Quiz

Compiler Construction, Fall 2011

Wednesday, December 21, 2011

Last Name : _____

First Name : _____

Exercise	**Points**	**Achieved Points**
1	20	
2	25	
3	20	
4	20	
5	25	
Total	110	

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 **3 hours 45 minutes**.
- It is advisable to do the questions you know best first.
- 100 points already count as perfect score.

Problem 1: Lexical Analysis (20 points)

Consider the alphabet $\Sigma = \{ +, : \}$. Scala's List object defines (among others) the following operators:

Concat	*++*
Concat2	*++:*
Prepend	*+:*
Append	*:+*
Add	*::*
AddList	*:::*

a) Determine the tokenizing of the following strings using the longest-match rule.

i) $+ + : : : + : : + + :$

ii) $+ + : + + : : : : : + + :$

b) Design a lexical analyzer which can tokenize the above language of operators by giving a deterministic finite automaton and explain how it will use this automaton to tokenize the following string. Make sure to show how your lexical analyzer will return the tokens by name, i.e. it should not just accept or reject the input.
$: : + : + + + :$

Problem 2: Parsing (25 points)

Warm-up: Consider the following grammar:

$$S' \mapsto S\$ \tag{1}$$
$$S \to AAAA \tag{2}$$
$$A \to a \tag{3}$$
$$A \to E \tag{4}$$
$$E \to \epsilon \tag{5}$$

a) Describe precisely the language defined by this grammar.

b) Give the Earley sets obtained by parsing the input "a".

Implementing the Earley parsing algorithm: (Consider leaving this part for the end)
We now consider grammars whose starting non-terminal is S' and in which the only rule involving S' is $S' \mapsto S\$$. We denote non-terminals by capital latin letters (like A), terminals by lower case latin letters (like a), and strings of terminals and non-terminals by greek letters (like α).
Recall that when computing an Earley set as described in the course, an Early parser needs to select an arbitrary item in the set and apply scanning, prediction, or completion to it. When implementing an Earley parser, we need to decide how to make the choice of item to consider. If we make an arbitrary choice we may select an item which has already been processed and hence do useless computation.
Instead, it may be convenient to represent the Earley sets as linked lists. These lists can then be used as work queues that prevent the needless processing of already processed items.
Consider an implementation of the Earley parsing algorithm that uses lists of items L_i to represent the Early sets S_i. Given a word $x_1 x_2 \cdots x_n$ the algorithm works as follows: First, L_0 is initialized to the list $([S' \to \bullet S\$, 0])$. The algorithm starts by computing L_0, then L_1, then L_2, and so on until L_n. To compute L_i, $i \in [1..n]$, the algorithm performs a **single** traversal of the list L_i, from head to tail. During the traversal of L_i, for each element e of L_i, the algorithm performs Scanner(e, i) **then** Predictor(e, i) **then** Completer(e, i), as described below.

- Scanner(e, i): if $e = [A \to \cdots \bullet a \cdots, j]$ and $x_{i+1} = a$ then **append** $[A \to \cdots a \bullet \cdots]$ to the list L_{i+1}.
- Predictor(e, i): if $e = [A \to \cdots \bullet B \cdots, j]$ and $[B \to \bullet\alpha, i]$ is not already in L_i, then **append** $[B \to \bullet\alpha, i]$ to L_i for all items $B \to \alpha$ in the grammar.
- Completer(e, i): if $e = [A \to \cdots \bullet, j]$ then **append** $[B \to \cdots A \bullet \cdots, k]$ to the list L_i for all items $[B \to \cdots \bullet A \cdots, k]$ in L_j.

The algorithm accepts the word if and only if the list L_n contains the item $[S' \to S\$\bullet, 0]$ at the end of the execution. Note that appending means inserting an element at the **end** of a list. Hence this algorithm is **different** from the one described in the course because it specifies a particular order in which to process items. Also note that the algorithm appends Earley items to a list while it traverses it.

c) Give the Earley sets computed by the algorithm described above when parsing the input "a". Is the algorithm correct?

d) Is the algorithm correct if the grammar used has no nullable non-terminals?

e) Propose a fix to the algorithm.

Justify your answer in each case!

