CS 441 Assignment 2 – Scanner / Parser

Learning Goals:

- Use of LLM for problem decomposition & code drafting
- LLM-assisted construction of a parser, either by building everything ad-hoc, or using tools & libraries provided by Racket to generate them.
- Code documentation

For this assignment, you'll build a scanner and parser for a simple programming language.

The Scanner

A scanner, also called a lexer, takes in a text file and turns out a list of language tokens. There are two basic approaches. In the first, the scanner runs in its entirety before parsing begins. The scanner is a function (possibly with some auxiliary or sub-functions) that takes in the input file as a single large string, or as a list of strings, one line per string—Racket has input functions that will do either. The scanner function then returns a list of tokens, consisting of data items, each beginning with a label, followed by its value or string representation:

```
'( (id "x") (assign ":=") (id "y") (plus "+") (num  "23") (times "*")
(lparen "(") (id "sqrt") (id "a") (rparen ")")  (colon ":"))
```

The parser then processes this list of tokens.

Another approach is to call the parser first, and have the parse proceed incrementally; when the parser is ready to continue, it will call a `next-token` function to get the next single token, which is retrieved from input at that time (or returned from a list built up earlier). This method might require managing state, which will be more of a challenge in a functional language.

Whether the scanner proceeds incrementally, one token at a time, or processes the entire input file and returns a list of tokens, is an implementation detail. You may use whichever seems most straightforward.

Note that if the scanner detects something that cannot possibly be part of the grammar—for example, this grammar has no use for the '$', '^', or '@' characters, so they shouldn't appear outside a quoted string—then your scanner may declare an error and end the program (though a diagnostic message and some indication about where the error was found would be helpful).

You may use any of the scanner-building tools provided by Racket. You may use any large language model (LLM) you choose for assistance. As with program 1, save your transcripts.

The parser

The parser's job is to determine whether the tokens follow the grammar, and to build up a parse tree representing the abstract structure of the program. In this grammar, a program is a list of statements. A statement can be represented as a list of tokens, with a suitable label at the head of the list to indicate its type. For example, starting with

```
'((ID  "X") (assign-op ":=") (ID "Y") (add-op "+") (Integer 1))
```

we go through the grammar and eventually get a somewhat more complex list:

```
((STMT (ID "x") (assign-op ":=") (expr (and-expr (not-expr (compare-expr (add-expr
(mult-expr (negate-expr (value (id "Y")))) "+" (add-expr (negate-expr (value
(constant (integer 1))))))))))))
```

What on earth is all that? That's what we get from working down our grammar—each function identifies what its component should be, and calls the appropriate function; when the function returns, it's labeled what it found:

```
(
  (STMT
     (ID "x")
     (assign-op ":=")
     (expr
       (and-expr
          (not-expr
            (compare-expr
              (add-expr
                (mult-expr
                  (negate-expr
                     (value
                        (id "Y")
                     )
                  )
                 )
                “+"
                (add-expr
                  (negate-expr
                      (value
                         (constant
                           (integer 1)
)))))))))) ) (full cascade of closing parens not shown, to save space)
```

But after going through all that, we've verified that "`X := Y + 1`" is in fact a valid statement in this grammar. If every statement in the grammar is valid, then we'll have a list of statements—a program.

Extra Credit! Remember in our discussion of the compilation process, we discussed how the code optimizer, while not strictly necessary, is usually part of a compilation system. This is particularly important for compilers, where we're writing the machine code. But the same principle applies to interpreters. For example, if the above statement were simplified to

```
(STMT (ID "x") (assign-op ":=") (expr (id “Y”) (add-op “+”) (integer 1)))
```

it would be much easier to understand if printed, and would execute much faster without having to drill down through multiple layers of sublists. For up to 10 points (1 letter grade) extra credit, write your parser so that once correctness is determined, the intermediate levels that are no longer needed are deleted.

The grammar itself is listed at the end of this document.

