CS 550
Take-Home Quiz
Due Monday, November 9, at the beginning of class

Name: Atanu Roy

Part I: Formal Definitions of Programming Language Syntax

Syntax

There are three recognized ways of formally defining the syntax of a programming language that are in general use. In order of most common to least common, these are:

1. Extended Backus-Naur Form (EBNF)
2. Context-free grammars
3. Syntax diagrams

All of these are equivalent in terms of their expressive power.

Tokens

A subset of the syntax specification of any programming language is the specification of the tokens of the programming language. Tokens can (and are) formally defined with:

1. Regular expressions
2. EBNF
3. Syntax diagrams (google "syntax diagrams for pascal," for example)

Note: You may use any source you like in answering these questions. In each instance where you use another source, cite that source. You will also find help on this site:

http://www.cs.montana.edu/ross/classes/spring2009/cs450/pages/sessions.html

A. Diagram the different modules of a complete compiler that generates target native machine code for some processor and operating system, describing the input to each module and the output of each module.

Please refer to the next (opposite) page.

x. What is the purpose and function of the front end of a compiler?

A compiler frontend will accept the source code, analyze it and build an intermediate representation (IR) of the code. This IR will be the input to a virtual machine (compiler back end) which, in turn, converts it into a machine-level code. The frontend tokenizes, parses & creates abstract syntax tree for a code. Moreover, it creates & manages the symbol table. It also includes error handling at each phase. Moreover, the frontend is able to do some optimization of
Different modules of a compiler.

```
Source line of code

Lexical Analyzer

Tokens

Preprocessor (might be present)

Syntax Analysis

Parse Tree

Semantic Analyzer

Abstract Syntax Tree

Intermediate Code Generator

Intermediate code - optimizer

Optimized Intermediate Code

Target Code Generator

Machine code file

Source

[Aho, Sethi, Ullman +

Dr. Ross' class notes]
3. What is the purpose and function of the back end of a compiler?

The back end includes those portions of the compiler that depend on the target machine. The back end can be synonyously be termed as the virtual machine. It is a program that takes the intermediate representation from the front-end, optimizes it, and generates the machine-readable code of the target machine in which it is running.

4. Why is scanning separated from parsing in a compiler?

Scanning is separated from parsing due to the following reasons: a) Lexical rules of a language are quite simple & we don't need a powerful tool like grammar. b) RE generally provide a more concise & easier-to-understand token than grammar. c) Repeating the syntactic structure of a language into lexical & non-lexical part provides a convenient way of modularizing the front-end of a compiler into manageable sized components.

5. What are the two most common methods for parsing? Describe each briefly and in general terms.

The two most common methods of parsing are a) top-down & bottom-up parsing.

Top-Down Parsing: This is a type of parsing in which the parse tree for an input string is created from the root of the tree & creating the nodes is preorder. In this type of parsing, we replace the left side of a grammar rule with its right side.

Bottom-Up Parsing: In this type, we try to parse an input string starting at the leaves (the terminals) & going up the parse tree towards the root. On this type of parsing, we replace the right side of a grammar rule by its left side.

Suppose that an identifier is described in English this way:

- Must start with a letter
- Thereafter may consist of letters, digits, and the underscore character
- May not have two underscore characters in sequence
- May not end with an underscore

6. Give a regular expression definition for identifier

```
letter \rightarrow \text{A} | \text{B} | \ldots | \text{Z} | \text{a} | \text{b} | \ldots | \text{z}
digit \rightarrow 0 | 1 | 2 | \ldots | 9
id \rightarrow \text{letter} \ (\text{letter} \ | \text{digit} \ | \text{[ -][ -]} \ (\text{letter} \ | \text{digit})^*)
```

7. Give an EBNF definition of identifier

```
letter := \text{A} | \text{B} | \ldots | \text{Z} | \text{a} | \text{b} | \ldots | \text{z}
digit := 0 | 1 | 2 | \ldots | 9
id := \text{letter} \ (\text{letter} \ | \text{digit} \ | \text{[ -][ -]} \ (\text{letter} \ | \text{digit})^*)
```

8. Give either a finite state automaton or syntax diagram definition of identifier:

http://qarshol.priv.no/ download /text/bnf.hwi

- Finite State Automaton:

```
[A-Z,a-z]  [0-9]
```

- Syntax Diagram: [Diagram Image]
9. Suppose that an if statement can be of the forms:

```
if condition then
  statements
end if
```

or

```
if condition then
  statements
else
  statements
end if
```

a. Give one EBNF rule that encompasses both versions of this if statement. Do not further define condition or statements.

