Quiz solutions

Compiler Construction, Fall 2012

Wednesday, December 19, 2012

Last Name : _____

First Name : _____

Exercise	**Points**	**Achieved Points**
1	10	
2	10	
3	20	
4	25	
5	15	
6	20	
Total	100	

Problem 1: Lexical Analysis (10 points)

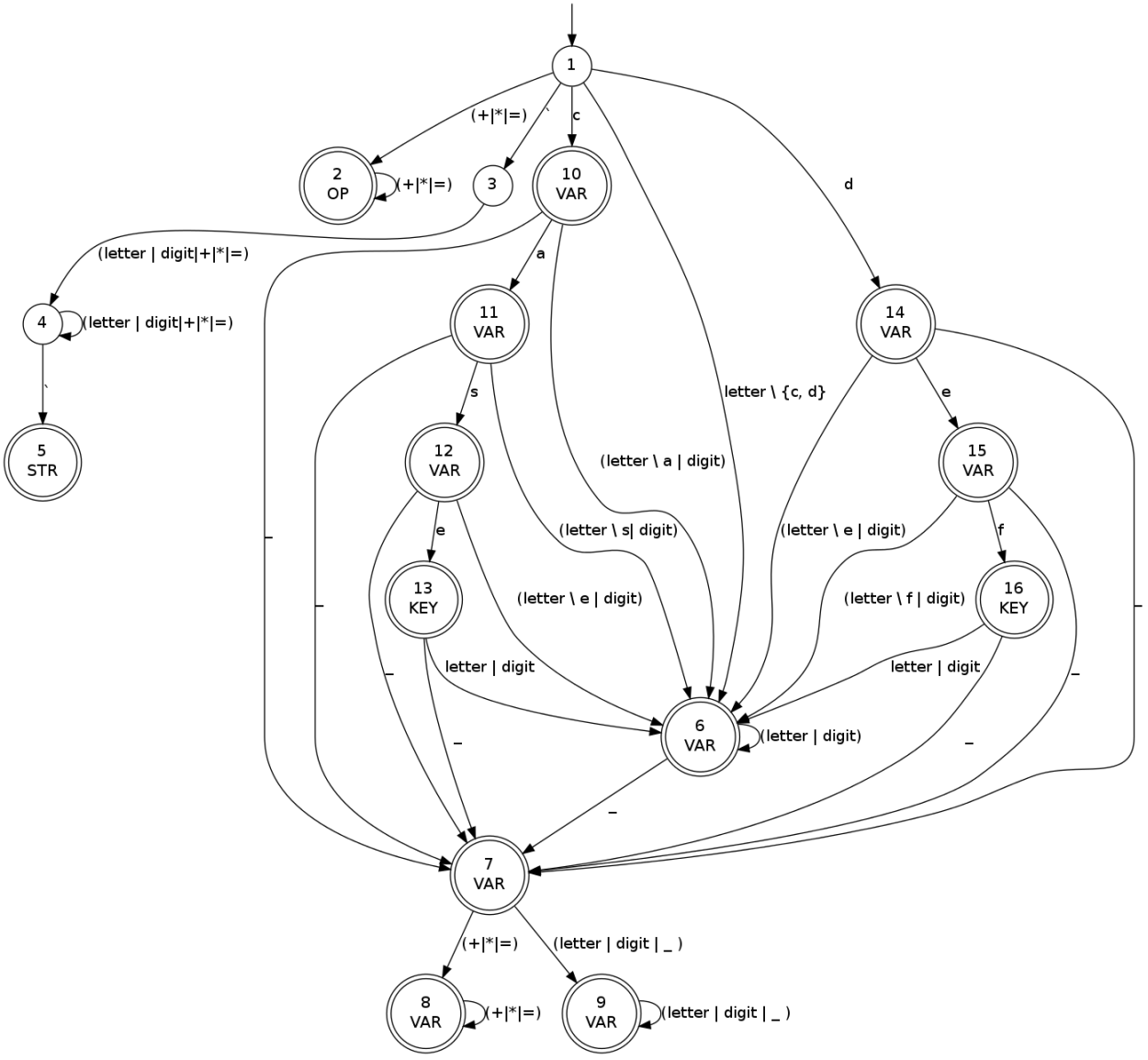
a) [2 pts]

```
big_bob | ++= | ‘def‘
VARID      OP    STRINGID

+* | ‘case‘ | type_x | == | func123_def | =+= | case | ** | def_77
OP   STRING    VARID   OP    VARID        OP    KEY    OP    VARID
```

b) [8 pts]

In the automaton, VAR stands for VARID, and STR for STRINGID. All missing links go to the error state.



