CMPS 104A Assignment 3: LALR(1) Parsing with `bison`

Due on Wednesday, November 13, 23:59:59.

1. Overview

Augment your string table and scanner by adding an `oc` parser to the project. The output of this project will be an abstract syntax tree written to the file *program*`.ast`, in addition to the files generated from the previous two projects. All specifications from the first two projects apply, and in addition, the `-y` flag must turn on `yydebug`.

SYNOPSIS

```
oc [-ly] [-@ flag...]  [-D string] program.oc
```

The main function will open a pipe to the C preprocessor as before, but will never call `yylex()`. Instead it makes a single call to `yyparse()`, which will call `yylex()` as needed. When `yyparse()` returns, the main function will call a function to dump the AST. The function `yyparse()` calls `yyerror()` to print an error message whenever it detects a syntax error and has to recover. The `-y` flag must turn on the `yydebug` switch. Generate a file called *program*`.ast`, based on the input file name, and also generate all files specified in earlier projects.

2. The Grammar of `oc`

Table 2 shows the context-free grammar of `oc`. Your task is to translate that descriptive user-grammar into LALR(1) form acceptable to `bison`. You may, of course, take advantage of `bison`'s ability to handle ambiguous grammars via the use of precedence and associativity declarations. The dangling `else` problem should also be handled in that way.

You will not be able to feed the grammar directly to bison, because it is necessary to eliminate the metagrammar's optional and repetitive brackets. In addition, `bison` is not able to handle the *operator* as a single token.

[x]	Brackets indicate that x is optional.
[x]...	Brackets and three dots mean that x occurs zero or more times.
x| y	A bar indicates an alternation between x and y.
symbol	Nonterminal symbols are written in lower case.
'`void`'	Tokens representing themselves are '`quoted`'.
NUMBER	Token classes are written in SMALL CAPITALS.
operator	See Table 3 for special token classes in *italics*.

Table 1: Metagrammar which describes the notation for the `oc` grammar

program	→	[structdef | function | statement]...
structdef	→	'`struct`' IDENT '`{`' [decl '`;`']... '`}`'
decl	→	type IDENT
type	→	basetype ['`[]`']
basetype	→	'`void`' | '`bool`' | '`char`' | '`int`' | '`string`' | IDENT
function	→	type IDENT '`(`' [decl ['`,`' decl]...] '`)`' block
block	→	'`{`' [statement]... '`}`' | '`;`'
statement	→	block | vardecl | while | ifelse | return | expr '`;`'
vardecl	→	type IDENT '`=`' expr '`;`'
while	→	'`while`' '`(`' expr '`)`' statement
ifelse	→	'`if`' '`(`' expr '`)`' statement ['`else`' statement]
return	→	'`return`' [expr] '`;`'
expr	→	binop| unop | allocator | call | '`(`' expr '`)`' | variable | constant
binop	→	expr *operator* expr
unop	→	*operator* expr
allocator	→	'`new`' basetype '`(`' [expr] '`)`' | '`new`' basetype '`[`' expr '`]`'
call	→	IDENT '`(`' [expr ['`,`' expr]...] '`)`'
variable	→	IDENT | expr '`[`' expr '`]`' | expr '`.`' IDENT
constant	→	INTCON | CHARCON | STRINGCON | '`false`' | '`true`' | '`null`'

Table 2: Grammar of `oc`

Precedence	Associativity	Arity	Fixity	Operators
lowest	right	binary/ternary	matchfix	`if else`
	right	binary	infix	`=`
	left	binary	infix	`== != < <= > >=`
.	left	binary	infix	`+ -`
.	left	binary	infix	`* / %`
.	right	unary	prefix	`+ - ! ord chr`
	left	variadic	postfix	`f(...)`
	left	binary	postfix	`e[e] e.i`
	–	unary	prefix	`new`
highest	–	unary	matchfix	`(e)`

Table 3: Operator Predecende in `oc`

You will need to explicitly enumerate all possible rules with operators in them. However, using bison's operator precedence declarations, the number of necessary rules will be reduced. Table 3 shows operator precedence and associativity. There is actually more information there than that which will be useful in `%left` and `%right` declarations.