Problem 3: Type Checking (20 points)

Consider a language similar to the one used in the course on type checking (the precise definition of the language does not matter).
We add a polymorphic type $Option[T]$ to the language, where T can be any existing type, and a polymorphic map type $Map[U, V]$ with get() and set() operations, where U and V can be any existing types. The only members of the $Option[T]$ type are $None$ and $Some(t)$ for any t of type T. Both $Option$ and Map are immutable. We extend the expression syntax as follows:

$$expr \rightarrow \text{None} \tag{1}$$
$$expr \rightarrow \text{Some}(expr) \tag{2}$$
$$expr \rightarrow \text{new Map}[type,type]() \tag{3}$$
$$expr \rightarrow ident.\text{get}(expr) \tag{4}$$

We extend the statement syntax as follows:

$$stmt \rightarrow \text{case } expr \text{ of None => } stmt \mid \text{Some}(ident) \text{ => } stmt \tag{5}$$
$$stmt \rightarrow ident.\text{set}(expr, expr) \tag{6}$$

where "|" is a character in the language (not a separator for grammar rules) and *ident* is an identifier.
Informally, the semantics of the extensions is the following:
$var.get(expr)$ (line 4) returns a value of an $Option$ type. The value $None$ indicates that the key $expr$ is not in the map and a value $Some(t)$ indicates that the map associates the value of $expr$ to t. The pattern matching expression in line 5 has the same meaning as in Scala. Notably, it should be possible to use *ident* inside the statement in the $Some(ident)$ case. Since Map is immutable, $var.set(expr, expr)$ at line 6 returns a new updated map object.

a) Complete the following type rule templates so that they are consistent with the described meaning of the extensions and sufficient to type check the extension if we exclude subtyping.

$$\frac{\cdots}{\Gamma \vdash None : \cdots} \qquad \frac{\cdots}{\Gamma \vdash Some(t) : \cdots} \qquad \frac{\cdots}{\Gamma \vdash \text{new Map}[U, V]() : \cdots} \qquad \frac{\cdots}{\Gamma \vdash x.get(t) : \cdots}$$

$$\frac{\cdots}{\Gamma \vdash \text{case } expr \text{ of } None \text{ => } s_1 \mid Some(t) \text{ => } s_2 : \cdots} \qquad \frac{\cdots}{\Gamma \vdash x.set(e_1, e_2) : \cdots}$$

Continued on the next page...

b) Using the rules defined in a) and the four rules below, type-check the following statement in the environment
$\Gamma = \{x \mapsto Map[Int, String]; \quad y \mapsto Int; \quad println \mapsto String \to void\}$. by giving its full type derivation tree.

```
case x.get(y) of
  None => println(‘‘key not found’’)
  | Some z => println(‘‘key maps to:’’); println(z)
```

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash f : (T_1 \to T)}{\Gamma \vdash f(e_1) : T} \qquad \frac{\Gamma \vdash t_1 : void \quad \Gamma \vdash t_2 : void}{\Gamma \vdash t_1; t_2 : void} \qquad \frac{}{\Gamma \vdash \text{"}chars\text{"} : String}$$

c) Give sound subtyping rules for the option and map types. These rules should be as permissive as possible. Explain your choices.

Problem 4: Code generation (20 points)

In the following exercise we consider compilation to a stack machine that uses JVM instructions.
Part 1: Consider the high-level translation of the conditional statement.

```
                 ⟦if (c) sThen else sElse ⟧ =
                 ⟦c⟧
                 if_eq nElse
nThen:           ⟦ sThen ⟧
                  goto nAfter
nElse:           ⟦ sElse ⟧
nAfter:
```

Suppose that we extend our language with an additional repeat-until loop construct, which executes the code in the body of repeat until the condition *cond* evaluates to true:

```
repeat {
  ...
}
until (cond)
```

a) Give the high-level translation of this construct in the style of the conditional statement above.

