COS 301 Programming Languages
Fall 2012
Project Assignment #2 Due Tuesday Oct 16
Syntax, Operators, Scope and Primitive Data Types

Write an 8-to-12 page paper that addresses the following topics for your language:

- summary of basic syntactic structure (including statement terminators or separators; block structure; syntactic peculiarities, etc.)
- Discuss the units or levels of scope and the nature and type (run-time or compile-time) of name bindings within the different levels of scope.
- primitive data types available, including range limitations or lack thereof
- operators for primitive data types and their precedence and associativity

"Primitive" data types are data types that are not defined in terms of other data types. For many languages this will include strings; however (for example) in C it does not because C defines strings as arrays of characters.

Note that the goal is to summarize (and highlight the interesting points), not to provide reference manuals. For example you should assume that your audience knows what logical operator is and they do not need a description of standard logical operators such as AND, OR, NOT. If however your language supports XNOR or distinguishes strict equality (===) from equality (==) it would be worthy of mention. Some languages may have dozens of operators. You may want to include lists of operators, data types, etc. in appendices.

The difficult part of writing this paper is to make it short. Remember your audience and do not try to write a reference manual. For example, when discussing integer ranges, you could simply mention that they are limited by the target machine architecture or that they are unlimited in principle.

Continue developing the annotated bibliography of your references. As discussed in class, your bibliography may include references not cited in this particular paper if they were used in earlier papers.

Programming Assignment #2

Your assignment is to write a lexer and a recursive descent parser for a small grammar. The recursive descent parser will simply be a recognizer that determines whether or not a string is in the language; it will not construct an actual parse tree. Note that the calling sequence corresponds exactly to the parse tree. Output should be similar to the output from the example parser in the text (see next page)

The grammar is a grammar of Boolean expressions. The meta-character | is not the same as the OR operator **|**.

```
<bool_expr> ::= <and_term> { | <and_term> }
<and_term> ::=  <bool_factor> { &  <bool_factor> }
<bool_factor> ::=  <bool_literal> | !<bool_factor> |
                   ( <bool_expr> ) |  <relation_expr>
<relation_expr> ::=  <id> { relop <id> }
<id> ::=  letter { letter | digit }
<bool_literal> ::= true | false
<relop> ::=  = | <> | > | < | >= | <=
```

Constructing the lexer is actually the more difficult part of this assignment. You can always construct the parser first by creating an input stream of tokens (e.g., 1 = Left Paren, 2 = Right Paren, etc); then create a lexer that simply returns the next integer token. Then worry about actually parsing characters in a "program" later.

Include output for the following expressions in an appendix:

```
foo & !( a2 > bar & w < foo | x < y)
A1 & B1 | A2 & B1 | (! C | A <> B )
```

Example output from the Sebesta parser:

```
Next token is: 25 lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 lexeme is 47
Enter <term>
Enter <factor>
```