Quiz

Compiler Construction, Fall 2010

Monday, December 20, 2010

Last Name : ____________________

First Name : ____________________

Exercise	**Points**	**Achieved Points**
1	10	
2	10	
3	20	
4	20	
5	40	
Total	100	

General notes about this quiz

- This is an open book examination. You are allowed to use any written material. You are not allowed to use the notes of your neighbors.
- You have totally **three hours**.
- The points of the questions are not equal. It is advisable not to spend most of your time on the questions with less grade.

Problem 1: Lexical Analysis (10 points)

The increment operator in C++ is ++ (in fact C++ means incrementing the language C). Increment can be both prefix and suffix, so $++x$ and $x++$ effectively increase the value of x. Consider the alphabet $\Sigma = \{+, x\}$. We want to generate all the valid expressions that can be generated using these two symbols. The symbol + of the alphabet can be used in an increment operator (x++), can be a binary operator (x+x) or a unary operator (+x). The lexical analyzer returns the following classes of tokens:

1. **PLUS:** The binary or unary operator +
2. **VAR:** The variable x
3. **INC:** The increment operator ++

For example consider the following expressions and their corresponding tokenizing.

Expression	Tokens	
$+x$	PLUS VAR	
$x++$	VAR INC	
$x+x+x$	VAR PLUS VAR PLUS VAR	
$++x++$	INC VAR INC	
$x+ +x$	VAR PLUS PLUS VAR	
$x+++x$	VAR INC PLUS VAR	
$x+++ +x$	VAR INC PLUS PLUS VAR	

a) Determine for each row of the table if the tokenizing can be done by a longest matching lexical analyzer or not. You can put a ✓ or × beside each row in the table.
For the negative answer justify why the result cannot be generated by a longest matching lexical analyzer.

b) Consider a restriction on the language which allows only three operands ++x, x++ and x and only one operator, binary +. Design a lexical analyzer which can tokenize the generated language by giving a deterministic finite automaton.

c) Do you need the longest match rule for the analyzer in part b?

Problem 2: Parsing (10 points)

Consider the following grammar on $\Sigma = \{\Rightarrow, \texttt{,}, \texttt{Int}\}$. The symbol \$ shows the end of file.

$$\begin{array}{rlll} 1: & S' & \rightarrow S\$ \\ 2: & S & \rightarrow T \Rightarrow S \\ 3: & T & \rightarrow S \texttt{,} T \\ 4: & T & \rightarrow \texttt{Int} \\ 5: & S & \rightarrow \texttt{Int} \end{array}$$

a) Compute the *first* and *follow* of the non-terminals S', S and T.

b) Determine if there is an input for which there exist at least two different parse trees.

c) Assume that the input consists of exactly n tokens, i.e., the lexical analyzer returns exactly n tokens before hitting the \$. For each of the following items determine if they are possible to occur during Earley parsing.

- $(S \rightarrow T\bullet \Rightarrow S\ ,\ 1)$
- $(T \rightarrow S \texttt{,} T\bullet\ ,\ n-1)$

d) Make an equivalent grammar with minimal changes so that the grammar can be recognized by an LL(1) parser.

Problem 3: Type Checking (20 points)

Consider the following piece of code. Determine if the code type-checks according to the rules given in the lecture notes.

- If the code actually type checks then give the complete type derivation tree.
- If there is a type error in the program you should point out the place(s) on the tree in which a type rule is broken.

The derivation tree should be complete and contain all the necessary details. The rules in the tree should be followed from top to bottom without any hidden assumptions.

```
class A
class B extends A {
  def func(a: List[A], b: List[B]): List[A] = b
}
val a: Array[A]
a(0) = new B
val b = a(0).func(a(1),a(3))
```

In the following some type rules are listed which may be useful for solving the problem. Note that it may not be necessary to use all these rules in the solution of this problem and also you may need to use some other rules to make the complete tree.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash x : Int \qquad \Gamma \vdash y : Int}{\Gamma \vdash (x + y) : Int} \qquad \frac{\Gamma \vdash e_1 : T_1 \cdots \Gamma \vdash e_n : T_n \ \Gamma \vdash f : (T_1 \times \cdots \times T_n \rightarrow T)}{\Gamma \vdash f(e_1, \cdots, e_n) : T}$$

$$\frac{\Gamma \vdash x : T_0 \ \Gamma \vdash T_0.m : T_0 \times T_1 \times \cdots T_n \rightarrow T \ \forall i \in \{1, \cdots, n\}.\Gamma \vdash e_i : T_i}{\Gamma \vdash x.m(e_1, \cdots, e_n) : T} \qquad \frac{(x, T) \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash (x = e) : \text{void}}$$

