Quiz

Compiler Construction, Fall 2012

Wednesday, December 19, 2012

Last Name : _____

First Name : _____

Exercise	Points	Achieved Points
Total	0	

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.
- Please write the answer of each question on a **separate sheet**.
- Please **return all sheets with quiz questions**, regardless whether you wrote something on them or not. You are free to use your own paper for writing or ask us for paper.

Problem 1: Lexical Analysis (10 points)

Consider the following types of tokens denoting different definitions of identifiers:

OP	$(+ \mid * \mid =)^+$
STRINGID	‘\‘ ‘ $stringLit^+$ ‘\‘ ‘
VARID	$letter\ \ (letter \mid digit)^*\ \ (\ (letter \mid digit \mid \text{‘\_‘})^* \mid\ \text{‘\_‘}(+ \mid * \mid =)^*\ )$

where we take $letter = [a - zA - Z]$, $digit = [0 - 9]$ and $stringLit = [a - zA - Z0 - 9 + * =]$. That is, an identifier can be a string of one or more operator characters (+, *, =), or it can be any string literal between backticks (the identifier name is then taken without the backticks). Or it can start with a letter, followed optionally by a sequence of letters and digits and can then be optionally followed by *either*

- a combination of letters, digits and underscores *or*
- an underscore followed by any number of operator characters

Hence, `x`, `xId_98xyz` and `xId_++=` are valid identifier, but `xId_+xy` is not.
In addition, suppose that the language that we consider has two keywords `case` and `def`, which belong to the token class KEY.

a) **[2 pts]** Determine the tokenizing of the following strings using the longest-match rule, including the type of each token.

i) `big_bob++=‘def‘`

ii) `+*‘case‘type_x==func123_def=+=case**def_77`

b) **[8 pts]** Design a lexical analyzer which can tokenize the above language of operators by giving a deterministic finite automaton. You can use *letter*, *digit*, and *stringLit* where applicable for the respective characters. Please create a simpler version of lexical analyzer that accepts only one token at a time (the longest one possible for a given input), as opposed to one accepting a sequence of tokens; to process a sequence of tokens, the lexical analyzer would be invoked repeatedly until the end of the input is reached.

Problem 2: Grammars (10 points)

Consider the following grammar:

```
Exp -> Exp + Exp
Exp -> Exp < Exp
Exp -> Exp * Exp
Exp -> Exp : Exp
Exp -> num
Exp -> ( Exp )
```

We now want to modify the grammar such that the following precedence rules are enforced:

`*` `+` `<` `:`

where $*$ binds more tightly than any other and : binds less tightly than any other.
In addition, the following associativity should be respected:

- $<$ is non-associative, i.e. an expression like $5 < 6 < 7$ is not permitted
- : is right associative, i.e. $5 : 6 : 7$ should be parsed as $5 : (6 : 7)$
- $+, *$ are left associative, i.e $5 + 6 + 7$ should be parsed as $(5 + 6) + 7$

a) **[8 pts]** Write an unambiguous grammar that derives the same set of strings from Exp as the grammar above and whose parse trees respect the left and right associativity of operators.

b) **[2 pts]** Show the parse tree for $2 : 3 < 4 + 5 : 6 * 7 * 8$ using the unambiguous grammar.

Problem 3: Parsing (20 points)

Consider the following grammar for postfix expressions:

```
E -> EE+
E -> EE*
E -> EE-
E -> num
```

That is, instead of writing $1+2$ we write 12+. Recall that the advantage of this notation is that one does not need parentheses. Precedence is uniquely determined by the order of the operands and operators. For example, instead of writing $3-(4*5)$ in a conventional notation, we write "3 4 5 $*$ $-$", whereas instead of $(3-4)*5$ we write "3 4 $-$ 5 $*$" or "5 3 4 $-$ $*$".

a) **[5 pts]** Use the generalized CYK parsing algorithm to parse the string `2 3 4 + 5 - *`. List the set of all triples (E, i, j) that indicate that E can derive substring from i to j. Give the resulting parse trees.

b) **[15 pts]** Is this grammar ambiguous or not? If it is ambiguous, show an input for which the CYK parser discovers two different parse trees. If it is not ambiguous, prove that every input has at most one parse tree.

