P2: Parser and Tree Builder
Due Wednesday, April 6th at 11:59pm

- Implement parser and tree builder for the provided grammar (see below).
- Verify the project grammar is LL(1) or rewrite as needed in an equivalent form.
- Have your parser generate error (line number and tokens involved) or print OK message upon successful parse.
- Use your scanner module and fix if needed.
- Project P2 will be tested assuming white spaces separate all tokens.

- Invocation:

```
frontEnd [file]
```

  - As in previous projects, a filename may be specified or the information can be read from stdin
  - Wrong invocations may not be graded
  - ***Program must compile and run on clark.rnet.missouri.edu***
- Additional requirements:
  - Implement the parser in a separate file (parser.c and parser.h) including the initial auxiliary parser() function and all nonterminal functions.
  - Call the parser function from main.
  - The parser function generates error or returns the parse tree to main.
  - In testTree.c (and testTree.h) implement a printing function using preorder traversal with indentations as before for testing purposes (2 spaces per level, print the node's label and any decorations (e.g. specific ID name) from the node; one node per line).
  - Call the printing function from main immediately after calling the parser and returning the tree.
  - The printing function call must be later removed for P3 unless debugging.

BNF

(**Please ensure this uses only tokens detected in your P1, *no exceptions***.) <S> is the starting nonterminal.

<S> -> **Name Identifier Spot Identifier** <R> <E>
<R> -> **Place** <A> <B> **Home**
<E> -> **Show Identifier**
<A> -> **Name Identifier**
<B> -> empty | **.** <C> **.** <B> | <D> <B>
<C> -> <F> | <G>
<D> -> <H> | <J> | <K> | <L> | <E> | <F>
<F> -> **{ If Identifier** <T> <W> <D> **}** | **{ Do Again** <D> <T> <W> **}**
<G> -> **Here Number There**
<T> -> **<<** | **<-**
<V> -> **+** | **%** | **&**
<H> -> / <Z>
<J> -> **Assign Identifier** <D>
<K> -> **Spot Number Show Number** | **Move Identifer Show Identifier**
<L> -> **Flip Identifier**
<W> -> **Number** <V> **Number** | **Number .**
<Z> -> **Identifier** | **Number**

Lexical Definitions (same as project P1)

- All case sensitive
- Alphabet:
  - All upper- and lower-case English letters, digits, plus the extra characters as shown below, plus WS
  - *No other characters are allowed and they should generate errors*
  - Each scanner error must display "SCANNER ERROR:" followed by input string and line number
- Identifiers
  - begin with a ***lower-case*** letter (a-z) and
  - continue with ***one or more*** letters or digits (no underscores)
    - e.g. d3, aDD8, z920, bfa, are all valid and a, A3, 382, are all invalid identifiers
- Keywords
  - **`Again If Assign Move Show Flip Name Home Do Spot Place Here There`**
- Operators and delimiters group
  - **`&  +  /  %  .  {  }  <<  <-`**
- Numbers
  - any sequence of decimal digits (0-9), no sign, no decimal point
- Comments start with **`*`** and end with **`*`**

P2 Suggestions

- Ensure the grammar is LL(k=1).
  - Note that left factorization need not be applied when **all** RHS productions begin with the same α, as this can be handled in implementation (as discussed in class).
- Note that the parser calls the scanner, but the parser may need some setup in main.
- Implement the parser in two iterations:
  - Starting without the parse tree.
    - Have your parser generate error (line number and tokens involved) or print OK message upon successful parse.
    - For each <nonterminal>, use a void function named after the nonterminal and use only explicit returns.
    - Decide how to pass the token.
    - Have the main program call the parser, after setting up the scanner.
    - Be systematic: assume each function starts with unconsumed token (not matched yet) and returns unconsumed token.
    - Use version control and be ready to revert if something gets messed up.
  - Only after completing and testing the above to satisfaction, modify each function to build a subtree, and return its root node.
    - Assume each function builds just the root and connects its subtrees.
    - Modify the main function to receive the tree built in the parser, and then display it (for testing) using the preorder treePrint().
- Some hints for tree:
  - Every node should have a label consistent with the name of the function creating it (equal to the name?)

  - Every function creates exactly one tree node (or possibly none)
  - All syntactic tokens can be thrown away, all other tokens (operators, IDs, Numbers) need to be stored
  - When storing a token, you may need to make a copy depending on your interface

Testing

- Create files using the grammar to generate programs, starting with simplest programs, adding one different statement at a time and then building sequences of statements and nested statements.
- Make sure to have sequences of statements, nested statements (blocks using **`Begin`**…**`End`**), nested Ifs and loops (**`Repeat`**), variables in various blocks, etc, and to test all operators.
- You may skip comments but then test comments in some files.
- ***Feel free to share test files with others and compare output.***
- Some example test files:

Example 1 input:

```
Name prog1
Spot prog1
Place
Name id1
Home
Show prog1
```

Example 2 input:

```
Name prog2
Spot prog2
Place
Name id2
/ id1
Show id2
Home
Show prog2
```

Example 3 input:

```
Name prog3
Spot prog3
Place
Name id1
```

```
.
Here 2 There
.
Show id1
Home
Show prog3
```

Example 4 input:

```
Name prog4
Spot prog4
Place
Name id2
.
{ If id2 <- 5 . Show id2 }
.
Home
Show prog4
```

Grading

- Programming and architectural style: 10 points
- Execution correctness: 90 points

Programs missing makefile or that won't properly compile on clark will not be graded.