$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 \times T_2 <: T_1' \times T_2'} \qquad \frac{T_1' <: T_1 \cdots T_n' <: T_n \qquad T <: T'}{(T_1 \times \cdots \times T_n \rightarrow T) <: (T_1' \times \cdots \times T_n \rightarrow T')}$$

$$\frac{T <: T'}{\text{collection}[T] <: \text{collection}[T']}\text{(for immutable collection)} \qquad \frac{\Gamma \vdash a : \text{Array}[T] \qquad \Gamma \vdash i : \text{Int}}{\Gamma \vdash a(i) : T}$$

Problem 4: Code Generation (20 points)

Part I. The following code is generated by a compiler for the method `foo`.

a) What does the function compute on the input(s)?

b) Write one possible corresponding Scala method for which this code would be a correct compilation.

```
0:   iload_1
1:   iconst_1
2:   if_icmpgt 5
3:   iconst_1
4:   ireturn
5:   iload_1
6:   aload_0
7:   iload_1
8:   iconst_1
9:   isub
10:  invokevirtual foo
11:  imul
12:  ireturn
```

Part II. Consider the high-level translation of the conditional statement using branch.

```
         [[if (c) sThen else sElse ]] =
          branch(c,nThen,nElse)
nThen:   [[ sThen ]]
          goto nAfter
nElse:   [[ sElse ]]
nAfter:
```

Translate the loop statement and the conditional in the following code using the high-level branch instruction. You do not need to translate the other instructions. Just leave them between translation brackets ⟦ ⟧. Assume that "for" is translated similar to "while".

```
def insertionSort(a: Array[Int]): Array[Int] = {
  var i: Int = 0
  var temp: Int = 0
  var result = a
  for( k <− 1 until a.length){ // translate
    temp = a(k)
    i = k
    while(i > 0 && temp < a(i − 1)) { // translate
      a(i) = a(i−1)
      i = i − 1
    }
    a(i) = temp
  }
  return result
}
```

Problem 5: Register Allocation (40 points)

MIPS is a well-known 32-bit register-based architecture. The CPU uses 32 registers of 32 bits. Data can be loaded and stored from a memory, but we don't care about memory instructions in this problem. Consider the following set of instructions. In the MIPS assembly, when a name starts with \$, this usually means that it is a register. For a register $\$x$, the notation $R[x]$ denotes its value.

Name	Instruction Syntax	Meaning
Branch if less than	**blt** $\$rs, \$rt, \mathit{Label}$	if($R[rs] < R[rt]$) jump to *Label*
Branch if greater than	**bgt** $\$rs, \$rt, \mathit{Label}$	if($R[rs] > R[rt]$) jump to *Label*
Branch on equal	**beq** $\$s, \$t, \mathit{Label}$	if($R[rs] == R[rt]$) jump to *Label*
Load Immediate	**li** $\$a, I$	load the 32-bit constant I to the register $\$a$
Move	**move** $\$rt, \$rs$	$R[rt] \leftarrow R[rs]$
Subtract	**sub** $\$d, \$s, \$t$	$\$d = \$s - \$t$
Add	**add** $\$d, \$s, \$t$	$\$d = \$s + \$t$

In this problem you are to implement a simple program using the given MIPS instructions. Note that you are not allowed to use any other MIPS instruction that is not mentioned above and you cannot invent your own instruction.

a) The function Fibonacci is defined recursively as the following.

$$\mathit{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathit{fib}(n-2) + \mathit{fib}(n-1) & \text{if } n \geq 2 \end{cases}$$

For example, we can compute the value fib(4) as the following.

$\mathit{fib}(3) = \mathit{fib}(1) + \mathit{fib}(2) = 1 + \mathit{fib}(0) + \mathit{fib}(1) = 1 + 0 + 1 = 2.$

Implement *fib* using the given MIPS instructions. For this purpose use a set of temporary variables $\{\$t0, \cdots, \$t31\}$ instead of the actual registers. Assume that the value n is initially in the temporary variable $\$t0$ and the final result value should also be stored in $\$t0$.

b) Build the control-flow graph.

c) Compute the set of live variables before and after each instruction.

d) Draw the interference graph and color it.

e) Show the generated code with assigned actual registers. Assume the actual MIPS registers are $\{\$r_0, \cdots, \$r_{31}\}$.