Project (Part 2)

This project will explore the topics learned in class, and give you some simple hands-on experience. After completing all three parts, you should have a working interpreted language that is unique to you. You are encouraged to have fun and go above and beyond if you have the time.

In Part 2 , you will then need to make another small, but substantial change, and create the parser for the modified language. Part 2 assumes you have your lexer form part 1. The changes you make may affect your lexer. So, you may need to go back and update the lexer. If for whatever reason, you do not want to use your lexer from part 1, a lexer will be provided. It will be attached to assignment, and contain instructions for using it. You will be making other changes to your language as the semester goes forward. So, take that into consideration when writing your code.

All code should be written in either C,C++,Java, or Python.

1 What to do

- Once again modify your language. Use the knowledge in class to decided what change to make. Think about when things should be bound, and the design issues the book presents for the language feature you are considering. For instance, In chapter 6, the book gives the method for implementing arrays. You can use this information to decide the exact kind of arrays you want (assuming that is the change you wish to make).
- If the changes to your language causes the grammar to no longer be an LL grammar, then modify it.
- Implement the recursive decent parser in your chosen language. At minimum the parser should tell if a program is in the language or not. Preferably, you should print out nice error messages. This will help you with debugging too.

2 What to turn in

Upload your submission as a zip archive containing the following:

- Source code (c, c++, python, or java files)
  - Source code should not require a particular IDE to compile and run.
  - Should work on the cs1 and cs2 machines
- Readme (Plain text document)
  - List the files included in the archive and their purpose
  - Explain how to compile and run your project
  - Include any other notes that the TA may need

- Write-up (Microsoft Word or pdf format)
  - An description of your changes. This can be formal (modifying the grammar and semantics) or informal (A text description)
  - In addition, if you did not complete some feature of the project, why not?
    * What unsolvable problems did you encounter?
    * How did you try to solve the problems?
    * Where do you think the solution might lay?
      · What would you do to try and solve the problem if you had more time?

3 Grading

The grade for this project will be out of 100, and broken down as follows:

The change to the language	25
The parser code	65
Correct Output	10

If you were not able to complete some part of the program discussing the problem and potential solutions in the write-up will reduce the points deducted for it. For example, suppose there is a bug in your code that sometimes allows two customers to approach the same worker, and could not figure out the problem before the due date. You can write 2-3 paragraphs in the write-up to discuss this issue. Identify the error and discuss what you have done to try to fix it/find the problem point, and discuss how you would proceed if you had more time. Overall, inform me and the TA that you know the problem exists and you seriously spend time trying to fix the problem. Normally you may lose 5 points (since it is a rare error) but with the write-up you only lose 2. These points can make a large difference if the problem is affecting a larger portion of the program.

4 The Base Language

I provide a copy of the language from part 1 for convenience.

4.1 Syntax

(1) `<prog>` → `<stmt_list>`
(2) `<stmt_list>` → ϵ
(3) | `<stmt>` ";" `<stmt_list>`
(4) `<stmt>` → `<print>`
(5) | `<input>`
(6) | `<assign>`
(7) | `<if>`
(8) | `<while>`

(9) <print> → "print" <p-arg>
(10) <p-arg> → **STRING**
(11) | <expr>
(12) <input> → "get" **ID**
(13) <assign> → **ID** "=" <expr>
(14) <if> → "if" <expr> "then" <stmt_list> "else" <stmt_list> "end"
(15) <while> → "while" <expr> "do" <stmt_list> "end"
(16) <expr> → <n_expr> <b_expr>
(17) <b_expr> → ϵ
(18) | "and" <n_expr>
(19) | "or" <n_expr>
(20) <n_expr> → <term> <t_expr>
(21) <t_expr> → ϵ
(22) | "+" <n_expr>
(23) | "-" <n_expr>
(24) <term> → <factor> <f_expr>
(25) <f_expr> → ϵ
(26) | "*" <term>
(27) | "/" <term>
(28) | "%" <term>
(29) <factor> → <value> <v_expr>
(30) <v_expr> → ϵ
(31) | ">" <value>
(32) | ">=" <value>
(33) | "<" <value>
(34) | "<=" <value>
(35) | "==" <value>
(36) | "!=" <value>
(37) <value> → "(" <expr> ")"
(38) | "not" <value>
(39) | "-" <value>
(40) | **ID**
(41) | **INT**

4.1.1 Tokens

This subsection describes the token used in the above grammar. Provided for each token is a regex and a description. The regex is for those that know regular expressions and prefer it as a description. The description says the same thing in English. Preprocessing describes how the lexeme is transformed before passing it to the parser.

