graph.
- Write the data-flow analysis equation for analyzing the set of reaching variable definitions at the entry of each basic block. A variable definition, e.g., “a = 3”, can reach the entry of a basic block B if there is a control flow path from “a = 3” to block B along which there is no intervening definition to *a*. Show both the intermediate and final result of performing reaching definition analysis. Hint: label each definition site in the code with a unique identifier.
- convert the CFG into SSA form.

Solution:

control flow graph:



Dataflow equation for reaching definition analysis:

$$DefIn(n) = \cup_{m \in pred(n)} (DEDef(m) \cup (DefIn(m) - DefKill(m))) \quad (1)$$

The local sets (DEDef and DefKill) and DefIn sets of each basic block.

Block	DEDef	DefKill
B1	da1,db1	da2,db2,d
B2	dc1	dc2
B3	da2,dd1,db2	da1,db1,d
B4	de1	de2
B5	de2,db3	de1,db1,d
B6	dk1,dc2	dc1

SSA form:

```

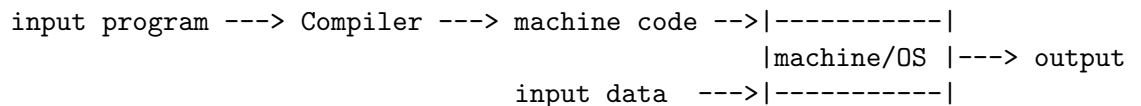
      c0 := ...
B1    a0 := b0 + c0   (da1)
      b1 := a0 - d0   (db1)
      |
      V
B2 |--->L1:a1 := phi(a0,a2)
   |      b2 := phi(b1,b3)
   |      c1 := phi(c0,c2)
   |      d1 := phi(d0,d2)
   |      c2 := c1 + 1   (dc1)
   |      if (c2 > d1) goto L3  ----|
   |      |
   |      V
B3 |      a2 := a1 + d1   (da2)
   |      d2 := d1 * c1   (dd1)
   |      b3 := a2 - b2   (db2)
   |      if (b3 == d2) goto L2 -|
   |      |
   |      V
B4 |      e1 := b3       (de1)
   |---- goto L1
   |
   |
B5    L2:e2 := a2 + b3 (de2)<--|
      b4 := d2 - 1   (db3)
      |
      V
B6    a3:=phi(a2,a1)    <-----|
      e3:=phi(e1,e2)
      b5:=phi(b2,b4)
      d3:=phi(d2,d1)
L3:k1 := a3 - e3 (dk1)
      c3 := d3 + b5   (dc2)

```

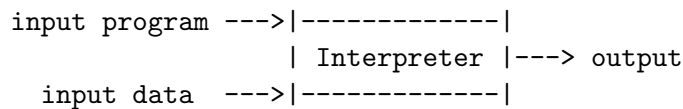
6. (18pts) Use a common definition to explain each of the following pairs of concepts. Briefly summarize how the two concepts in each pair differ and use diagrams or examples to illustrate the difference.

For example, the following explains the concepts Compilation and interpretation. They are two different approaches to implement a programming language. Compilation translates the input program to machine code before evaluating the program, while interpretation evaluates the input computation directly without going through translation. The following diagrams illustrate the compilation and interpretation processes respectively.

Compilation:



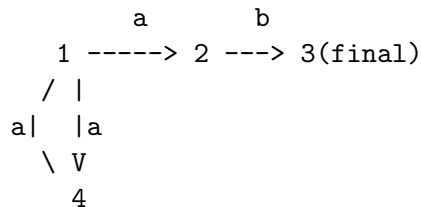
Interpretation:



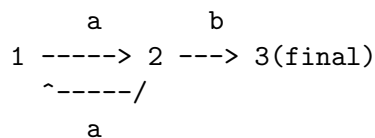
- (a) NFA and DFA (show an example of each automata)

NFA and DFA are two different but equivalent types of finite state machines. NFA is non-deterministic finite automata and can define multiple transition targets on each pair of (state, input string), while a DFA is deterministic finite automata and can define only a unique transition target on each pair of (state, input symbol).