```
Stmt := "if" condition "then" Stmt "endif"
| "if" condition "then" Stmt "else" Stmt "endif"
```

b. Give context free grammar rules that define the if statement. Do not further define condition or statements. Do not worry about making these rules conform to either LL or LR grammars.

```
Stmt → if condition then Stmt endif
| if condition then Stmt else Stmt endif
```

10. After looking at the programming language of your project or others' projects, how would you describe the software process involved in interpreting a program?

When a source code is fed to the front-end of a compiler, a lexer tokenizes the code. It converts the tokens to parsers which in turn creates a parse tree of the tokenized code. Now the semantic analyzer takes the parse tree and adds semantic information to it. The output of this phase is typically a non-optimized intermediate code. For some languages (like Java) the precompiled is stored and can be executed using a program called a virtual machine. But some languages (like Python, Ruby) immediately executes the intermediate code.

There is a program called the virtual machine, which runs on the target machine. This program takes as input the intermediate representation and outputs the target machine language code. Moreover a few languages has implemented the JIT compiling. It searches for the hotspots inside the code and re-compiles on the fly.
JIT compiler will convert the bytecode directly into native machine code. The performance improvement over interpreters originate from caching the results of translating blocks of code, not simply re-evaluating each line or operand each time it is met. Languages like Java have implemented JIT but Mozilla's new JavaScript engine has incorporated JIT compiling recently. Languages like Groovy does not have a VM to run on. It uses the JVM.

[Source: Dr. Ross' class discussion]
11. We have made the statement that no LL(1) grammar can be ambiguous. Provide clear, intuitive reasoning that could be used as the basis for a formal proof that this statement is true.

Let us have a grammar like 0) $S \rightarrow E\$  1) $D \rightarrow E+E$  2) $E \rightarrow E+E$  3) $E \rightarrow \text{id}$

So, if we have to parse a string at $b+c$, there can be more than a parse tree, proving the grammar to be ambiguous. Now the parse table will be

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>12</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$$</td>
<td></td>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>
```

Since there is a conflict in the parse table, thus without back-tracking, it would be virtually impossible to parse a string of this grammar. Since LL parsing doesn't support back-tracking, thus it proves the validity of the above statement. But if it was possible to look-ahead by $3$ tokens, this grammar may have been parsed.

[Concept of LL(3) - taken from Wikipedia]

12. We have made the statement that no LL(1) grammar can be left recursive. Provide clear, intuitive reasoning that could be used as the basis for a formal proof that this statement is true.

Assume we are using the following left-recursive grammar to parse the string $a+b$.

0) $S \rightarrow E\$  1) $E \rightarrow E+T$  2) $T \rightarrow a$  3) $T \rightarrow b$

So, a snippet of parse table for this grammar will be like

```
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
```

No, when the parser is at the non-terminal $E$ and the lookahead is an “identifier” ($a$), it goes to the part containing two rules.

Thus it may choose any one of them. Suppose it chooses rule 2. So it cannot parse the string, while choosing 1 would have allowed it to parse the string. Thus, when it chooses 2 and reaches a dead-end, it can backtrack & choose 1 to parse the string. But back-tracking is not permitted in LL parsing. Thus we may conclude saying that since left recursion causes conflict & there is no way for the compiler...
Ano12 Continuation.

to intuitively choose the right rule which will end up parsing the string. The only way is to choose a rule & if it fails to parse the string (like the example provided) it has to backtrack & since back-tracking is not allowed in LL(1) grammar, thus the validity of the statement is proved.