Solutions

1. True/False [20 pts]

(parts a–j; -1 point for each wrong answer, 0 points for each blank answer, 2 point for each correct answer. Therefore, the score for this problem is max(0, 2·#correct − #incorrect).

(a) Every LL(1) grammar is also LR(1).

True

(b) Every LR(1) grammar is unambiguous.

True

(c) The following grammar is LL(1):

\[
\begin{align*}
A & \rightarrow BC \\
& \mid CB \\
B & \rightarrow aBb \\
& \mid x \\
C & \rightarrow aCb \\
& \mid y
\end{align*}
\]

False

(d) LL parsers are often implemented with a bottom-up recursive-descent parser.

False

(e) Reference counting garbage collection cannot collect cycles of garbage.

True

(f) A mark–sweep garbage collector takes time proportional to the number of objects allocated in the heap.

True

(g) Generational garbage collection is based on the assumption that older objects are more likely to be garbage.

False
(h) A call to a method inherited from more than one superclass is always ambiguous.

*False*

(i) In the worst case, translation of an NFA to a DFA can result in an automaton with twice as many states.

*False*

(j) Local variables are allocated on the call stack or in registers.

*True*

2. **Scanning [12 pts]**

In a simplified XML, there are three kinds of tags:

- a start tag, consisting of a `<`, an identifier, and a `>` (e.g., `<p>`),
- an end tag, consisting of a `<`, `/`, an identifier, and a `>` (e.g., `</p>`), and
- a start–end tag, consisting of a `<`, an identifier, a `/`, and a `>` (e.g., `<br/>`).

For our purposes, an identifier is a non-empty sequence of letters. The regular expression for a letter is `[A-Za-z]`.

(a) [4 pts] Write a single regular expression that matches all three kinds of tags. Feel free to define abbreviations as long as the language remains regular.

*Answer:*

```
< ( [A-Za-z]+ /? | / [A-Za-z]+ ) >
```

(b) [6 pts] Draw, as a “circle-and-arrows” diagram, an NFA for the regular expression you wrote in (a). Be sure to indicate the start state and any final, accepting states. Use a double circle to indicate an accepting state.

*Answer:*
(c) [2 pts] Is the NFA you drew in (b) deterministic? Explain why or why not.

*Answer:* Yes. Every node has a single out-edge for each symbol. There are no \( \varepsilon \)-transitions.

3. **Parsing** [20 pts]

Consider the following LL(1) grammar for a subset of Lisp:

\[
\begin{align*}
Pgm & \rightarrow \ Exp \ \$ \\
Exp & \rightarrow \ atom \\
& \mid \ ' \ Exp \\
& \mid \ ( \ Exp \ Exps \ ) \\
Exps & \rightarrow \ Exp \ Exps \\
& \mid \ \varepsilon
\end{align*}
\]

$ is used to indicate end-of-file. \( \varepsilon \) indicates the empty string. The goal symbol is \( Pgm \).

(a) [6 pts] Give a parse tree for \( \text{(cdr ' (a b c)) $} \).

*Answer:*

\[
(Pgm \ (Exp \\
\quad (Exps \ (Exp "cdr") \\
\quad \ (Exps \ (Exp "\" \\
\quad \quad (Exp "(" \\
\quad \quad \ (Exp "a") \\
\quad \quad \ (Exps \ (Exp "b") \\
\quad \quad \quad (Exps \ (Exp "c") \\
\quad \quad \quad \ (Exps))) \\
\quad \quad "\)) \\
\quad \ (Exps))) \\
"$))
\]

(b) [6 pts] Consider a recursive descent parser running on the same input as in (b). When the ' is matched, what functions of the parser (including the current function) are active on the parser’s call stack?

*Answer:*

```
Pgm
Exp
Exps
Exp
```
(c) [6 pts] In an OO programming language of your choice, sketch how you would implement AST classes for this subset of Lisp. Include the superclass and fields of each class, if any; do not bother with methods or constructors.

Answer:

```java
class Pgm { Exp e; }
class Exp { }
class Atom extends Exp { String name; }
class Quote extends Exp { Exp e; }
class Apply extends Exp { Exp e; List<Exp> es; }
```

4. Type-checking, symbol tables [30 pts]

In this problem, we will explore how to extend a MiniJava type-checker similar to the one use in the project to support abstract classes and methods.

An abstract method cannot have a body. A class with an abstract method must also be declared abstract. A class that inherits an abstract method and does not override it must also be abstract.