STRING

As a regex: "([^"]|\")*"

Description: A quotation mark followed by zero or more characters, where quotation marks must be preceded by a backslash, followed by another quotation mark.

Preprocessing: The first and last quotation marks are removed. Scanning from left to right, "\\" is replaced with "\", "\t" is replaced with a tab, "\n" is replaced with a newline, "\"" is replaced with """, and any "\" that is followed by anything else is removed.

ID

As regex: [_a-zA-Z][_a-zA-Z0-9]*

Description: A letter or underscore followed by a combination of zero or more letters, underscores or digits.

INT

As Regex: (+|-)?[0-9]+

Description: an optional "+" or "-" followed by one or more digits.

4.2 Static Semantics

1 <stmt_list>.*ids* = {}

3 <stmt_list>[0].*ids* = { <stmt>.*id*} ∪ <stmt_list>[0].*ids*
<stmt>.*ids* = <stmt_list>[0].*ids*

4 <print>.*ids* = <stmt>.*ids*

5 <stmt>.*id* = <input>.*id*

6 <stmt>.*id* = <assign>.*id*
<assign>.*ids* = <stmt>.*ids*

7 <if>.*ids* = <stmt>.*ids*

8 <while>.*ids* = <stmt>.*ids*

9 <p-arg>.*ids* = <print>.*ids*

11 <expr>.*ids* = <p-arg>.*ids*

12 <input>.*id* = **ID**.*id*

13 <assign>.*id* = **ID**.*id*

16 <n_expr>.*ids* = <expr>
<b_expr>.*ids* = <expr>

18 <n_expr> $.ids =$ <b_expr>

19 <n_expr> $.ids =$ <b_expr>

20 <term> $.ids =$ <b_expr>
<t_expr> $.ids =$ <b_expr>

22 <n_expr> $.ids =$ <t_expr> $.ids$

23 <n_expr> $.ids =$ <t_expr> $.ids$

24 <factor> $.ids =$ <term> $.ids$
<f_expr> $.ids =$ <term> $.ids$

26 <term> $.ids =$ <f_expr> $.ids$

27 <term> $.ids =$ <f_expr> $.ids$

28 <term> $.ids =$ <f_expr> $.ids$

29 <value> $.ids =$ <factor> $.ids$
<v_expr> $.ids =$ <factor> $.ids$

31 <value> $.ids =$ <v_expr> $.ids$

32 <value> $.ids =$ <v_expr> $.ids$

33 <value> $.ids =$ <v_expr> $.ids$

34 <value> $.ids =$ <v_expr> $.ids$

35 <value> $.ids =$ <v_expr> $.ids$

36 <value> $.ids =$ <v_expr> $.ids$

37 <expr> $.ids =$ <value> $.ids$

38 <value> $[1].ids =$ <value> $[0].ids$

39 <value> $[1].ids =$ <value> $[0].ids$

40 Predicate: **ID** $.id \in$ <value> $.ids$

4.3 Dynamic Semantics

This section gives the dynamic semantics of the language using denotational semantics. Consider the *demsem* function the denotational semantics for this language. We will use a mapping from variable name to value to represent the symbol table of the program during execution, and in code can be represented as a HashMap or similar datatype in your language of choice. We will use a sequence of characters to represent the output of a program, with ϵ representing the empty sequence. I will also assume that all strings will be represented as sequences of characters. Assume there is a function *append* that, when given two sequences, appends the second sequence to the first. Also assume, there is a function

seq that takes an integer and gives a sequence of characters representing that integer as text. Assume there are the functions *head*, which maps a sequence to its first element, *tail*, which maps a sequence to a new one created by removing the first element, *clean*, which maps a sequence of input characters to a new sequence by removing any non-digits from the front of the sequence, and *int* that maps a sequence of digits to the corresponding integer. If the sequence is empty, *int* will give zero. A state, as well as the meaning of a program, will be a 3-tuple consisting of a variable name mapping function, a sequence of input characters and an output sequence. The initial state for any program is $(\{\}, i, \epsilon)$, where i is some sequence of characters the user will input. If a token (represented by all caps and bold font) appears as a value on the right hand side of a function definition, then replace it with its lexeme. So if a **ID** was generated by the lexer from an x, then replace **ID** with x.

