Assignment 2: Java Translator Program
Due: Tuesday, 20 October 2015 at 11:59pm

Overview

The E programming language is similar to C, C++, Java, Pascal, and Modula-2, but it differs syntactically and semantically, and is considerably simpler.

You are to write a Java program that translates E programs to their semantically equivalent C programs (not C++). That is, the input to your program — henceforth, the *translator* is an E program; the output from your program is a C program — henceforth, the *generated code*, or simply *GC*. To verify that your translator works correctly, you are to compile and execute the GC. This assignment should help you develop a better understanding of parsing and compiling, and some familiarity and practical programming experience with Java.

The general approach to translation you will be using does all its work in one pass. That is, its three activities of syntactic analysis, semantic analysis, and code generation are intermixed. As your translator reads a token or a small group of tokens in the E program, it will determine if the token(s) is syntactically and semantically valid, and generate code for the token(s). The E language is designed to facilitate one-pass translation. In contrast, many real compilers perform multiple passes, *e.g.*, one pass for each of the three activities.

The E Language

The following grammar partially describes the E language. In the BNF below, nonterminals are in lowercase; terminal symbols are enclosed in single quotes to avoid potential confusion in the use of characters that are both meta-symbols in the grammar and terminals in E, *e.g.*, "[".

```
program ::= block
block ::= declaration_list statement_list
declaration_list ::= {declaration}
statement_list ::= {statement}

declaration ::= '@' id { ',' id }

statement ::= assignment | print | do | if
print ::= '!' expr
assignment ::= ref_id '=' expr
ref_id ::= [ '~' [ number ] ] id
do ::= '<' guarded_command '>'
if ::= '[' guarded_command { '|' guarded_command } [ '%' block ] ']'
guarded_command ::= expr ':' block

expr ::= term { addop term }
term ::= factor { multop factor }
```

```
factor ::= '(' expr ')' | ref_id | number
addop ::= '+' | '-'
multop ::= '*' | '/'
```

The following rules complete the syntactic definition of E:

- An E program is followed by an end-of-file indicator; extra text is not legal.
- The nonterminal `id` represents a nonempty sequence of letters. Similarly, the nonterminal `number` represents a non-empty sequence of digits.
- Special characters (*e.g.*, `'['` and `'!'`) serve the role of keywords. Hence there are no reserved words.
- As is the case in most languages, tokens are formed by taking the longest possible sequences of constituent characters. For example, the input "abcd" represents a single identifier, not several identifiers. Whitespace consists of one or more blanks, tabs, or newlines.

  For example, "x:10" and "x : 10" are equivalent. Note that whitespace delimits tokens; *e.g.*, "abc" is one token whereas "a bc" is two.
- A comment in E begins with a "#" and consists of all characters up to, but not including, a newline or end-of-file.

E's semantics follow C's semantics for the most part, but there are significant differences. The important semantic points are:

- E has only one type of variable: integer; as with local variables in C, the initial value of E variables is not defined.
- As in C, variables must be declared before use, redeclaration of a variable (in the same block) is an error, and the declaration of a variable in an inner block hides the declaration of variable(s) of the same name declared in outer blocks. Thus, this part of E's scoping rules are just like C's. Unlike C, though, E provides the scoping operator `'~'`. Its meaning is as follows:

  ~0 x: the variable x that is declared in the current block.

  ~1 x: the variable x that is declared in the immediately enclosing block (*i.e.*, 1 level up).

  ~2 x: the variable x that is declared in the enclosing block 2 levels up

  ...: etc.

  ~x: the variable x that is declared at the global level.

  ~N x is an error if N is greater than the current depth of nesting. Note that ~x acts like C++'s scope resolution operator, *i.e.*, ::x.
- An expression in a guard—*i.e.*, as the expr before the `':'` in a guarded_command—is considered true if non-positive and false if positive. For example,

```
i = 1
<i-10: !i i=i+1>
```

  is a loop that prints out the first 10 positive integers. (The value of `i` is 11 after the loop.)

