CSE 6341: Lisp Interpreter Project, Part 2

Overview
In this project you will build a parser on top of the scanner from Project 1. Do not use a parser generator (e.g., yacc or CUP). Internally, instead of using a parse tree, your parser will build a binary tree representation of the input program. This binary tree will then be printed by your pretty printer. **Your submission should compile and run in the standard environment on *stdlinux*.**

Input
The input language for your parser is defined by the following context-free grammar:

```
<Start> ::= <Expr> <Start> | <Expr> eof
<Expr> ::= atom | ( <List> )
<List> ::= <Expr> <List> | ε
```

Here <Start>, <Expr> and <List> are non-terminals. There are four terminals: **atom ( ) eof** corresponding to the tokens defined in Project 1. To build the parser, it would be useful to reorganize your scanner to have an internal variable 'current' representing the current token, together with the following functions:

```
void Init() { current = getNextToken(); } // get the first token
Token GetCurrent() { return current; } // does not read anything from input
void MoveToNext() { current = getNextToken(); }
```

Parsing
This simple input language is easy to parse using the standard technique of *recursive-descent parsing*. The high-level structure of the parser is the following; **please study this pseudocode carefully**:

```
Main() { Scanner.Init(); ParseStart(); }
ParseStart() { do {
                ParseExpr(); // parse a top-level expression
                Print(); // pretty-print the binary tree for the top-level expression you just parsed
               } while (Scanner.GetCurrent() is not EOF); }
ParseExpr() {
        if (Scanner.GetCurrent() is an atom) { Scanner.MoveToNext(); } // consume atom
        else if (Scanner.GetCurrent() is open parenthesis) {
                Scanner.MoveToNext(); // consume open parenthesis
                while (Scanner.GetCurrent() is not closing parenthesis) { ParseExpr(); }
                Scanner.MoveToNext(); // consume closing parenthesis
        } else { report parse error }
}
```

This is just the "skeleton" of the parser – it does not show how the results of the parsing are actually used.

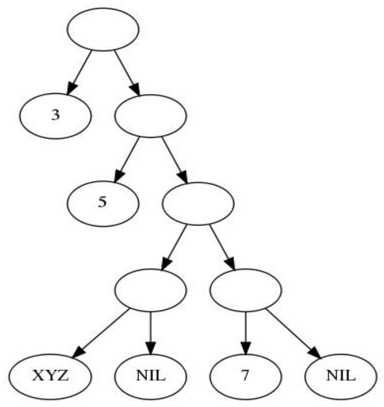
Internal Representation
Instead of building a parse tree during parsing, your implementation should build a binary tree representation that is traditionally used to represent values and programs in Lisp. In these binary trees, *leaf nodes* (i.e., nodes with no children) are atoms. Each *inner node* has exactly two children – a left child and a right child.

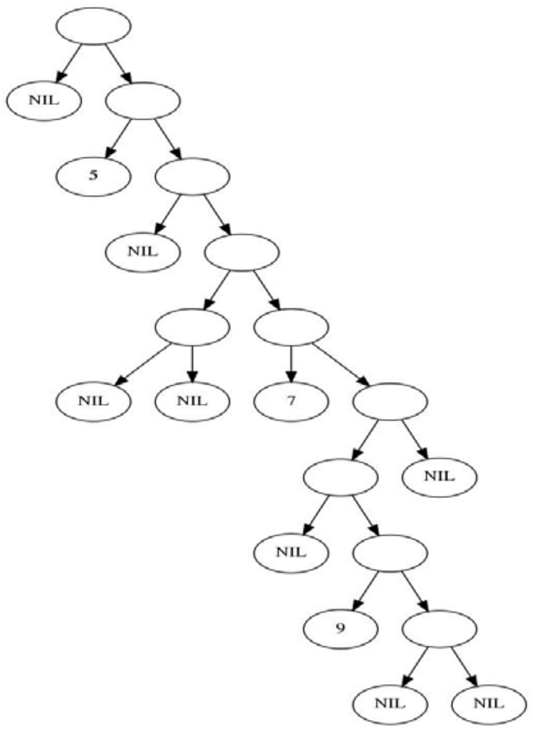
Each string derived from non-terminal <Expr> is either an atom or a list (E_1 E_2 … E_n) for $n \geq 0$. If the expression is an atom, it is represented by a binary tree containing one node (i.e., a node for that atom). If the expression is a list, it is represented by a binary tree containing n inner nodes I_1 I_2 … I_n such that

- The root of the tree is I_1
- The left child of I_k is the root of the binary tree for E_k for $1 \leq k \leq n$
- The right child of I_k is I_{k+1} for $1 \leq k \leq n-1$
- The right child of I_n is a leaf node corresponding to the special literal atom NIL

In the case when n=0 (that is, we have an empty list), the binary tree contains one leaf node corresponding to the special literal atom NIL.