3. Abstract Syntax Trees

The abstract syntax tree (AST) needs to have a node for every rule given in Table 2. Every node references the child nodes of which it consists. These child nodes may be leaf nodes (identifiers or constants) or other expressions. Constants and identifiers are always leaf nodes. In general, interior nodes may have an arbitrarily large number of children. This is the case wherever the grammar shows "..." indicating repetition.

There are different ways to implement abstract syntax trees. You can use the existing `struct astree` and the functions `adopt1`, `adopt1sym` and `adopt2` from the provided `astree.h` file. Alternatively, you can also create new structs or C++ classes to represent the nodes of the AST.

Be sure to have interior nodes for all the nonterminals in the AST except `expr` and `statement`. These should be replaced be the concrete nodes they represents (`binop`, `return`, etc.).

3.1 The Parser

Start out with the dummy `parser.y` which will generate a header file and a C source file. Develop it incrementally, possibly using routines from the example `expr-smc`, bearing in mind that that code does not exactly fit this project. The parser needs a way to communicate with the main function, but has no communication results or parameters, so use a global variable (e.g. `yyparse_astree`) for that purpose.

All actions in the parser should be simple. Use function calls when that is not the case. In general, actions should have one of the following three forms:

```
a : a b { $$ = $1; $$->add($2); }
  | c   { $$ = fncall (args...); }
  |     { $$ = new node (args...); }
```

3.2 Printing the AST

After constructing an AST from a file called *program*.`oc`, write a file called *program*.`ast`, containing a representation of the AST in text form, printed using a depth-first pre-order traversal, showing depth via indentation. Each line is indented to show its distance from the root in multiples of two spaces and contains information according to Table 4. The exceptions are `expr` and `statement` which should not appear in the output.

Type of Node		Examples

Nonterminal	The name of the nonterminal in the grammar representing this rule	`program`
		`function`
		`binop`

Terminal	The name of the token as determined by `get_yytname`, followed by the lexical information associated with the token in parentheses.	`TOK_INTCON (23)`
		`'+' (+)`
		`TOK_IDENT (foo)`

Table 4: Output format for *program*.`ast`

```
int count = 0;
while (count <= 100) {
   count = count + 1;
   puti (count);
   endl ();
}
```

Listing 1: Example oc program (10-hundred.oc)

```
program
  vardecl
    type
      basetype
        TOK_INT (int)
    TOK_IDENT (count)
    constant
      TOK_INTCON (0)
  while
    binop
      variable
        TOK_IDENT (count)
      TOK_LE (<=)
      constant
        TOK_INTCON (100)
    block
      binop
        variable
          TOK_IDENT (count)
        '=' (=)
        binop
          variable
            TOK_IDENT (count)
          '+' (+)
          constant
            TOK_INTCON (1)
      call
        TOK_IDENT (puti)
        variable
          TOK_IDENT (count)
      call
        TOK_IDENT (endl)
```

Listing 2: Abstract Syntax Tree for the example (10-hundred.ast)

