Parser Generators

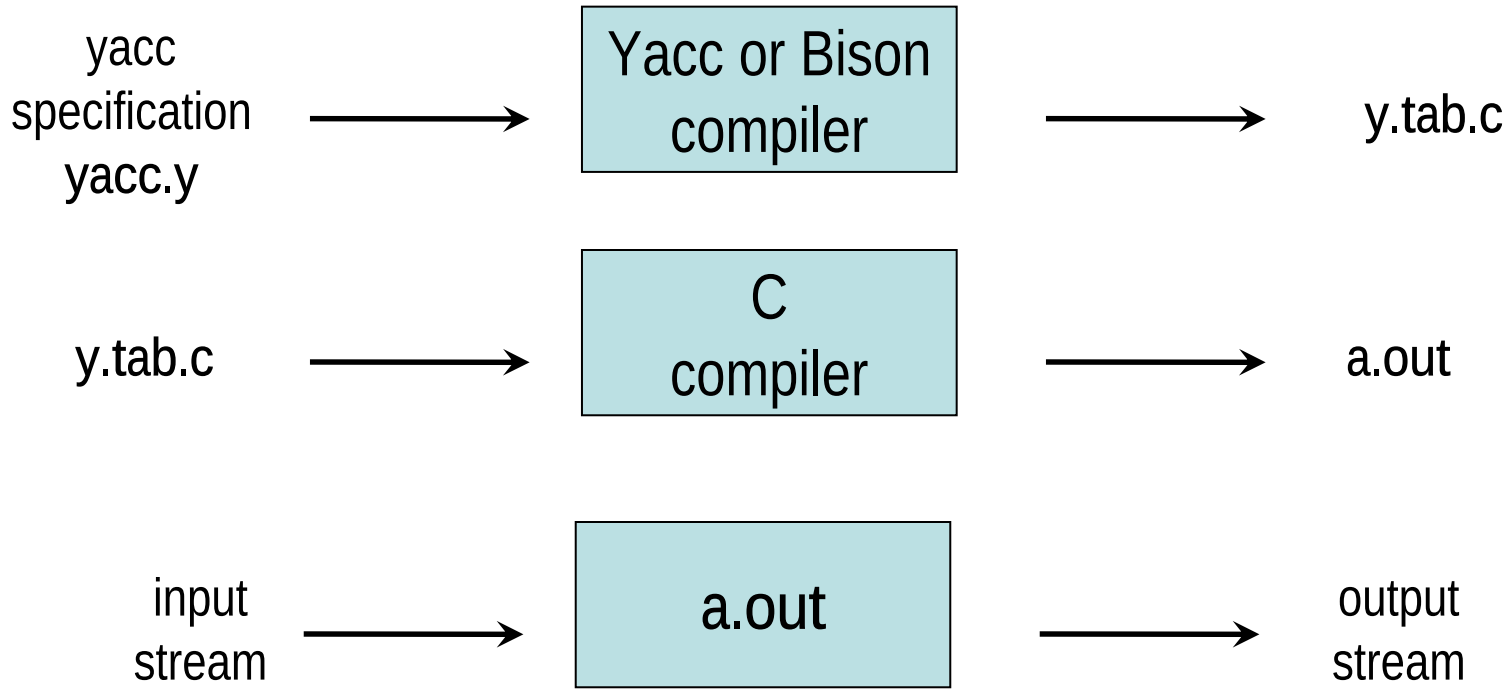
Reading: Section 4.9 of Text Book

Tools to facilitate the construction of front end of the compiler

- *ANTLR* tool
  - – Generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler)
  - – Generates LALR(1) parsers
- *Bison*
  - – Improved version of Yacc

Introduction to Bison

Bison is a general purpose parser generator that converts a description for an *LALR(1) context-free grammar* into a C program file that is bison parse that grammar.



Introduction to Bison

The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions.

The tokens come from a function called the lexical analyzer that must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens.

Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this

The Bison parser file is C code which defines a function named *yyparse* which implements that grammar. This function does not make a complete C program: you must supply some additional functions.

Stages in Writing Bison program

1. Formally specify the grammar in a form recognized by Bison
2. Write a lexical analyzer to process input and pass tokens to the parser.
3. Write a controlling function that calls the Bison-produced parser.
4. Write error-reporting routines.

Bison Specification

- A *bison specification* consists of four parts:

  %{
  C declarations
  %}
  Bison declarations
  %%
  Grammar rules
  %%
  Additional C codes

Productions in Bison are of the form

Nonterminal **:** tokens/nonterminals { *action* }
| tokens/nonterminals { *action* }
…
;

Bison Specification

Bison Declaration

Tokens that are single characters can be used directly within productions, e.g. **'+', '-', '*'**

Named tokens must be declared first in the declaration part using

%token *TokenName (Upper Case Letter)*

e.g %token INTEGER IDENTIFIER

%token NUM 100

- *%left*, *%right* or *%nonassoc* can be used instead for *%token* to specify the precedence & associativity (precedence declaration). All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity.
- *%union* declare the collection data types
- *%type <non-terminal>* declare the type of semantic values of non-terminal
- *%start <non-terminal>* specify the grammar start symbol (by default the start symbol of grammar)

Bison Specification

Grammar Rules

In order for Bison to parse a grammar, it must be described by a *Context-Free Grammar* that is LALR(1).