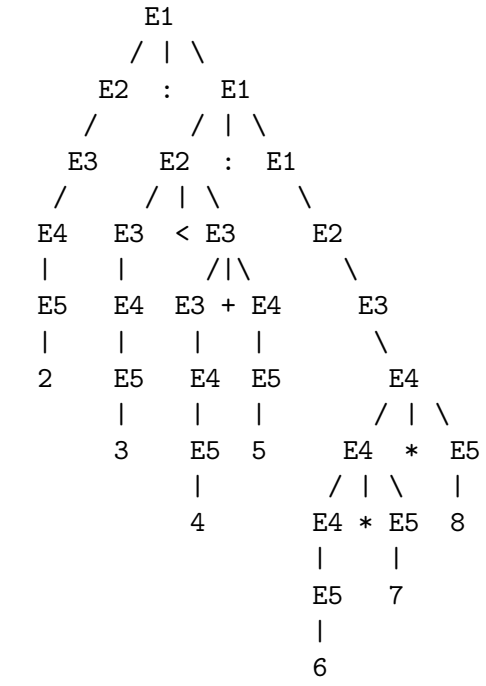
Problem 2: Grammars (10 points)

a) [8 pts]

E_1 -> E_2 : E_1 | E_2
E_2 -> E_3 < E_3 | E_3
E_3 -> E_3 + E_4 | E_4
E_4 -> E_4 * E_5 | E_5
E_5 -> (E_1) | num

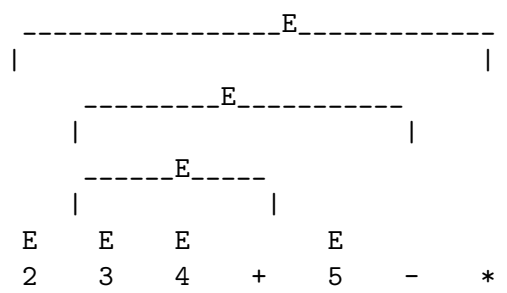
b) [2 pts]

2: ((3 < (4+5)) : ((6*7)*8))



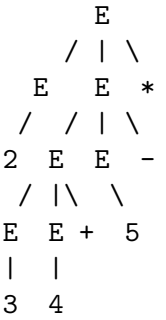
Problem 3: Parsing (20 points)

a) [5 pts]
The CYK algorithm runs as follows:



Thus we get the following triples:
(E, 0, 1), (E, 1, 2), (E, 2, 3), (E, 4, 5), (E, 1, 4), (E, 1, 6), (E, 0, 7)

The only parse that we get is the following:



b) [15 pts]
We claim that the grammar is not ambiguous.

There are two ways to show this.

First, consider parsing the language in reverse. For this we reverse all the grammar rules such that `E -> EE+` becomes `E -> +EE`. The resulting grammar is LL(1) as each grammar rule starts with a distinct nonterminal. Thus, we can parse the reverse string unambiguously with an LL(1) parser, and reverse the resulting parse tree to obtain the parse tree of the original grammar.

Secondly, we can show this by induction on the length of the input. In the following, we assume that the string can be parsed.

The inductive hypothesis is: For every string w of length n, there exists at most one suffix v of w (that is, v such that there exists u for which $w = uv$) such that v belongs to the language of the grammar. Moreover, if such v exists, it has a unique parse tree.

base case: n = 1 The only string that can be parsed successfully is `E -> num`. The unique suffix and parse tree is `num`.

(This is not needed for the proof, but we can observe that for $n = 2$ the only suffix that can be parsed is of the form `num`. For $n = 3$, the strings that can be parsed successfully are all of the form `num num op` for some `op`. The parser must choose one of `E -> EE+`,

`E -> EE*` or `E -> EE-`. The suffix is then the entire string and since the final character determines the applied rule uniquely, the parse tree is unique.)

inductive case: Let $n \geq 0$. Assume that the inductive hypothesis holds for all strings of length i for $i \leq n$ and we prove it for $n + 1$.

There are two possibilities: either the string ends with a number, or with an operator. If the string ends with a number then the unique suffix is simply this number, which has only one parse tree.

If the last token is an operator, the suffix is the string that can be parsed by applying one of the first three rules. Which one is determined (uniquely) by the operator. Consider the string of length n that we obtain from taking our string without the last character.