Problem 4: Type checking and soundness (25 points)

Part 1: Suppose that we have

```
class Phone {
  def getNumber: Int = { ... }
  def call (n: Int) = { ... }
}
class MobilePhone extends Phone {
  // more functionality
}
class AntiquePhone(anno: Int) extends Phone {
  // more functionality
}
```

a) [10 pts] Using the type checking rules from class and from the homeworks, show that the following code type checks. Don't forget to state the environments and the subtyping relations. (Use may want to use the shorthands `P, MP, AP` for `Phone, MobilePhone, AntiquePhone` respectively. Also, you can split your type derivation tree in pieces, but please make sure it is clear which part belongs where.)

```
var a: Phone
var b: MobilePhone
a = new AntiquePhone(1981)
b = new MobilePhone
a.call(b.getNumber)
```

Part 2: Now consider the following language where we have only object types with predefined inheritance hierarchy and method names, with no primitive values such as Int or Bool. We focus on type checking sequences of variable declarations, method calls, constructor invocations, and assignments.
Expressions we consider can be of one of the following forms, where T can denote any regular object type:

- `x.methodName(p)` standard method call (for this exercise assume at most one parameter), where x and p are variable names;
- `new T()` class constructor (for this exercise you can assume no constructor parameters);
- `null` special value that can be assigned to any variable.

Statements can have one of the following forms:

- `var x:T = e;` variable declaration with simultaneous assignment.
- `x = e;` standard assignment

The goal of our type system is preventing null-pointer exceptions. We introduce for each regular object type T the *null-annotated* types

- T^+ meaning that a variable of this type may also be `null`
- T^- meaning that a variable of this type cannot be `null`

Clearly, it holds $T^- <: T^+$.
The programmer should still only write type T in the code, but the type checker will choose one of T^+ and T^- for type checking. The type of variable is determined at its declaration and always has the same type as the expression that is assigned to it.
For example, if `val x:T = null;` then variable x has type T^+ and if `val x:T = new T;` then variable x has type T^-. Note that we do not want to perform any global type inference; the type of a variable is determined once and for all at the declaration time.
The program will have a runtime error, when trying to call a method on a `null` value. Thus, the **soundness property** that we are interested in is the following:
If the program type checks, runtime errors from dereferencing a null value cannot occur.

b) **[8 pts]** Give *sound* type rules for this language. Remember, that the types in your rules have to be T^- or T^+, T only appears in the code.

To simplify matters, you can assume for this exercise that methods always accept `null` as a parameter and may also return `null`.

c) **[7 pts]** Explain why your type rules are sound. An example of an unsound (broken) type system is one that allows assignment $x = y$; when $y : T^+$ and $x : T^-$. Argue that your system is sound and that well-typed programs do not execute method calls on null receivers. For this, show that, if a program type checks and a variable x is declared as having type T^-, then during program execution x will never store `null`. Your argument should be detailed enough so that it would *not* work for some unsound type system, so your argument should refer to your type rules.

Problem 5: Code generation (15 points)

In the following exercise we consider compilation to a stack machine that uses JVM instructions. 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:
```

a) [5 pts] Suppose that we extend our language with an additional do-until loop construct, which executes the code in the body at least once and stops executing it when the condition condition *cond* evaluates to true:

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

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

b) [10 pts]

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

```
1  def bubbleSort(arr: Array[Int]) {
2        var swapped = true
3        var j = 0
4        var tmp = 0
5        while (swapped) {
6              swapped = false
7              j += 1
8              for ( i <− 0 until (arr.length − j)) {
9                    if (arr(i) > arr(i + 1)) {
10                         swap(arr, i, i+1)      //swaps values in positions i and i+1
11                         swapped = true
12                   }
13             }
14       }
15   }
```

Problem 6: Data flow analysis (20 points)

Consider the following code fragment. Assume that the function *input()* returns a (random) integer in the range $[-5, 5]$ (i.e. -5 and 5 included). Suppose our language now also allows `break` and `continue` statements within loops with the following meaning:

continue jump out of the current loop iteration and continue with the next loop iteration

break jump out of the current loop iteration and continue executing after the loop

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

```
1    var x = 2
2    var y = input()
3
4    while ( x < 5 ) {
5      y = y + 2*x
6
7      if (y >= 25)
8        break
9      else if (y < 15)
10      continue
11     else
12       x = x + 1
13   }
14
15   val z = x/y
```

a) **[5 pts]** Draw the control flow graph for the code. (You don't need to transform the assignment in line 5 into simple statements.)

b) **[13 pts]** Now run the dataflow analysis as done in class using the domain of continuous intervals until convergence. 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:

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

c) **[2 pts]** What are the possible ranges of the variables at the end of the code fragment as computed by the analysis? In particular, what does the analysis say about the run-time safety of the last statement?