CS245 Lab 2: Parsing (some) Scheme Expressions

Due any time Friday of the 8th week of classes (28 Oct 2011); handed out 7th Wednesday.

In this lab, you will begin work on a translator that converts certain Scheme expressions into equivalent C++ expressions. Your grammar and parser must handle uses of the operators +, -, *, and /, as well as literal integers (like 42), certain uses of let, and calls to a function named input that I’ll provide. Parentheses will be present according to the usual usage in Scheme, so legal inputs could include various things like “42” or “(+ 5 2)” or “(+ 5 2 7)” or “(+ 5 2 (+ (* 3 2) 1))” (these could be translated into “42”, “5 + 2”, “5 + 2 + 7”, and “5 + 2 + ((3 * 2) + 1)”, respectively).

Your program should translate only one expression each time it is run. Arithmetic operations should allow two or more operands (e.g., “(+ 5 2 91 7)”) but you may limit the uses of let to only those that define one or two variables, rather than handling arbitrarily long lists of variables.

Your output must include enough parentheses to ensure the translated C++ expression will give the same answer as the original Scheme expression. You may, if you wish, remove unnecessary parentheses, but you are not required to do so. For example, you may translate the input “(+ 2 ( * 6 7))” into the result “2 + 6 * 7” or the result “((2) + ((6) * (7)))”.

Uses of the input function should be translated into calls to a C++ function input, which takes a string and requests that the user enter a (numeric) value for that variable, and returns that value. Thus, “(+ 32 (/ (* 9 (input "enter Celsius temperature"))) 5))” could be translated into ‘32 + 9 * input("enter Celsius temperature") / 5”.

The translation of let expressions will be covered later in this course — for now, your grammar and parser should accept them, but generate trivial code such as “/*** untranslated let expression -- wait for Lab 4 ***/ 0” for let expressions or uses of variables.

In this lab, you do not have to reject input that is illegal for reasons other than grammatical problems — for example, you do not need to produce error messages for undefined variables (such as a trivial example “(+ x 1)”) or type errors (such as “(input 12)”, “(+ 1 "enter y")”, or even “(let ((prompt "enter_y") (x 1)) (+ (input x) prompt))”; for now, just translate the first two into illegal C++ programs and translate the latter as you would any other let; in a later lab you will reject these but accept and translate legal uses of let including, for example, “(let ((prompt "enter_y") (x 1)) (+ x (input prompt)))”.

You do not need to build a lexical scanner; just use the one demonstrated in class and included with the starter files.

To get full credit for the translator labs, you must

- include, as a comment, a Context-Free Grammar for the language you parse,
- create parsing functions that correspond to the nonterminal symbols in that CFG, and
- have your parser build an Abstract Syntax Tree, and generate your result from the AST.

In this first translator lab, you should

1. obtain the files for the translator project using Eclipse’s CVS Repository Exploring perspective (or, if you prefer, command-line cvs),
2. create your complete context-free-grammar and parser (I recommend you start by building these, and just try to produce error messages for illegal programs by printing an error message to `cerr` and then calling the standard library function `exit`; also, I recommend that you think carefully, and express your thoughts in clear comments, regarding the question of whether a each grammar function expects that its first token will be the current token or the one after the current token),

3. set up an AST class — this may be completely new, something based on your trees from Lab 1, a C++ translation of something you wrote in CS106, or something from a standard library or the web, as long as you put in comments to make clear where you got it,

4. adjust your parser to build AST’s for legal expressions — since you don’t actually have to translate `let` expressions yet, you may choose to build their AST’s or leave them empty,

5. create, and call from your `main` function, a function to generate the appropriate C++ expression for AST’s, and finally

6. test your program (my tests will provide the `input` function and run the translated C++).

Team->Commit your work when you are done, and also (please) after each intermediate step.