Assignment #2: Syntax Error Recovery

Summary:

The purpose of this assignment was to extend the language given to us to include “if”, and “while” statements. The from there, we had to implement Syntax error recovery for the program. The point of this is to parse the input program and to catch errors, but not to just halt after reporting the first error. You must continue on to catch as many mistakes as possible.

How I solved the problem:

In order to extend the language, I simply found the first set of “stmt” and added those to the function in my parser. I continued on by updating all of the first sets for the production rules, in order to achieve correctness for parsing my grammar.

For recovering from errors, I attached exception handlers to key production rules (program, stmt, cond, expr). From there if an incorrect token was found I simply followed the procedure below (from Programming Language Pragmatics by M. Scott).

```
loop
      if next_token ∈ FIRST(statement)
            statement()
            return;
      else if next_token ∈ FOLLOW(statement)
            return;
      else
            get_next_token();
```

When throwing an error I would include what was expected and at what token # it was expected at.
Example:

```
>> read read
>> Syntax Error: expected "id" at Token #: 2
```

Grammar (terminals in red):

P → SL $$

SL → S SL | ε

S → id := E | read id | write E | if C SL end | while C SL end
| # sLiteral

C → E ro E

E → T TT

T → F FT

F → (E) | id | literal

TT → ao T TT | ε

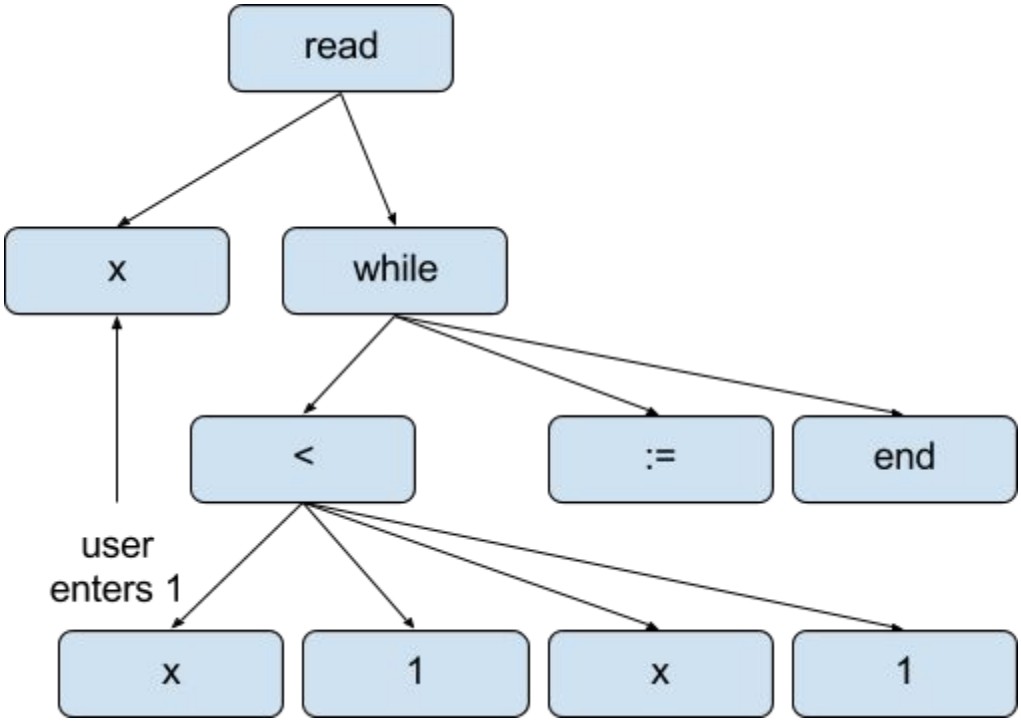
FT → mo F FT | ε

ro → == | != | < | > | <= | >=

ao → + | -

mo → * | /

Syntax Tree (Visualization):



The parse tree above is for the code:

```
>> read x
>> while x < 1
>>    x := 1
>> end
```

Everyone of my nodes in the tree has at most 4 children

- Child1: id, left hand side of expression, conditional
- Child2: right hand side of the expression
- Child3: end token
- Child4: next instruction

I chose to actually make the syntax tree in order to make interpreting the code easier. I did this with a node implementation I made. It holds the image of the token, the four possible children, and the token type used by the interpreter later.

How To Run:

1. Open terminal to directory of project
2. type “make” into the terminal
3. run program by typing “./parse < <textfile with code inside>

```
Example:

    Jean-Marcs-MacBook-Air:A2 JeanMarc$ make
    c++    -c -o parse.o parse.cpp
    g++ -o parse parse.o scan.o
    g++ node.hpp
    g++ interpreter.hpp
    Jean-Marcs-MacBook-Air:A2 JeanMarc$ ./parse < testCode.txt
```

Files Included:

- ➔ scan.h : header file for scanner
- ➔ scan.cpp : scanner file (breaks code into tokens)
- ➔ parse.cpp : parser file (parses incoming source code tokens)
- ➔ node.hpp : header file for Node implementation
- ➔ interpreter.hpp : header file for interpreting the syntax tree

Extra Credit:

- Interpreter
  - I implemented an interpreter to execute the code processed by my compiler and parser. When the parser is run it constructs a syntax tree from the given input. After parsing is finished, I pass on the root Node from the syntax tree to be interpreted. From there the tree is evaluated from left to right, executing each and every procedure, and evaluating expressions along the way.
  - HOW_TO_RUN_NTERPRETER:
    - Copy and paste code into terminal after executing program. Paste code without EOF token.
    - add “~” as the last token of the code, to let the parser know you would like to execute your program.

EXAMPLE:

```
>> ./parse
>> # "Enter_Number"
>> read x
>> while x < 5
>>   write x
>>   x := x + 1
>> end
>> ~
>>
>> ~~~~~~~ RUN PROGRAM HERE (INTERPRETER) ~~~~~~~
>> Enter_Number
>> 0
>> 0
>> 1
>> 2
>> 3
>> 4
```

- Do not pipe in programs with “~”, and EOF token. It will not work. When EOF is received, cin no longer works. (Because I had work with the scanner that was given to me. I would have implemented it differently to allow user input from cin after scanning and parsing). Do it as it says above.

- HOW I MADE IT:
    - I created an unordered map that holds all variables that have been declared inside the parsed code. All variables once they have been added to the map can be referenced from anywhere inside the program. In other words, all variables are Global Variables.
    - I use one function to decide how to proceed with the current procedure that's at the root of the tree. After that instruction is finished it returns it's next instruction to be called. Meanwhile the "eval" function evaluates any tree. Used for conditionals and numerical expressions.

- Extended Language
    - You can see above that I chose to add the ability to print out strings (Extended stmt production by adding "#"). In code you signify printing by typing "#" , then follow it by a string surrounded by double quotes. Example :

```
~ # "Print!"
~ Print!
```

    - You constraints of this function:
        - String may not contain spaces or quotes. You may use underscores.
        - String may not be more than 100 characters
        - Can only be used for Strings, and not other data types