- As in most languages, the if statement tests its guards in lexical order, until it finds one that is true; the corresponding block is then executed. If no guard is found true, the else block (if any) is executed. If no else is present, then the if has no effect.

To guide you in developing your program, this assignment is divided into several parts.

Part 1: The Scanner

A scanner is sometimes called a lexer — since it does lexical analysis — or a tokenizer — since it breaks up its input into tokens. For example, the *GetToken* procedure described in Louden is a scanner.

You will be provided with a nearly complete scanner. You need to provide a few omitted tokens (in Scan.java and TK.java). This part is simple, but look at the scanner code to see how it works and use the provided test scripts. In fact, use the provided test scripts before you make any changes; from their output, it should be pretty clear what tokens are missing.

For simplicity, the scanner imposes a limit on the length of tokens. If the scanner encounters a token whose length exceeds that limit, it prints an appropriate error message and ignores the extra characters. Also, if the scanner encounters a character that is not in E's character set, it prints an appropriate error message, ignores the unknown character [1], and continues by examining the next character in the input. Finally, the scanner simply discards whitespace and comments.

Part 2: The Parser

First, rewrite the grammar using syntax graphs. Then, determine the first sets; this should be straightforward because the grammar is simple. Finally, translate the syntax graph into a parser. See the textbook for details. If an error, *e.g.*, missing "=" in an assignment statement — is encountered in parsing, give an appropriate error message via "System.err.println" and then stop your program via "System.exit(1)".

Test your parser to see that it recognizes syntactically legal E programs and complains about syntactically illegal programs.

You will be provided with a small part of the parser and the main program. You will need to copy from the previous part the Java code files (but not e2c.java) you used for your scanner.

Part 3: The Symbol Table

Up to this point, we have dealt with syntax. Now, we need to enforce the semantic constraints that deals with variables: We must detect undeclared and redeclared variables.

A table of variables in the current scope needs to be maintained. Each time a variable is declared, the table is checked. If the variable is already in the table entries for the current block, then it is being redeclared. Otherwise, the variable is added to the table. Each time a variable is referenced, the table is checked to ensure that the variable has been declared.

Here is one implementation approach. Use a stack of lists of variable names. When a new block is entered, a new list of variable names is created to hold the variables in this new block. Each (legal) variable declaration adds to the list for the current block. When a block is exited, the list for that block is popped from the stack.

[1] More precisely, such a character is treated as a whitespace as it delimits tokens.

An undeclared variable, then, is one that does not appear anywhere in the entire stack of lists. (Thus, the stack is used in an "impure" sense. It provides more than pop and push operations; it also provides search.) In a real compiler, the symbol table would be searched beginning with the newest block to locate the most recent symbol table entry with the given name; the symbol table entry would contain additional information such as the variable's type. Given that E's scoping rules are like C's, then variables should generally be located in that fashion too. However, E (unlike C) provides the scoping operator '~', whose use will affect how you represent levels and how you search in the symbol table. The numeric argument to '~' specifies the exact scoping level in which to find the given variable; it is an error if the variable is not found within that level, even if the variable is currently in scope (*i.e.*, appears at a different level in the symbol table).

For a redeclared variable, give an appropriate error message, and then continue by ignoring the redeclaration. For an undeclared variable (including a bad '~' reference), give an appropriate error message via "System.err.println" and then stop your program via "System.exit(1)".

Since the number of variables in the program is neither known in advance nor bounded, your program must use a dynamically allocated data structure for the symbol table. Break this part into subparts. First, get everything working without handling the scoping operator. Then, handle the scoping operator.

Part 4: Generating Code

Modify your parser so that it outputs appropriate C code. For example, for the E program:

```
@i
i =1
<i-10 : ! i        #this is a comment
          i=i+1>
```

your translator might produce something like:

```
main(){
int x_i;
x_i = 1;
while( (x_i-10) <= 0 ){
printf("%d\n",x_i);
x_i = x_i+1;
}
}
```

Note that since the scanner discards whitespace and comments, the output is not as neatly formatted as the E program. In fact, the actual output from your translator will probably be less formatted than shown above; *e.g.*, it is fine (and simpler) to output a single C token per line. Also note that the translator has prepended each E variable name with x_ to avoid conflicts with C reserved words. (Hint: You might want to further "munge" identifiers for implementing levels for the '~' operator.)