$$
\begin{aligned}
densem(\epsilon, (\theta, i, p)) &= (\theta, i, p) \\
densem(\texttt{<stmt>}\ \text{“;”}\ \texttt{<stmt\_list>}, (\theta, i, p)) &= densem(\texttt{<stmt\_list>}, densem(\texttt{<stmt>}, (\theta, i, p))) \\
densem(\text{“print”}\ \textbf{STRING}, (\theta, i, p)) &= (\theta, i, append(p, \textbf{STRING})) \\
densem(\text{“print”}\ \texttt{<expr>}, (\theta, i, p) &= (\theta, i, append(p, seq(out))) \\
&\quad \text{where } out = exprsem(\texttt{<expr>}) \\
densem(\text{“get”}\ \textbf{ID}, (\theta, i, p)) &= (\theta', i', p) \\
&\quad \text{where} \\
&\quad (x, i') = getInt(clean(i)) \\
&\quad \theta'(n) = \text{if } n = \textbf{ID} \text{ then } x \text{ else } \theta(n) \\
densem(\textbf{ID}\ \text{“=”}\ \texttt{<expr>}, (\theta, i, p)) &= (\theta', i, p) \\
&\quad \text{where} \\
&\quad \theta'(n) = \text{if } n = \textbf{ID} \text{ then } exprsem(\texttt{<expr>}, \theta) \text{ else } \theta(n) \\
densem(\texttt{<if>}, (\theta, i, p)) &= \text{if } exprsem(\texttt{<if>.<expr>}, \theta) \neq 0 \\
&\quad \text{then } densem(\texttt{<if>.<stmt\_list>}[0], (\theta, i, p)) \\
&\quad \text{else } densem(\texttt{<if>.<stmt\_list>}[1], (\theta, i, p)) \\
densem(\texttt{<while>}, (\theta, i, p)) &= \text{if } exprsem(\texttt{<while>.<expr>}, \theta) = 0 \\
&\quad \text{then } (\theta, i, p) \\
&\quad \text{else } densem(\texttt{<while>}, \\
&\qquad densem(\texttt{<while>.<stmt\_list>}, (\theta, i, p))) \\
exprsem(\texttt{<expr>}, \theta) &= \text{if } \texttt{<expr>.<b\_expr>} = \epsilon \\
&\quad \text{then } exprsem(\texttt{<expr>.<n\_expr>}, \theta) \\
&\quad \text{else } bexprsem(\texttt{<expr>.<b\_expr>}, \\
&\qquad exprsem(\texttt{<expr>.<n\_expr>}), \theta) \\
exprsem(\texttt{<n\_expr>}, \theta) &= \text{if } \texttt{<n\_expr>.<t\_expr>} = \epsilon \\
&\quad \text{then } exprsem(\texttt{<n\_expr>.<term>}, \theta) \\
&\quad \text{else } texprsem(\texttt{<n\_expr>.<t\_expr>}, \\
&\qquad exprsem(\texttt{<n\_expr>.<term>}), \theta)
\end{aligned}
$$