By inductive hypothesis, the string has a unique suffix of length m, which determines the second E in the grammar rule. That is, the split of the strings into the first E and the second E is uniquely determined. Also, the string up to position $n - m$ also has a unique suffix, which determines the first E of the grammar rule. Hence, either the string cannot be parsed at all, or the suffix is unique and has a unique parse tree.

(If the entire string can be parsed, then the suffix is the entire string, and thus the parse tree is unique.)

Problem 4: Type checking (25 points)

Let us be somewhat generous with grading this question. There are different ways of formalizing type checking rules and, as long as the rules used make some sense and the key steps of derivations are given, we can accept them.

a) [10 pts]
The environments are the following:

$$\begin{aligned}\Gamma &= \{(Phone, void \to Phone), (AntiquePhone, Int \to AntiquePhone),\\ &\quad (MobilePhone, void \to MobilePhone)\}\\ \Gamma_P &= \{(getNumber, void \to Int), (call, Int \to void)\} \oplus \Gamma\\ \Gamma_{MP} &= \Gamma\\ \Gamma_{AP} &= \Gamma\end{aligned}$$

and we have the following subtyping relations:
$AntiquePhone <: Phone$ and $MobilePhone <: Phone$

We now give the type derivation tree for the overall code:

$$\dfrac{\dfrac{\Gamma_1 \vdash \text{a = new AP(1981): void} \quad \Gamma_1 \vdash \text{b = new MP: void} \quad \Gamma_1 \vdash \text{a.call(b.getNumber): void}}{\Gamma_1 = \Gamma \oplus \{(a, P), (b, MP)\} \vdash \text{(a = new AP(1981); b = new MP; a.call(b.getNumber): void}}}{\Gamma \vdash \text{val a: P; val b: MP; a = new AP(1981); b = new MP; a.call(b.getNumber): void}}$$

and develop the subparts separately:

$$\dfrac{(a, P) \in \Gamma_1 \qquad \dfrac{\dfrac{(AP, Int \to AP) \in \Gamma_1 \quad \Gamma_1 \vdash \text{1981: Int}}{\Gamma_1 \vdash \text{new AP(1981): AP}} \quad AP <: P}{\Gamma_1 \vdash \text{new AP(1981): P}}}{\Gamma_1 \vdash \text{a = new AP(1981): void}}$$

$$\dfrac{(b, MP) \in \Gamma_1 \qquad \dfrac{(MP, void \to MP) \in \Gamma_1}{\Gamma_1 \vdash \text{new MP: MP}}}{\Gamma_1 \vdash \text{b = new MP: void}}$$

$$\dfrac{\dfrac{(a, P) \in \Gamma_1}{\Gamma_1 \vdash \text{a: P}} \qquad (call, Int \to void) \in \Gamma_P \qquad \dfrac{\dfrac{(b, MP) \in \Gamma_1}{\Gamma_1 \vdash \text{b: MP}} \quad MP <: P}{\Gamma_1 \vdash \text{b: Phone}} \quad (getNumber, void \to Int) \in \Gamma_P}{\Gamma_1 \vdash \text{a.call(b.getNumber): void}}$$

b) [8 pts]
We will denote T^* to be either T^- or T^+ (but not both in the same type rule). We will use the type variables R, S, T.

For any constructor of a class T, $(T, void \to T^-) \in \Gamma$.

$$1 : \frac{(T, void \to T^-) \in \Gamma}{\Gamma \vdash new\ T : T^-} \qquad 2 : \frac{}{\Gamma \vdash null : T^+ \text{ (for any T)}}$$

3: $$\frac{(x, R^-) \in \Gamma \qquad (p, S^+) \in \Gamma \qquad (fnc, S^+ \to T^+) \in \Gamma_R}{\Gamma \vdash x.fnc(p) : T^+}$$

$$4 : \frac{\Gamma \oplus (x, T^*) \qquad \Gamma \vdash e : T^*}{\Gamma \vdash (\text{val x: T} = \text{e}) : \text{void}} \quad 5 : \frac{(x, T^*) \in \Gamma \qquad \Gamma \vdash e : T^*}{\Gamma \vdash (\text{x} = \text{e}) : \text{void}}$$

c) [7 pts]

In order to prevent dereferencing a null value, we have to enforce the following condition: each variable only holds the values that its type permits it to hold. In particular, we have to make sure that whenever we have a variable of type T^-, null cannot be assigned to it.