It is illegal to create (with new) an instance of an abstract class. An abstract method or class is declared with the abstract keyword. The class and method declaration AST classes (ClassDeclaration, ClassExtendsDeclaration, and MethodDeclaration) are extended with a field that indicates whether the class or method is abstract. Since an abstract method cannot have a body, this is indicated in the AST by making the appropriate fields null.

Given an implementation of a compiler similar to the one used in the project, describe the changes you would make to the compiler to support abstract classes and methods. You need not write code or give specific names for classes, fields, etc., but be specific and be as complete as possible. Do not assume we know the details of the compiler you are extending beyond the basic structure that was outlined in class and in the project description—if a detail or an assumption is important, be sure to state it explicitly.

In particular:

(a) [10 pts] Describe what changes, if any, must be made to symbol table data structures (including the class table, type descriptors, typing environment, etc.).

Answer: Add a boolean flag isAbstract to the class table entry for each class. Add a boolean flag isAbstract to the method entry in the class table.

(b) [10 pts] Describe what changes, if any, must be made to the class-table-building pass.

Answer: At each class declaration, set the isAbstract flag in the class table entry if the class is declared abstract.

At each method declaration, set the isAbstract flag in the method entry if the method is declared abstract.
(c) [10 pts] Describe what changes, if any, must be made to the type-checking pass.

Answer: At each class declaration, if the class is not declared abstract, iterate through all the methods of the class. If a method is declared abstract, report an error. We can define the set of methods we need to check recursively. If \( C \) extends \( B \), then \( \text{methods}(C) \) includes methods declared in \( C \) plus \( \text{methods}(B) \) minus methods declared in \( C \).

At each method declaration, check \( \text{isAbstract} \) in the class table is set iff the method has a body. This could be done in the previous pass.

At each new expression, lookup in the class table to check if the class being instantiated is abstract.

5. IR [20 pts]

In this problem, we’ll consider a stack-based IR similar to Java bytecode. Each instruction pops zero or values from the operand stack, performs an operation, and then pushes the result (if any) onto the stack. Note that in this IR, the only values are integers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push ( n )</td>
<td>push the integer constant ( n )</td>
</tr>
<tr>
<td>push ( x )</td>
<td>push the value stored in local variable ( x )</td>
</tr>
<tr>
<td>pop</td>
<td>pop a value from the stack</td>
</tr>
<tr>
<td>store ( x )</td>
<td>pop the top-of-stack and store the value into local variable ( x )</td>
</tr>
<tr>
<td>add</td>
<td>pop two operands from the stack and push their sum</td>
</tr>
<tr>
<td>sub</td>
<td>pop two operands from the stack and push their difference</td>
</tr>
<tr>
<td>goto ( L )</td>
<td>jump to label ( L )</td>
</tr>
<tr>
<td>label ( L )</td>
<td>place label ( L ) in the instruction stream</td>
</tr>
<tr>
<td>ifeqz ( L )</td>
<td>pop one value from the stack and jump to ( L ) if ( = 0 )</td>
</tr>
<tr>
<td>ifltz ( L )</td>
<td>pop one value from the stack and jump to ( L ) if ( &lt; 0 )</td>
</tr>
<tr>
<td>ifgtz ( L )</td>
<td>pop one value from the stack and jump to ( L ) if ( &gt; 0 )</td>
</tr>
</tbody>
</table>

(a) [15 pts] Write the translation of the following constructs into the above IR:

i. [3 pts]
\[ x = y; \]
Answer:
push \( y \)
store \( x \)

ii. [3 pts]
\[ a - (b + 1) \]
Answer:
push \( a \)
push \( b \)
push \( 1 \)
add
sub
iii. [4 pts]
\[
i = 0;
while (true)
    i = i + 1;
\]

*Answer:*
push 0
store i
label L
push i
push 1
add
store i
goto L

iv. [5 pts]
\[
i = 1;
while (i < n)
    i = i + i;
\]

*Answer:*
push 1
store i
label L
push i
push i
add
store i
push i
push n
sub
ifltz L

(b) [5 pts] One constraint on this IR is that the operand stack must be exactly the same height for every jump to a given label L. Why is this constraint needed? What could happen without it?

*Answer: *The problem occurs if there are multiple jumps to the same label where the heights differ. The code at the label might pop more elements from the stack than are present. Another problem is that if the stack at the end of a loop body is higher than the stack before the loop body is executed, the stack may grow without bound.