CS 5500 – The Structure of a Compiler
HW #2

- This assignment is **due by 11:59 p.m. on Monday, Sept. 9, 2019**.
- This assignment will be worth **5%** of your course grade.
- A grading rubric also is posted on Canvas so that you can see approximately how many points each functional requirement is worth.
- You are to work on this assignment **by yourself**.
- You are strongly encouraged to **take a look at all of the sample input and output files** posted on the Canvas website **before** you actually submit your assignment for grading.

Basic Instructions:

For this assignment you are to write a program that will perform **syntax analysis** for the MIPL programming language. Specifically, you are required to implement a **recursive-descent parser**. You can use any programming language as long as your program compiles/interprets and executes on the cslinux machines. You are **NOT** allowed to use ***flex*** for your lexical analyzer and you are **NOT** allowed to use ***bison*** or ***antlr*** for the parser; you will receive **NO CREDIT** for this assignment if you do so – and we will be checking for that!

Because we are allowing you to implement your program in any language, you will have to submit a ***make*** file. There is plenty of information/documentation available on the internet for creating *make* files for various languages – look it up! We will be testing your program on several different input files. Your program should expect **the name of the input file as a command line argument;** name the input file **input.txt** in the command line of your make file. Your program should not require any other command line arguments. We use a script to grade the programs and it will not be providing any other command line arguments.

Your program should **output information about each token** that it encounters **and each production** it processes in the input source program. In later parts of this compiler project you will want to have the **ability to suppress** this output, so create an internal Boolean variable that you can set to turn on/off this functionality; do **NOT** use a command line argument for this purpose as our grading script will not be looking for that!

Your program should **continue parsing** an input file **until end-of-file is detected or a syntax error is detected**. Note that your program should **NOT** do any other kind of processing (e.g., no type checking, etc.).

MIPL Programming Language Description

You should review the MIPL (Mini Imperative Programming Language) token types defined in HW #1, and **fix anything in your lexical analyzer program that was not working correctly**; otherwise, it will likely cause you problems on this assignment.

In general, MIPL defines the basic variable types *integer*, *char*, and *boolean*, as well as the predefined Boolean constants *true* and *false*. Users may define variables of any of the above types, as well as one-dimensional arrays (indexed by integers) that contain elements of any one of the above types. Parameterless procedures are allowed and can have other procedure declarations nested within them. Procedures may have local variables, and may contain recursive calls. The following types of statements are allowed: precondition loops, conditional statements, procedure calls, compound statements, input statements, output statements, and assignment statements.

A context-free grammar for MIPL is given below where **< >**'s around a symbolic name are used to identify a nonterminal, and boldfaced and/or italicized symbols represent terminals:

Change your regular expression for *intconst* so that it now includes the optional sign (e.g., -23 should not be processed as two separate tokens anymore)

<prog lbl> → **program**

<prog> → <prog lbl> *ident* **;** <block> .

<block> → <var dec part> <proc dec part> <stmt part>

<var dec part> → ε | **var** <var dec > **;** <var dec lst>

<var dec lst> → <var dec> **;** <var dec lst> | ε

<var dec> → <ident> <ident lst> **:** <type>

<ident> → *ident*

<ident lst> → **,** <ident> <ident lst> | ε

<type> → <simple> | <array>

<array> → **array [** <idx range> **] of** <simple>

<idx> → ~~<int const>~~ *intconst*

<idx range> → <idx> **..** <idx>

<simple> → **integer** | **char** | **boolean**

<proc dec part> → <proc dec> **;** <proc dec part> | ε

<proc dec> → <proc hdr> <block>

<proc hdr> → **procedure** *ident* **;**

<stmt part> → <compound>

<compound> → **begin** <stmt> <stmt lst> **end**

<stmt lst> → **;** <stmt> <stmt lst> | ε

<stmt> → <assign> | ~~<proc stmt>~~ | // Assume proc calls can't happen for now!
<read> | <write> | <condition> | <while> | <compound>

<assign> → <variable> **:=** <expr>

<proc stmt> → <proc ident>

<proc ident> → *ident*

<read> → **read (** <input var> <input lst> **)**

<input lst> → **,** <input var> <input lst> | ε

<input var> → <variable>

<write> → **write (** <output> <output lst> **)**

<output lst> → **,** <output> <output lst> | ε

<output> → <expr>

<condition> → **if** <expr> **then** <stmt> <elsePart>
<elsePart> → **else** <stmt> | ε

<while> → **while** <expr> **do** <stmt>

<expr> → <simple expr> <opExpr>
<opExpr> → <rel op> <simple expr> | ε // test whether token in FIRST(<rel op>)

<simple expr> → <term> <add op lst>

<add op lst> → <add op> <term> <add op lst> | ε

<term> → <factor> <mult op lst>

<mult op lst> → <mult op> <factor> <mult op lst> | ε

<factor> → <sign> <variable> | <const> | **(** <expr> **)** | **not** <factor>

<sign> → **+** | **-** | ε

<add op> → **+** | **-** | **or**

<mult op> → ***** | **div** | **and**

<rel op> → < | <= | <> | = | > |>=

<variable> → *ident* <idx var>

<idx var> → **[** <expr> **]** | ε

~~<array var> → <entire var>~~

~~<entire var> → <var ident>~~

~~<var ident> → *ident*~~

<const> → *intconst* | // Change intconst regexpr to (optionally) include sign
charconst | <bool const>

~~<int const> → <sign> *intconst*~~

<bool const> → **true** | **false**

See the sample output files for **the particular names of terminals and nonterminals that we want you to output**.

Error Detection

When a syntax error is detected, **output "syntax error" preceded by the line number in the input file where the error was detected by the parser, and terminate the parse**. Don't output what token(s) you were expecting; our compiler is not that helpful ☹ Examples of errors are given in the sample output files. Note that where the parser detects the error will not always be the same line as where the error actually occurred.

Sample Input and Output:

Sample input and output files are posted on Canvas. Note that these files were created on a Windows PC, so you might need to run *dos2unix* on them before using them on a Linux/Unix machine. With the exception of whitespace and capitalization, the output produced by your program **MUST** be **IDENTICAL** to what our "grader" program produces! Use exactly the same terminal and nonterminal names, messages, etc. that are used in the sample output files. **Otherwise, the grading script will say your output is wrong!**

You can compare your output (e.g., **myOutput.out**) with the corresponding expected output file posted on Canvas (e.g., **input.txt.out**), ignoring differences in spacing and upper vs. lower case, using the following command (typed all on one line):

diff myOutput.out input.txt.out --ignore-space-change --side-by-side --ignore-case --ignore-blank-lines

To learn more about the *diff* command, see http://ss64.com/bash/diff.html

What To Submit For Grading:

You should submit via Canvas a single **zip** file containing **only source code files and a *make* file** for your homework submission (i.e., no data files!). Name your zip file using your last name followed by your first initial (e.g., Homer Simpson would name his **simpsonh.zip**).

***WARNING*: If you fail to follow all of these instructions, the automated grading script may reject your submission, in which case it will NOT be graded!!!** ☹