Passing data objects from lexer to parser (using Flex and Bison)

Nikhil Hegde, Department of CSE, IIT Dharwad

- Goal: recognize an INTLITERAL and STRINGLITERAL in the program text, print the line number and the value of the INTLITERAL as a semantic action in the parser.

```
STRINGLITERAL val:"Hello" at line:6
INTLITERAL val:1 at line:7
INTLITERAL val:2 at line:7
INTLITERAL val:0 at line:9
INTLITERAL val:10 at line:10
INTLITERAL val:1 at line:12
Accepted
```

```
1  PROGRAM test
2  BEGIN
3          INT a,b,c,x,y,z,h,j,k;
4    FUNCTION INT main()
5          BEGIN
6          STRING str:="Hello";
7          a := 1; b:=2;
8          IF (a = b)
9              j := 0;
10             WHILE (j <= 10)
11                     a := j;
12                     j := j+1;
13             ENDWHILE
14         ENDIF
15         RETURN a+b;
16         END
17 END
```

scanner.l file

A global variable to keep track of line numbers

`yylval` is the name of the data object passed from lexer to parser. `yylval` is the variable name. This variable is of type YYSTYPE, which is defined as a union in parser.

`intliteraldata` is the name (of 'field') in the data object's fields defined

type of the field is `std::pair<int, int>`

```
%{
#include<iostream>
#include<cstring>
#include "microParser.tab.hpp"
int lineCount=1;
%}

%%

[0-9]+                  { yylval.intliteraldata=new std::pair<int, int>(); (yylval.intliteraldata)->first=atoi(yytext)
;(yylval.intliteraldata)->second=lineCount;return INTLITERAL;}
\"[^\"]*\"              { yylval.stringliteraldata=new std::pair<std::string, int>; (yylval.stringliteraldata)->first=
std::string(yytext);(yylval.stringliteraldata)->second=lineCount;return STRINGLITERAL;}
\n                      {lineCount++;}
```

type of the field is `std::pair<std::string, int>`

increment counter tracking line number when 'newline' symbol is seen in input file

microParser.ypp file

`iostream` is needed for `std::pair`
Whatever you put in between %{ and %} is copied to microParser.tab.cpp file but not microParser.tab.hpp file.

This is the type of the data object passed from lexer to scanner. This object is a union and has fields `intliteraldata` and `stringliteraldata`. This union cannot contain only basic data types such as `int, float, char,` and pointers.

As the details of this type is required by the scanner and as this type contains `std::string` and `std::pair`, `scanner.l` needs to include corresponding headers. This inclusion must be done before `#include"microParser.tab.hpp"` because the `.hpp` file does not include these headers as mentioned above. Also note that all the fields inside `%union` are pointers types.

```
%{
#include<iostream>
int yylex();
void yyerror(char const* errmsg);
%}

%union{
std::pair<int, int>* intliteraldata;
std::pair<std::string, int>* stringliteraldata;
}

%token <intliteraldata> INTLITERAL
%token STRINGLITERAL
%type <stringliteraldata> STRINGLITERAL
%%
program: PROGRAM id _BEGIN pgm_body END {printf("Accepted\n");return 0;};

    ..other CFG rules go here

str: STRINGLITERAL {printf("STRINGLITERAL val:%s at line:%d\n",(($1)->first).c_str(), ($1)->second);delete $1;};
primary: LPAREN expr RPAREN {}
       | id {}
       | INTLITERAL {printf("INTLITERAL val:%d at line:%d\n",($1)->first, ($1)->second);delete $1;}
       | FLOATLITERAL {};
```

Note two ways of associating fields of the data object with tokens: 1) `%token <intliteral> INTLITERAL`. In this case you don't have to separately define `%token INTLITERAL` 2) `%token STRINGLITERAL` followed by `%type<stringliteraldata> STRINGLITERAL`. In this case, we mean explicitly the type of `stringliteral` is the type of the semantic record of STRINGLITERAL token.

- ($1) is the reference to data object of STRINGLITERAL. Whenever the category of the token matched in scanner is a STRINGLITERAL, the scanner creates an object of type as that of stringliteraldata, initializes it with appropriate values and sends it to parser. The return statement in scanner only returns the token category. The token value is set by the scanner using `yylval` object. The parser refers to this object using $1 in this case in the semantic action using ($1).
- c_str() is needed to convert std::string to C-style strings (char *). If you use cout to print you don't need this.
- `first` and `second` are the names by which you access the fields of a `std::pair` object
- delete the object created if no longer required.