Term Project _ Option 1: Parser Generator
CS 2336.004
Due Date: April 30th, 2017

In this project option, you are going to implement lexical/syntax analysis using Stack.

Syntax-Directed Translations
The front end of the compiler constructs a intermediate representation of the source program from which the back end generates the target program.
A syntax directed translation scheme is a syntax directed definition in which the net effect of semantic actions is to print out a translation of the input to a desired output form.

Stack
Stack is a LIFO (last in first out) storage with two abstract operations : push, pop. Push will put an item into stack at the top. Pop retrieve an item at the top of stack.

Calculations using stack
Because a stack is LIFO, any operation must access data item from the top. Stack doesn't need 'addressing' as it is implicit in the operators which use stack. Any expression can be transformed into a postfix order and stack can be use to evaluate that expression without the need for explicitly locate any variable. For example:

B + C - D ==>
B C + D - (post fix)

Intermediate Code:

```
push rvalue B
push rvalue C
add
push rvalue D
sub
```

Instruction set of stack
1. Instructions fall into three category:
   - a. Integer arithmetic: add, mul, div ...
   - b. Stack manipulation:
     - b.i. Push v: push v onto stack
     - b.ii. push lvalue L: push address of data location L
     - b.iii. push rvalue L: push contents of data location L
     - b.iv. pop: throw away the value on stack top
     - b.v. := : the rvalue on the top is placed in the lvalue below it and both are popped.
     - b.vi. copy: push a copy of top value on the stack.
   - c. Control flow:
     - c.i. label L: taget of jump to L
     - c.ii. goto L: nect instruction is taken from statement with label L
     - c.iii. gofalse L: pop the top value ; jump if it is 0.
     - c.iv. gotrue L: pop the top value ; jump if it is non zero.

c.v. halt: stop execution

Steps to be followed while working on this project

1. Lexical specification:

To do lexical analysis of a C-routine. The scanner must be able to process contructs like expressions, assignment statements, if statement, if-else-if nesting and while loop. Ensure handling multiple files for scanning.

2. Yacc specification:

Syntax analysis of the above construct. The code can be assumed to be semantically correct, so no semantic checks using yacc action need to be done.

3. Translation:

a. Expression:

a.i. expr -> expr1 ‘op’ expr2 { expr.t := expr1.t || expr2.t || op }

a.ii. expr-> id { expr.t := id.lexeme }

Attribute lexeme of an id gives its string representation.

Attribute t of a non terminal gives its translation.

|| is the concatenation operator.

b. Assignment:

b.i. stmt -> id := expr {stmt.t := ‘push lvalue ’ id.lexeme || expr.t || ‘:=’}

c. If statement:

c.i. stmt -> if expr then stmt1 {out := newlabel(); stmt.t := expr.t || ‘gofalse ’ out || stmt1.t || ‘label ’ out }

d. While statement:

d.i. stmt -> while expr then stmt1 {test = newlabel(); out = newlabel(); ‘label ’ test || expr.t || ‘gofalse ’ out || stmt1.t || ‘goto ’ test || ‘label ’ out }

Input guidelines

1. Input is a C-routine.
2. Ignore function calls.
3. The input must be completely realized by the above four translations (for example the binary search routine)

Example Input:

```
int n = 10;
int u = 0 ;
int v = 1 ;
int t ;
int i = 2;
while(i <= n ) {
        t = u + v; u = v; v = t;
}
```

Output:

```
push lvalue n
push 10
:=
push lvalue u
push 0
:=
push lvalue v
push 1
:=
push lvalue i
push 2
:=
label test
push rvalue i
push rvalue n
<=
gofalse out
push lvalue t
push rvalue u
push rvalue v
+ :=
push lvalue u
push rvalue v
:=
push lvalue v
push rvalue t
:=
goto test
label out
halt
```