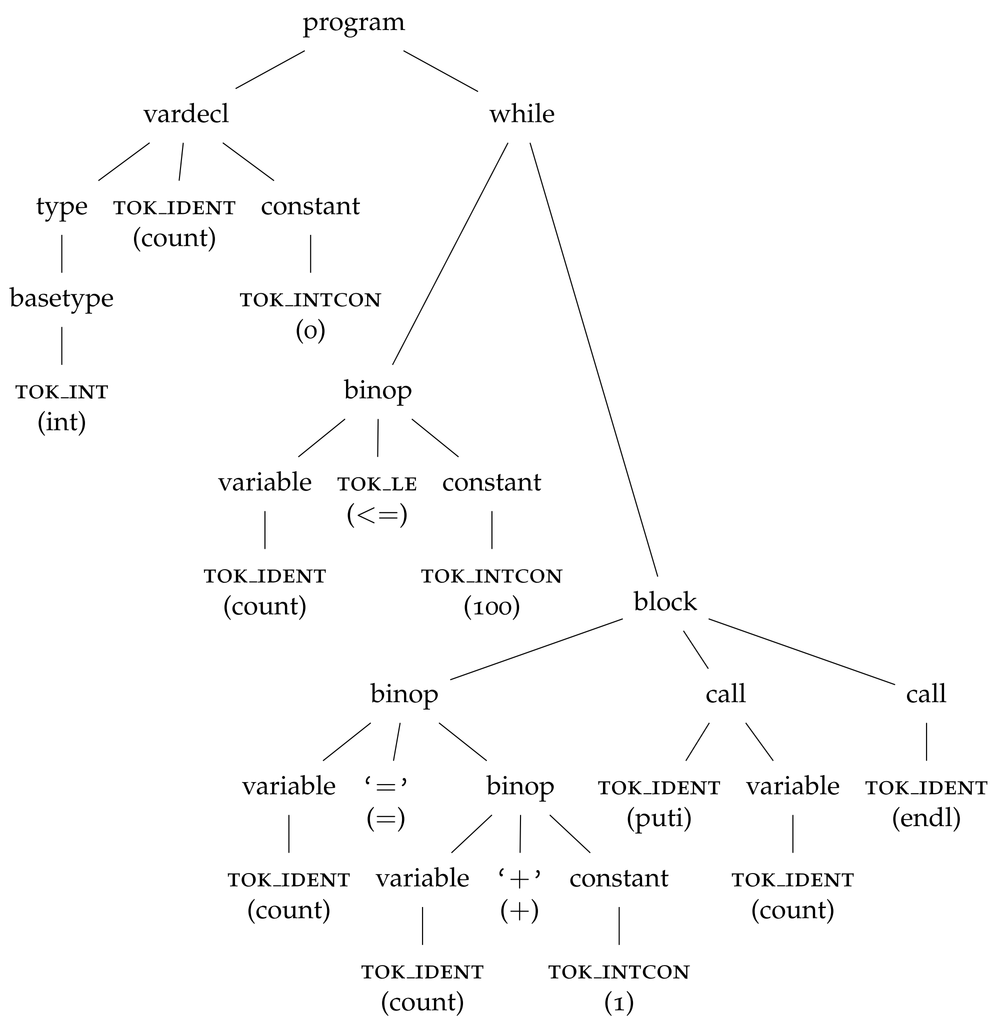

Figure 1: Abstract Syntax Tree for Listing 1

4. Grading

There are 60 points in total.

- (5) Good overall coding style (identifiers, indentation, comments)
- (15) `parser.y` productions according to the `oc` grammar
- (5) Precedence/associativity for operators
- (5) No shift/reduce or reduce/reduce conflicts (−1 pt/conflict)

If your program did not compile, ignore the following...

- (6) Non-zero exit code for any test files with syntax errors (`nosuch.oc`, `90-c8q.oc`, `92-uncomment.oc`, `94-syntax.oc`, `95-cobol.oc`, `96-unterminated.oc`) (−1 pt/error)
- (6) Zero exit code for all other test files (−1 pt/error)
- (1) Test with `-y` option produces token trace
- (1) `.str` files created as per project 1 specs
- (1) `.tok` files created as per project 2 specs
- (8) `.ast` files created with abstract syntax tree
- (2) Nonterminals in the AST shown as name of production in the grammar (`program`, `function`, `binop`)
- (2) Terminals in the AST shown as token symbol name followed by lexical information in parentheses (`TOK_IDENT (foo)`, `TOK_INTCON (23)`, `+ (+)`)
- (1) Nodes of the AST correctly indented
- (2) Nonterminals `expr` and `statement` omitted from AST

5. What to Submit

Submit `README`, `Makefile`, `scanner.l`, `parser.y`, and all of the header and C++ implementation files. Do not submit the file generated by `flex` or `bison`. To submit, log into the Linux Lab, either locally or by ssh, and

```
$ cd /path/to/my/files/
Check that README (and optionally PARTNER) exists
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.build
Check if there are any errors building oc
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.tests
Check if there are any unexpected results
$ submit cmps104a-wm.f13 asg3 README Makefile ... ...
```