Part 2: Consider the following function for modular exponentiation, i.e. for computing b^e mod m. Doing the computation directly is slow especially for large numbers, since the intermediate values in the computation of b^e can become rather large. The following code does the computation smarter using the "right-to-left binary" method.

```
 1:  def modular_pow(b: Int, e: Int, m: Int): Int = {
 2:    var result = 1
 3:    var exponent = e
 4:    var base = b
 5:    while (exponent > 0) {
 6:        if ((exponent & 1) == 1)
 7:           result = (result * base) % m
 8:        exponent = exponent >> 1
 9:        base = (base * base) % m
10:    }
11:    return result
12:  }
```

In the code above, `&` performs bitwise AND of its operands and `x % y` computes the remainder of the division of `x` by `y`.

b) Give the high-level translation of the **whole** function, i.e. translate the control-flow statements in the style of the conditional above. You can leave the code for the expressions on lines 7, 8, and 9 un-translated by writing for example ⟦result = (result * base) % m⟧ in their case. Do not forget to also give a local variable table that contains the names of variables and their corresponding slots.

c) Give the bytecode for line 7 and for the condition of the if-statement *(exponent & 1) == 1*.

These are selected bytecode instructions, mostly for integers, some of which you can use in this exercise.

iload_x	Loads the integer value of the local variable in slot x on the stack. $x \in \{0, 1, 2, 3\}$
iload X	Loads the value of the local variable pointed to by index X on the top of the stack.
iconst_x	Loads the integer constant x on the stack. x $\in \{0, 1, 2, 3, 4, 5\}$.
istore_x	Stores the current value on top of the stack in the local variable in slot x. $x \in \{0, 1, 2, 3\}$
istore X	Stores the current value on top of the stack in the local variable indexed by X.
ireturn	Method return statement (note that the return value has to have been put on the top of the stack beforehand.
iadd	Pop two (integer) values from the stack, add them and put the result back on the stack.
isub	Pop two (integer) values from the stack, subtract them and put the result back on the stack.
imult	Pop two (integer) values from the stack, multiply them and put the result back on the stack.
idiv	Pop two (integer) values from the stack, divide them and put the result back on the stack.
irem	Pop two (integer) values from the stack, put the result of $x_1 \% x_2$ back on the stack.
ineg	Negate the value on the stack.
iinc x, y	Increment the variable in slot x by amount y.
ior	Logical OR for the two integer values on the stack.
iand	Logical AND for the two integer values on the stack.
ixor	Logical XOR for the two integer values on the stack.
ifXX L	Pop one value from the stack, compare it zero according to the operator XX. If the condition is satisfied, jump to the instruction given by label L. XX $\in$ { eq, lt, le, ne, gt, ge, null, nonnull }
if_icmpXX L	Pop two values from the stack and compare against each other. Rest as above.
goto L	Unconditional jump to instruction given by the label L.
pop	Discard word currently on top of the stack.
dup	Duplicate word currently on top of the stack.
swap	Swaps the two top values on the stack.
aload_x	Loads an object reference from slot x.
aload X	Loads an object reference from local variable indexed by X.
iaload	Loads onto the stack an integer from an array. The stack must contain the array reference and the index.
iastore	Stores an integer in an array. The stack must contain the arrayreference, the index and the value, in that order.

Problem 5: Dataflow Analysis (25 points)

Consider the following code fragment. Assume that the function *input()* returns an integer in the range $[-128, 127]$. We want to perform a range analysis of the variables x and y using the domain of intervals. That is, at any program point each variable can have the following value:

- empty interval (noted $\perp$, bottom element of the lattice)
- regular interval $[a, b] = \{x \mid a \leq x \leq b\}$, with $-128 \leq a \leq b \leq 127$.
- $\top$ (top of the lattice, denoting any value is possible, including error values stemming from division by zero)

Operations on intervals are defined in the usual way:

$$[a, b] \circ [c, d] = [min(S), max(S)] \text{ where } S = \{x \circ y \mid a \leq x \leq b \ \&\& \ c \leq y \leq d\}$$

where $\circ \in \{+, -, *, /\}$.

```
var x = 1
var y = input()

if (x == y) {
  while (y < 5) {
    y = y + 1
    x = x + y + 2
  }
} else if (y <= 3 && y >= -7) {
  y = y * y
} else {
  y = x
}

val z = x/y
```

a) Draw the control flow graph for the code. Make sure your transitions only have simple statements.

b) Now run the dataflow analysis as done in class using the domain of continuous intervals. Indicate the ranges of the variables x and y at each point in your control flow graph. Recall that the join operation on intervals is defined as follows:

$$[a, b] \vee [c, d] = [min(a, c), \ \ max(b, d)]$$

c) What are the possible ranges of the variables at the end of the code fragment as computed by the analysis? In particular, what can you say about the run-time safety of the last statement?