To test your translator, examine its C output (the GC). Then, compile the GC and execute the resulting program. Verify that it does indeed execute as the source E program specifies.

Part 5: E Language Changes: Design and Implementation

Add a definite iteration statement (*e.g.*, like `for` in C or C++) to the language definition and your translator. This part has no one "correct" solution. Do, however, strive to be syntactically and semantically consistent with the other constructs in E. Give, in your README file:

- the new and modified BNF rules.
- a concise description as to why you chose the particular form you did.

To demonstrate that your implementation of the definite iteration is correct, include several well-commented E test program files and their "correct" output files. These tests should include tests that demonstrate error checking.

Notes

- On the CSIF systems, be sure to use

  ```
  /usr/java/jdk1.8.0_60/bin/javac
  /usr/java/jdk1.8.0_60/bin/java
  ```

  To do so for csh (or tcsh) users, *carefully* put

  ```
  set path = (/usr/java/jdk1.8.0_60/bin $path)
  ```

  in your .cshrc file. (You need to logout and login again to have this change take effect.) Test that you are using the correct Java by

  ```
  which javac java
  ```

  For users of other shells, make an analogous changes to the PATH environment variable.

- Do not be overly concerned with efficiency. On the other hand, do not write a grossly inefficient program.
- Strive for simplicity in your programming. Do not try to be fancy. For example, the syntax graphs translate directly into code. Learn and use that technique—do not spend your time trying to improve that code. You will find it most helpful to use the names of the nonterminals as the names of your parsing methods (except "if" and "do" are keywords in Java).
- Express your program neatly. Part of your grade will be based on presentation. Indentation and comments are important. Be sure to define each variable used with a concise comment describing its purpose. In general, each method should have a comment at its beginning describing its purpose. In this program, however, the parsing method do not need such introductory comments; a single explanatory comment at their beginning should suffice.

- Write your program so that it works on standard input. Do not bother trying to get it to work on a file whose name is specified on the command line. Thus, your program, named e2c, will be invoked as `java e2c` if you want input to come from the keyboard, or as `java e2c < t01.e` if you want input to come from the file t01.e; use `java e2c < t01.e > t01.c` to take input from t01.e and to output to t01.c. Output the generated code to stdout via "System.out.println" (or "System.out.print"). So as not to intermix generated code and error/warning messages, output error messages to stderr via "System.err.println".

- Your program must work with the provided Makefile and shell scripts.

- The provided shell scripts depend on the exit status of your program. If your program detects a "serious" error, your program should exit via "System.exit(1)", as described earlier. If your program finishes normally, your main method should just end normally or equivalently "System.exit(0)".

- Be sure to use the provided test files and test scripts. Your output must match the "correct" output, except for the wording of and the level of detail in the error messages. By "match", we mean match exactly character by character, including blanks, on each line; also, do not add or omit any blank lines. Your error messages can be more or less detailed as you wish, provided they are reasonably understandable. Don't spend a lot of time making fancy error messages, though.

- If you run your program on Windows, the output might look the same, but a file comparison program might complain due to extra \r's in Windows' output. Retest it on CSIF to be sure, and as is required.

- To convert String s to its integer equivalent and store the result in int num, use

```
num = Integer.parseInt(tok.string);
```

- You must follow the steps described above in developing your program. Grading will be divided as follows:

Part	Percentage
1	10
2	30
3	30
4	20
5	10

- In seeking assistance on this project, bring a listing of the last working part along with your attempt at the next part.

- You will be expected to turn in, at least, the last working part along with your attempt at the next part. So, be sure to save your work for each part. No credit will be given if the last working part is not turned in.

- Points will be deducted for not following instructions, such as the above.

- **Get started now to avoid the last minute rush.**