1 Introduction

For this project, we will construct a program that parses and evaluates a simple script that involves variable declarations, assignments, and arithmetic expressions. For example, such a script could look like this:

```
program Example;
    var x, y : integer;
begin
    x = 5;
    y = 2*x + 1;
    print( x+y );
end
.
```

Flex and Bison are tools for building programs that handle structured input, such as the piece of text above. To analyze the structure of the input, the job is divided into two parts: lexical analysis (also called lexing or scanning) and syntax analysis (or parsing). Scanning divides the input into meaningful chunks, called tokens, and parsing figures out how the tokens relate to each other. Flex is responsible for creating a scanner and bison for creating a parser. The parser will use the scanner as a subroutine.

We will instruct flex to create a program that breaks up the string above into a sequence of tokens. (Tokens are represented as integers.) And each token can have an associated value called semantic value. For example, `5` and `2` will both be translated to tokens `INT`. Note that the tokens are both called `INT`, but their semantic values are different: 5 and 2 respectively. The complete scanning of the text produces the following sequence of tokens.

```
PROGRAM ID;
    VAR ID, ID : TYPE;
BGN
    ID = INT;
    ID = INT*ID + ID;
    PRINT( ID+ID );
END
.
```

Next, to parse a sequence of tokens, we need to define a context-free grammar (CFG), a.k.a. syntax diagram, that generates such a sequence.

Recall the definition of a CFG, which has variables and terminals. Here, the tokens as well as punctuation characters are terminals. We will use the following CFG. Bold words and symbols as terminals.

program →
    program id ;
    declaration_list
    begin
    statement_list
    end
    .
declaration_list →
    declaration declaration_list
    | ϵ
declaration → **var** *identifier_list* **: type ;**
identifier_list →
    id
    | **id ,** *identifier_list*
statement_list →
    ε
    | *statement statement_list*
statement →
    id asgnop *expression* **;**
    | **print (** *expression_list* **) ;**
expression_list →
    ε
    | *expression*
    | *expression* **,** *expression_list*
expression →
    term
    | **addop** *term*
    | *expression* **addop** *term*
term →
    factor
    | *term* **mulop** *factor*
factor → **id**
    | **num**

| (*expression*)

num stands for integer or float literals; **addop** stands for addition or subtraction; **mul** stands for multiplication or division; and **asgnop** stands for assignment.

2 Flex

To instruct flex to create a C program (called scanner) that converts a string into a sequence of tokens, we have to specify the rules for matching patterns with tokens. The patterns are defined using regular expressions. For example, the expression `[0-9]` matches any digit from 0 to 9, and `[0-9]+` matches any non-negative integer in decimal, e.g. `56`, `015`, etc. The action we may want to perform when encountering the pattern `[0-9]+` is to set the semantic value of the token equal to the corresponding integer and return the token `INT`.

A flex input file consists of three sections, separated by a line containing only `%%`.

```
definitions
%%
rules
%%
user code
```

The definitions section contains declarations of simple name definitions to simplify the scanner specification. For example, we can name a pattern with `NUMBER [0-9]+`. We can also include C header files in the definitions section.

The user code section contains the C code for any subroutine you would like to define and use during the scanning process.

Most important is the rules section. The rules section contains a series of rules of the form:

```
pattern      action
```

where the pattern must be unindented and the action must begin on the same line. For example,

```
[0-9]+   {yylval.n = atol(yytext); return INT;}
```

The variable `yytext` is part of the scanner and holds the string that currently matches the pattern. The variable yylval is of a C union type and should hold the semantic value of the current token.

3 Bison

Bison will create a parser for a sequence of tokens according to the given CFG. In order to be useful, the program must do more than just parsing; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an associated action made up of C statements. Each time the parser recognizes a match for that rule, the action is executed. For our purpose, an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, consider the rule:

$expression \to expression + term$

Then we can associate with that rule the action

```
$$ = newexpr(ADD, $1, $3);
```

where `$$` means the semantic value of the top expression, `$1` the semantic value of the subexpression, and `$3` the semantic value of the term. The function `newexpr` will take the semantic values `$1` and `$3` as input and compute the semantic value `$$`. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how the sum was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

In the example above, the function `newexpr` doesn't directly add two semantic values together. Instead, it creates a node that describes the addition. Later, the evaluation of the node will be done at the end of the parsing. In other words, we'll be building an *abstract syntax tree*. An abstract syntax tree is basically a parse tree that only contains the necessary information for evaluation. The syntax is "abstrac" in the sense that it does not represent every detail appearing in the parse tree, but rather just the structural or content-related details. For instance, parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes.

4 Documentation

Documentation for flex: `https://westes.github.io/flex/manual/`. Documentation for bison: `https://www.gnu.org/software/bison/manual/`.