A non-terminal in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it is in *lower case*, such as *expr*, *declaration.*

A Bison grammar rule has the following general form:

RESULT: COMPONENTS... ;

where RESULT is the nonterminal symbol that this rule describes and COMPONENTS are various terminal and nonterminal symbols that are put together by this rule.

For example, *exp: exp '+' exp ;* says that two groupings of type 'exp', with a '+' token in between, can be combined into a larger grouping of type 'exp'.

Bison Specification

Grammar Rules

Multiple rules for the same RESULT can be written separately or can be joined with the vertical-bar character ‘|’ as follows:

```
RESULT:     RULE1-COMPONENTS...
          | RULE2-COMPONENTS...
          ...
          ;
```

If COMPONENTS in a rule is empty, it means that RESULT can match the empty string.

For example, here is how to define a comma-separated sequence of zero or more ‘exp’ groupings:

```
expseq:     /* empty */
          | expseq1
          ;
expseq1:    exp
          | expseq1 ',' exp
          ;
```

It is customary to write a comment ‘/* empty */’ in each rule with no components.

Bison Specification

Semantic Actions

To make program useful, it must do more than simply parsing the input, i.e, must produce some output based on the input.

Most of the time the action is to compute semantics value of whole construct from the semantic values associated with various tokens and groupings.

For Example, here is a rule that says an expression can be the sum of two sub-expression,

```
expr: expr '+' expr {$$ = $1 + $3;}
    ;
```

The action says how to produce the semantic value of the sum expression from the value of two sub expressions

In bison, the default data type for all semantics is `int`. i.e. The parser stack is implemented as an integer array. It can be overridden by redefining the macro YYSTYPE. A line of the form `#define YYSTYPE double` in the C declarations section of the bison grammar file.

Bison Specification

Semantic Actions

To use multiple types in the parser stack, a “union” has to be declared that enumerates all possible types used in the grammar.

Example:

```
%union{
double val;
char *str;
}
```

This says that there are two alternative types: double and char*.
Tokens are given specific types by a declaration of the form:

```
%token <val> exp
```

Interfacing with Flex

`bison` provides a function called `yyparse()` and *expects* a function called `yylex()` that performs lexical analysis. Usually this is written in lex. If not, then `yylex()` whould be written in the C code area of bison itself.

If `yylex()` is written in flex, then bison should first be called with the -d option: `bison -d grammar.y`

This creates a file `grammar.tab.h` that containes `#defines` of all the `%token` declarations in the grammar. The sequence of invocation is hence:

```
bison -d grammar.y
flex grammar.flex
gcc -o grammar grammar.tab.c lex.yy.c -lfl
```

Programming Example

This is an example for bison with flex,
bison only example can be found on page 262.

```
/* Mini Calculator */
/* calc.flex */

%{

#define YY_NO_UNPUT

using namespace std;

#include <iostream>
#include <stdio.h>
#include <string>

#include "tok.h"
int yyerror(char *s);
int yylineno = 1;
%}

…
```

→

```
…

digit                   [0-9]
int_const  {digit}+

%%

{int_const}     { yylval.int_val = atoi(yytext); return INTEGER_LITERAL; }
"+"             { yylval.op_val = new std::string(yytext); return PLUS; }
"*"             { yylval.op_val = new std::string(yytext); return MULT; }

[ \t]*          {}
[\n]            { yylineno++;     }

.               { std::cerr << "SCANNER "; yyerror(""); exit(1);  }
```

Programming Example

```
/* Mini Calculator */
/* calc.y */

%{
#define YY_NO_UNPUT
using namespace std;
#include <iostream>
#include <stdio.h>
#include <string>
int yyerror(char *s);
int yylex(void);
int yyparse();
%}
%union{
  int           int_val;
  string*       op_val;
}
%start        input
%token <int_val> INTEGER_LITERAL
%type         <int_val>    exp
%left         PLUS
%left         MULT
%%
input:  /* empty */
        | exp{ cout << "Result: " << $1 << endl; }
        ;
exp:          INTEGER_LITERAL    { $$ = $1; }
              | exp PLUS exp          { $$ = $1 + $3; }
               exp MULT exp           { $$ = $1 * $3; }
              ;
%%
int yyerror(string s)
{
  extern int yylineno;     // defined and maintained in lex.c
  extern char *yytext;// defined and maintained in lex.c

  cerr << "ERROR: " << s << " at symbol \"" << yytext;
  cerr << "\" on line " << yylineno << endl;
  exit(1);
}

int yyerror(char *s)
{
  return yyerror(string(s));
}

int main(int argc, char **argv)
{
  if ((argc > 1) && (freopen(argv[1], "r", stdin) == NULL))
  {
    cerr << argv[0] << ": File " << argv[1] << " cannot be
opened.\n";
    exit( 1 );
  }

  yyparse();

  return 0;
}
```

Assignments

Q.1. Design a grammar for a logic language that depicts a “condition”. The language has two atoms **t** for true and **f** for false. The output of any program should be either **t** or **f**. The language can contain the above atoms or comparison statements like (< 2 3) which evaluates to **t** or combinations of logic statements like (& (> 4 2) (< 3 6)). Comparison operators are: <, >, and =; logic operators are & (logical and), | (logical or) and ~ (logical not).

Compile the above grammar using bison and print the result for any logical expression. Did your grammar have any conflicts?

Q.2. Q. 4.50, 4.51 and 4.52 of text book.