Examples: the binary tree representations for inputs (3 5 (XYZ) 7) and (NIL 5 ( ) (( )) 7 (( ) 9 ( ))) are





Output
As in Project 1, all output should go to UNIX *stdout*. This includes error messages – do not print to *stderr*.

A *top-level expression* is a sequence of tokens that can be derived from non-terminal <Expr> based on production <Start> ::= <Expr> <Start> | <Expr> **eof**. In other words, this is an expression that is *not* contained inside another expression. The input is a sequence of top-level expressions ending with **eof**. For each input top-level expression, parse it and then pretty-print it immediately after it is parsed. If it cannot be parsed, an error message is printed and the parser exits immediately to the operating system (the rest of the input is ignored). The error message should be ERROR: …. where ERROR is in upper case and the string explains briefly the parsing problem.

If the top-level expression can be successfully parsed, pretty-print it followed by newline. For this project, pretty printing will use the so-called "dot notation". In later projects we will use the more compact "list notation" which will be discussed in class. Given the binary tree representation of an input expression, the dot notation is very simple. If the tree is a leaf, just print that atom. Otherwise, print "(" followed by the dot notation for the left subtree followed by " . " followed by the dot notation for the right subtree followed by ")". Note the spaces around the dot. For example, if the input contains the four top-level expressions

123 (3 5 (XYZ) 7) (NIL 5 ( ) (( )) 7 (( ) 9 ( )))(DEFUN F23 (X) (PLUS X 12 55))

the output will be
```
123
(3 . (5 . ((XYZ . NIL) . (7 . NIL))))
(NIL . (5 . (NIL . ((NIL . NIL) . (7 . ((NIL . (9 . (NIL . NIL))) . NIL))))))
(DEFUN . (F23 . ((X . NIL) . ((PLUS . (X . (12 . (55 . NIL)))) . NIL))))
```

Invalid Input

To emphasize a point made earlier: your parser not only should process correctly all valid input, but also should recognize and reject invalid input (i.e, strings that do not belong to the language defined by the grammar). Handling of syntactically-invalid input is part of the language semantics, and it will be taken into account in the grading of the project. Your interpreter should not crash on invalid input (no segmentation faults, no uncaught exceptions, etc.). If an input expression cannot be parsed, a message "ERROR: (brief explanation of parsing problem)" should be printed and the interpreter should exit back to the OS; the rest of the input file will be ignored. Your score will partially depend on the handling of incorrect input and on printing error messages as described above.

Project Submission

On or before 11:59 pm, **January 26 (Thursday),** you should submit the following:

- One or more files for the scanner (source code)
- A makefile Makefile such that *make* on *stdlinux* will build your project to executable form.
- A text file called Runfile containing a single line of text that shows how to run the interpreter on *stdlinux*.
- If there are any additional details the grader needs to know in order to compile and run your project, please include them in a separate text file README.txt

Login to *stdlinux* in the directory that contains your files. Then use the following command:

```
submit   c6341ab   lab2   Makefile    Runfile     sourcefile1    sourcefile2 …
```

Make sure that you submit only the files described above: do not submit files x.o, x.class, x.doc, etc.

Important: every time you execute *submit*, it **erases all files** you have submitted previously. If you need to resubmit, submit all files, not only the changed files. If you resubmit after the project deadline, the grader cannot recover older versions you have submitted earlier – only the last submission will be seen, and it will be graded with a late penalty. If the grader asks you to make some changes, email him/her the modified files rather than resubmitting them through *submit*.

If the time stamp on your electronic submission is **12:00 am on the next day or later**, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will **not** be accepted and you will receive zero points for this project. If the grader has problems with compiling or executing your program, she/he will e-mail you; you must respond within 48 hours to resolve the problem. Please check often your email accounts after submitting the project (for about a week or so) in case the grader needs to get in touch with you.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own: all design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or for documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with **severe** consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see *http://oaa.osu.edu/coamresources.html*). I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.