CSE4100: Compilers **Fall 2015**

Homework 3: LALR(1) Parsing (Part I)

Out Date: 10/01/2015
Due Date: 10/08/2015

Objectives

The purpose of this homework is to write the parser for `C--`. You will be using bison to produce a `C` file with functions that implement your parser. By the end of this homework you will actually recognize whether a string belongs to the language `C--`. Producing an actual parse tree is the focal point of the next homework. *It is highly advisable to go over the documentation for bison in order to complete this homework.*

Word of caution This is a big homework. You will not get an extension. You should start **immediately**. Pace yourself! I strongly encourage everyone to make use of the documentation (of bison). Work incrementally, testing small pieces of the grammar. Do not attempt to write everything and test all at once.

Handout

There is very little code handed out. Your starting point is the end-product of your lexical analyzer homework (or the solution that is handed out as part of this assignment). Since you already have a grammar file (i.e., **`grammar.y`**), your work revolves around one task: derive an LALR(1) grammar[1] from the BNF grammar provided as input. Specifically, make sure that you produce a parser from bison that

- Uses at most one token of lookahead
- Is free of shift-reduce conflicts
- Is free of reduce-reduce conflicts.

To test your code, you can simply augment the **`parser.C`** file to call the parser. The output is a single bit of information: the string (input) does or does not belong to the language generated by the grammar. It is your responsibility to thoroughly test all your code by writing sample `C--` programs. We will test your parser on an extensive test suite. Once you are ready to handin, archive the directory again (as a zip file) and proceed to the handin section of the website.

1 Grammar

Your first task is to derive a grammar that is LALR(1) compliant. You are starting from an Extended Backus-Naur Form (EBNF `http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form`) grammar. The challenge is to find and apply the necessary rewrites so that the grammar become LALR(1) and still recognize the same language (`C--`). Figure 1 shows the complete grammar for `C--`.
The symbols in italics are the non-terminals. The symbols in blue, normal font are the terminals and the red bold symbols are the EBNF annotations. EBNF is reminiscent of regular expressions. Specifically

- a fragment **[** α **]** means that the string α is optional.
- a fragment { $\alpha|\beta|\gamma$ } is an alternative and means that we could see either α or β or γ.
- a fragment α^* is a Kleene closure of α, meaning that the string α can appear $0, 1$, or more times.

[1]Recall that LALR is a minor variant of LR that uses lookaheads in a a special way.

```
classList  ::=  classDecl+
classDecl  ::=  class ID [ extends ID ] { decl* } ;
     decl  ::=  type ID ;
              | type ID ([formals]) { statement* }
              | ID ( [formals] ) {statement* }
  formals  ::=  formal[,formal]*
statement  ::=  expr ;
              | type ID ;
              | lvalue = expr ;
              | { statement* }
              | while ( expr ) statement
              | if (expr) statement [else statement]
              | return expr ;
              | ;
   formal  ::=  type ID
     type  ::=  { int | bool | void | ID } [ [] ]

     expr  ::=  expr op expr
              | uop expr
              | NUMBER
              | true
              | false
              | ( expr )
              | lvalue
              | new ID ( [actuals] )
              | new type [expr ]
       op  ::=  &&|||==|!=|<=|>=|<|>|+|-|*|/
      uop  ::=  ! | −
   lvalue  ::=  ID [ ( [actuals] ) ]
              | lvalue [ expr ]
              | lvalue . ID [ ([actuals] ) ]
              | this
  actuals  ::=  expr[,expr]*
```

Figure 1: EBNF Grammar for `C--`.

Together with this grammar, assume the traditional precedence for the arithmetic operators (e.g., * binds more than +). Finally, note the presence of the two terminals `true` and `false` denoting the corresponding boolean values.
Rewrite the grammar to eliminate the EBNF construction and produce an LALR(1) compliant grammar that correctly handles operator precedence. Feel free to use the bison precedence and associativity rules (e.g., `%left` and `%right`) on tokens. Recall that the order in which tokens are specified in the grammar file affects *precedence* whereas the `%left` and `%right` annotations affects *associativity*.

2 Bison

Once you have a sensible LALR(1) grammar (it might still have shift-reduce and reduce-reduce conflicts, but it no longer uses any EBNF notations), it is time to turn to bison. You must use bison to derive a parse function aptly named `yyparse`. `bison` takes as input a `grammar.y` file that contains the required specifications. As you already know, the bison specification provides all the necessary tie-in to the flex scanner. The code bison will produce will simply call `yylex`. The `grammar.y` you received for the lexical analysis homework was still missing a production section (between the two sets of double percent signs) as well as directives to define operator precedence. *You must specify all your grammar productions and resolve all the operator precedence issues.* Once this is done, you can run bison to create your parser. If the grammar is indeed LALR(1), you should end-up with two files `grammar.c` and `grammar.h`. If you have conflicts (e.g., shift-reduce or reduce-reduce) you **must** fix them. There is only one conflict that is expected to remain: it is caused by the "dangling else" issue. If you recall, the *dangling else* is not fixable in languages with an optional else and no "end-if" syntax. Thankfully, this is easy to cope with and bison will deal with that shift-reduce conflict by simply favoring the shift over the reduce which is precisely the right thing to do in this case.

To inform `bison` that you *expect* one shift-reduce conflict and that this conflict should be ignored, you can add a directive in `grammar.y`. The code fragment below shows where the modification should be done. Recall that `%pure-parser` simply precludes the use of global variables in the generated code. The second directive says that the `yyparse` function will have one argument of type `Parser*`. The third directive (which you should have!) states that bison can accept as valid a grammar that has exactly one conflict. Any additional conflicts will prevent code generation.

```
%pure-parser
%parse-param { Parser* parser }
%expect 1  // we can deal with 1 shift-reduce (dangling else)
```

Hint *To ease your task I strongly suggest that you enter your grammar into bison a little bit at a time, resolving the conflicts as you go. For instance, you could start with a grammar with only basic expressions and build up to full expressions. Then add classes and methods and finally add method bodies. If you try it all at once, you will have many conflicts and will be overwhelmed.*

3 Driver

The top-level driver is actually quite trivial and reproduced below for your convenience. A few observations are in order

- The run method defines an external symbol named `yyparse` with the correct type signature. Bison generates `yyparse` of course. The parsing will start once you call `yyparse` (last line of the run method).
- The `yyin` global variable specifies the file that your scanner will read from (we will not keep on reading from the standard input!). If you assign a file handle to `yyin`, the scanner will use it as its source. You can obtain a file handle via the `fopen` C library function.
- If you have strange bugs in your parser, you can set a variable named `yydebug` to true. If you do so, the generated parser will produce debug output when it hits a syntax error. This output will refer to the underlying PDA of course as well as the state of the parse stack at the time of the error. This is a pure runtime debugging feature in case your grammar (while being LALR(1)) does *not* recognize the `C--` language correctly.
- The function `yyerror` is called by `yyparse` when it discovers syntax errors. It is your opportunity to set breakpoints, print the messages, etc....

```
#include "parser.H"
#include <iostream>
#include <iomanip>
Parser::Parser() {}
Parser::~Parser() {}
void Parser::run(const char* fn) {
   extern int yyparse(Parser*);
   extern FILE* yyin;
   if (fn!=0)
      yyin = fopen(fn,"r");
   yyparse(this);
}
int yyerror(Parser* p,const char* s) {std::cerr << s << std::endl;return 0;}
```

Beware: if you copy/paste from the PDF, some symbols may not be ASCII. Make sure that all the symbols come through alright.

Have fun!