Deliverables:
You will submit the following:

- The Racket code file containing your parser. YOU MUST SUBMIT CODE AS A TEXT FILE. DO NOT PASTE THE CODE INTO A WORD PROCESSOR OR PDF. **WORD PROCESSING DOCUMENTS OR PDFS SUBMITTED AS CODE WILL NOT BE GRADED.**
- The code files you used to test your parser. Again, these must be plain text files or they will not be counted.
- The transcript of your LLM interactions. As with the last program, you need not save your interaction with the LLM when working out principles, reviewing material in general, researching code libraries, etc., but should include anything related to writing code.
- A 5-10 minute Panopto video describing your process and walking through your code. You'll want to do this on your computer, not your phone, as you'll need to screen-share your code. The video can be recorded directly in Panopto, or in Zoom. You should include:
    - A brief discussion of what sources you used
    - The main features of the code—what were your significant design decisions? How did you go about it?
    - How did you test the code?
    - Any problems with coding, LLM interactions, etc.
    - What are you proudest of about the code? Did you have any pleasant surprises, things that seemed to fall together well?
    - What gave you the most trouble? Were there parts of it that you just couldn't get to work? (Were there parts you didn't *think* you could get to work, but you finally figured out?) How close did you get? (If it's not fully functional as the deadline approaches, *turn in what you've got;* If you don't turn anything in, I have no choice but to give a grade of 0.)

This grammar is not case sensitive. 'If', 'if', and 'IF' are equivalent. This is often dealt with by converting everything to one case, usually lower-case, and going from there. Though in that case, one must be careful not to convert quoted strings.

This grammar is not LL(1) – there are 2 rules starting with ID – but I believe it's LL(2).

```
"Start Symbol"   = <Lines>
```

{String Chars} = Any printable character EXCEPT the double quote """, which cannot appear inside a string, only as a string delimiter.

```
{WS}            = space, tab, CR, LF characters

NewLine         = {CR}{LF}|{CR}
Whitespace      = {WS}+

Remark          = REM{Space}{Printable}*
```

NOTE: This is borrowed from BASIC. The REMark indicates the rest of the line is a comment and can be disregarded, but *the remark is itself a statement.* This means that

```
Y := X + 1  REM increment index
```

is a syntax error. It should be

```
Y := X + 1 : REM increment index
```

Note the presence of the colon to indicate the end of the first statement on the line, separating the assignment statement from the remark statement.

```
ID              = {Letter}{Letter | digit}*
String          = '"'{String Chars}*'"'
Integer         = {digit}+
Real            = {digit}+.{digit}+

<Lines>       ::= <Statements> NewLine <Lines>
                | <Statements> NewLine

<Statements>  ::= <Statement> ':' <Statements>
                | <Statement>

<Statement>   ::= DEF ID ( <ID List> )
                | ENDDEF
                | END
                | IF <Expression> THEN <Statements> ENDIF
                | ID ':=' <Expression>
                | ID ( <Expression List> )
                | PRINT <Print list>
                | RETURN <Expression>
                | WHILE <Expression> DO <Statements> ENDWHILE
                | Remark

<ID List>  ::= ID ',' <ID List>
             | ID

<Value List>      ::= <Value> ',' <Value List>
                    | <Value>

<Constant List>   ::= <Constant> ',' <Constant List>
```

```
                    | <Constant>

<Integer List>    ::= Integer ',' <Integer List>
                    | Integer

<Expression List> ::= <Expression> ',' <Expression List>
                    | <Expression>

<Print List>      ::= <Expression> ';' <Print List>
                    | <Expression>
                    |

<Expression>  ::= <And Exp> OR <Expression>
                | <And Exp>

<And Exp>     ::= <Not Exp> AND <And Exp>
                | <Not Exp>

<Not Exp>     ::= NOT <Compare Exp>
                | <Compare Exp>

<Compare Exp> ::= <Add Exp> '='  <Compare Exp>
                | <Add Exp> '<>' <Compare Exp>
                | <Add Exp> '><' <Compare Exp>
                | <Add Exp> '>'  <Compare Exp>
                | <Add Exp> '>=' <Compare Exp>
                | <Add Exp> '<'  <Compare Exp>
                | <Add Exp> '<=' <Compare Exp>
                | <Add Exp>

<Add Exp>     ::= <Mult Exp> '+' <Add Exp>
                | <Mult Exp> '-' <Add Exp>
                | <Mult Exp>

<Mult Exp>    ::= <Negate Exp> '*' <Mult Exp>
                | <Negate Exp> '/' <Mult Exp>
                | <Negate Exp>

<Negate Exp>  ::= '-' <Value>
                | <Value>

<Value>       ::= '(' <Expression> ')'
                | ID
                | ID '(' <Expression List> ')'
                | <Constant>

<Constant> ::= Integer
             | String
             | Real
```