$$
\begin{aligned}
exprsem(\texttt{<term>}, \theta) &= \text{if } \texttt{<term>.<f\_expr>} = \epsilon \\
&\quad \text{then } exprsem(\texttt{<term>.<factor>}, \theta) \\
&\quad \text{else } fexprsem(\texttt{<term>.<f\_expr>}, \\
&\qquad exprsem(\texttt{<term>.<factor>}), \theta) \\
exprsem(\texttt{<factor>}, \theta) &= \text{if } \texttt{<factor>.<v\_expr>} = \epsilon \\
&\quad \text{then } exprsem(\texttt{<factor>.<value>}, \theta) \\
&\quad \text{else } vexprsem(\texttt{<factor>.<v\_expr>}, \\
&\qquad exprsem(\texttt{<factor>.<value>}), \theta) \\
exprsem(\text{“(”}\ \texttt{<expr>}\ \text{“)”}, \theta) &= exprsem(\texttt{<expr>}, \theta) \\
exprsem(\text{“not”}\ \texttt{<value>}, \theta) &= \text{if } exprsem(\texttt{<value>}, \theta) = 0 \text{ then 1 else 0} \\
exprsem(\text{“-”}\ \texttt{<value>}, \theta) &= -exprsem(\texttt{<value>}, \theta) \\
exprsem(\textbf{ID}, \theta) &= \theta(\textbf{ID}) \\
exprsem(\textbf{INT}, \theta) &= \textbf{INT} \\
bexprsem(\text{“and”}\ \texttt{<n\_expr>}, v, \theta) &= \text{if } v \neq 0 \text{ and } exprsem(\texttt{<n\_expr>}, \theta) \neq 0 \text{ then 1 else 0} \\
bexprsem(\text{“or”}\ \texttt{<n\_expr>}, v, \theta) &= \text{if } v \neq 0 \text{ or } exprsem(\texttt{<n\_expr>}, \theta) \neq 0 \text{ then 1 else 0} \\
texprsem(\text{“+”}\ \texttt{<n\_expr>}, v, \theta) &= v + exprsem(\texttt{<n\_expr>}, \theta) \\
texprsem(\text{“-”}\ \texttt{<n\_expr>}, v, \theta) &= v - exprsem(\texttt{<n\_expr>}, \theta) \\
fexprsem(\text{“*”}\ \texttt{<term>}, v, \theta) &= v \times exprsem(\texttt{<term>}, \theta) \\
fexprsem(\text{“/”}\ \texttt{<term>}, v, \theta) &= \frac{v}{exprsem(\texttt{<term>}, \theta)} \\
fexprsem(\text{“\%”}\ \texttt{<term>}, v, \theta) &= v \mod exprsem(\texttt{<term>}, \theta) \\
vexprsem(\text{“>”}\ \texttt{<value>}, v, \theta) &= \text{if } v > exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
vexprsem(\text{“>=”}\ \texttt{<value>}, v, \theta) &= \text{if } v \geqslant exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
vexprsem(\text{“<”}\ \texttt{<value>}, v, \theta) &= \text{if } v < exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
vexprsem(\text{“<=”}\ \texttt{<value>}, v, \theta) &= \text{if } v \leqslant exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
vexprsem(\text{“==”}\ \texttt{<value>}, v, \theta) &= \text{if } v = exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
vexprsem(\text{“!=”}\ \texttt{<value>}, v, \theta) &= \text{if } v \neq exprsem(\texttt{<value>}, \theta) \text{ then 1 else 0} \\
getInt(i) &= (int(x), i') \\
&\quad \text{where } (x, i') = getIntSeq(\epsilon, i) \\
getIntSeq(i_1, i_2) &= \text{if } digit(head(i_2)) \\
&\quad \text{then } getIntSeq(append(i_1, head(i_2)), tail(i_2)) \\
&\quad \text{else } (i_1, i_2)
\end{aligned}
$$

4.4 Some Example Programs

This section contains a few example programs with output. You can use these to check your understanding of the syntax and semantics. In the output, italics are used to indicate the input given by the user. The “↵” symbol is used to show where a newline was printed

Code	Output
`print "Hello World!";`	Hello World

Code	Output

`get x;`
`get y;`
`get z;`
`if (x > y and y) - z`
`then`
`  print "\t It is true!\n";`
`else`
`  print "\t It is false!!\n";`
`end;`
	3
	2
	1
	It is false!!↵

Code	Output

`print "Welcome to my program.\n";`
`print "Enter a number:  ";`
`get x;`
`if x > 0`
`then`
`  x=x*2;`
`else`
`  x=-x;`
`end;`
`print x;`
	Welcome to my program.↵
	Enter a number: *3*
	6

Code	Output

`print "How many numbers will you input?  ";`
`get num;`
 	
`i=0;`
`sum=0;`
`while i < num`
`do`
` get _x;`
` sum = sum + _x;`
`end;`
 	
`q = sum / num;`
`r = sum % num;`
 	
`print "The average is ";`
`print q;`
`print " ";`
`print r;`
`print "/";`
`print num;`
`print "\n"`
	How many numbers will you input? *3*
	2
	4
	5
	The average is 3 2/3↵