An example NFA:



An example DFA:



- (b) Synthesized attribute and inherited attribute. Give an example context-free grammar with syntax directed definitions for evaluating both types of attributes. They are categorization of attributes in syntax-directed definition (attribute grammars), where every node in the parse tree is associated with a set of attributes and rules are defined to specify the evaluation of these attributes. A

synthesized attribute of a parse tree node  $n$  is always evaluated using attribute values of the children of  $n$ , while an inherited attribute of  $n$  can be evaluated using attribute values of  $n$ 's parent and siblings.

Example: given the following syntax-directed definition

```
A := {A1.pos = A.pos; } A1 + {A2.pos = A1.pos + A1.size; } A2 {A.size=A1.size+A2
    | num {A.size = 1; }
```

where the attribute  $pos$  is inherited attribute and  $size$  is synthesized attribute.

- (c) Type checking and type inference. Give an example expression to illustrate the information required for each analysis.

They are semantic analysis parts of the compilation which examine the input program to ensure that computation is defined correctly on the supported types of values. Type checking assumes that the types of all variables must be declared by the programmer and are known prior to the analysis, while type inference does not require the types of all variables are known and tries to infer the types of variables from the use of variables in the input program.

For example, to perform type checking on the expression “ $a + b * c + 2$ ”, the algorithm requires that the types of all variables ( $a$ ,  $b$  and  $c$ ) are known (declared by programmers). To perform type inference, not all types of the variables are required. If we only know the type of  $a$  is integer, the type inference algorithm can infer that both  $b$  and  $c$  must also have integer type for the expression to be correct.

7. (18pts) Answer the following questions.

- (a) List three program optimizations that you've studied in class. Briefly summarize the optimization in 2-5 sentences. Specifically, what are they used for? What program analysis (i.e., what information regarding the input program) do they require?

Solution:

Three optimizations:

value numbering : for eliminating redundant expressions in the input program. require value numbering analysis where each expression must be associated with a unique number that symbolizes the runtime value of the expression, so that redundant expressions will have an

lazy code motion: for moving partially redundant code (eg., loop invariant code) to less frequently evaluated places. Require availability, anticipatability analysis of expressions and analysis on the placement of expressions.

dead code elimination: for removing useless code from the input program. Need to identify what statements are critical or have external side effects so that they cannot be eliminated.

- (b) List three components that belong to the backend of most modern compilers. Summarize each component in 2-5 sentences. Specifically, what is the functionality of each component? Specify a popular algorithm used in each component.

Three components:

instruction selection: selecting the best sequence of machine instructions as target

of code translation. Popular algorithm: pattern-matching based tree tiling or peephole optimization.

Instruction scheduling: reorder instructions to better utilize resources available in the processor. Popular algorithm: list scheduling.

register allocation: allocate registers to scalar variables in the input program to reduce the memory access cost. Popular algorithm: graph-coloring.

- (c) What are the definition(meaning) of LL(1) and LR(1) parsers? Summarize the parsing steps in 3-6 sentences for each kind of parsers. What are the respective rows, columns, and entries of a LL(1) or LR(1) parse table?

Solution: LL(1): Predictive top-down parser that parses the input program from left to right and tries to find the left-most derivation starting from the starting non-terminal. At each step, tries to find a production to replace the current non-terminal using the next input symbol as lookahead. Each row of a LL(1) parse table corresponds to a non-terminal  $N$  in the language grammar, each column corresponds to an input symbol  $s$ , and each entry in the table defines which production to choose when trying to replace the current non-terminal  $N$  while the next input symbol is  $s$ .

LR(1): Stack-based bottom-up parser that parses the input program from left to right and tries to find the right-most derivation by reducing the input program eventually to the starting non-terminal. At each step, based on the next input symbol, tries to decide whether the symbols in the parsing stack has become the next handle of production to reduce. Each row of the parse table corresponds to a state of the deterministic finite automata that are used to find handles of the input program to reduce to non-terminals, each column corresponds to an input token or a non-terminal symbol in the context-free grammar of the input language, and each entry defines what action to take (shift, reduce or accept) at each automata state when processing the next input token or right after a reduction to a non-terminal.