For the below argument, we assume that the code type checks according to our type rules.

Objects or `null` values are the most basic elements in our language. By the rules 1 and 2 we see that we enforce correct types: newly created objects have type T^- and `null` will always get a type T^+.

From this, we can create more complex expressions by using method calls (rule 3). The return type of a method is always T^+, so that we cannot assign a `null` value a non-nullable type.

Thus, we have shown that the type rules for expressions are sound.

Finally, these expressions can be assigned to a variable. If the variable is declared at the same time (rule 4), it gets the same null-annotated type as the expression, hence we will not store a `null` value in a non-nullable type by the correctness of typing of our expressions.

If we have an assignment without declaration (rule 5), the variable must have the same null-annotated type as the expression, so by the same reasoning no `null` value can be stored in a non-nullable type.

Note that our rules allow assigning $e : T^-$ to $x : T^+$ by subtyping, but not the other way around.

Hence, we have shown for all statements in our language that if they type check according to our rules, a variable with type T^- will never store a `null` value. Then, given our type rule for method calls (rule 3), this prevents null-dereferencing a null value as method calls are only allowed on non-nullable variables.

Problem 5: Code generation (15 points)

a) [5 pts]

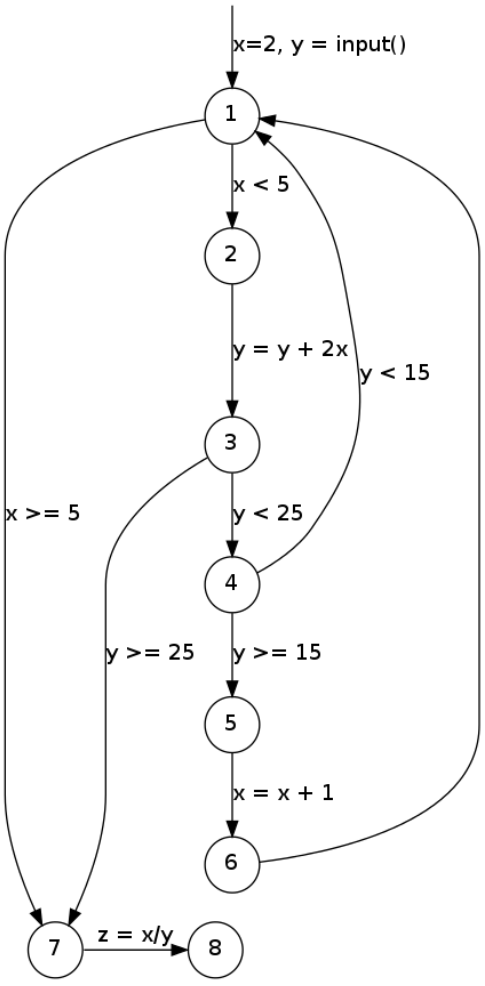
```
                ⟦do { body } until (c)⟧ =
nStart:         ⟦ body ⟧
                ⟦c⟧
                 if_eq nStart
nAfter:
```

b) [10 pts]

```
                ⟦ initializations⟧
nStart:         ⟦ swapped ⟧
                 if_eq nAfter
                ⟦ swapped = false; j += 1; var i = 0; ⟧
fStart:         ⟦ i  <  arr.length - j ⟧
                 if_eq fAfter
                ⟦ arr(i)  >  arr(i+1) ⟧
                 if_eq iAfter
                ⟦ swap(...); swapped = true; ⟧
iAfter:         ⟦i = i + 1;⟧
                 goto fStart
fAfter:          goto nStart
nAfter:
```

Problem 6: Data flow analysis (20 points)

a) [5 pts]



b) [13 pts]

Running interval analysis until convergence leads to the following ranges at each CFG node:

node in CFG	x	y	z
1	[2, 5]	[-5, 24]	$\bot$
2	[2, 4]	[-5, 24]	$\bot$
3	[2, 4]	[-1, 32]	$\bot$
4	[2, 4]	[-1, 24]	$\bot$
5	[2, 4]	[15, 24]	$\bot$
6	[3, 5]	[15, 24]	$\bot$
7	[2, 5]	[-5, 32]	$\bot$
8	[2, 5]	[-5, 32]	$\top$

c) [2 pts]

At line 15, according to the analysis, x can have values $[2, 5]$ and y can have values $[-5, 32]$. Thus, the analysis would conclude that a divide-by-zero is possible at line 15.