Bison and parsing

From the area of compilers, we get a host of tools to convert text files into programs. After lexical analysis, the second part of that process when you are dealing with traditional languages such as C is syntax analysis, which also known as parsing.

A good tool for creating parsers is `bison`. It takes a specification file and creates an syntax analyzer, previously called `y.tab.c` by `yacc` and now is generally just `FILENAME.tab.c`.

Parsing terms

- Production rules define a parser. Informally, these can be expressed in BNF/EBNF form.
- Production rules are made up a left hand side with a non-terminal, and righthand side made up terminals and non-terminals.
- A terminal "represents a class of syntactically equivalent tokens" [Bison manual].

Attributes for terminals and non-terminals

Terminals and non-terminals can have attributes.

Constants could have the value of the constant, for instance.

Identifiers might have a pointer to a location where information is kept about the identifier.

Some general approaches to syntax analysis

Use a compiler-compiler tool, such as `bison`.

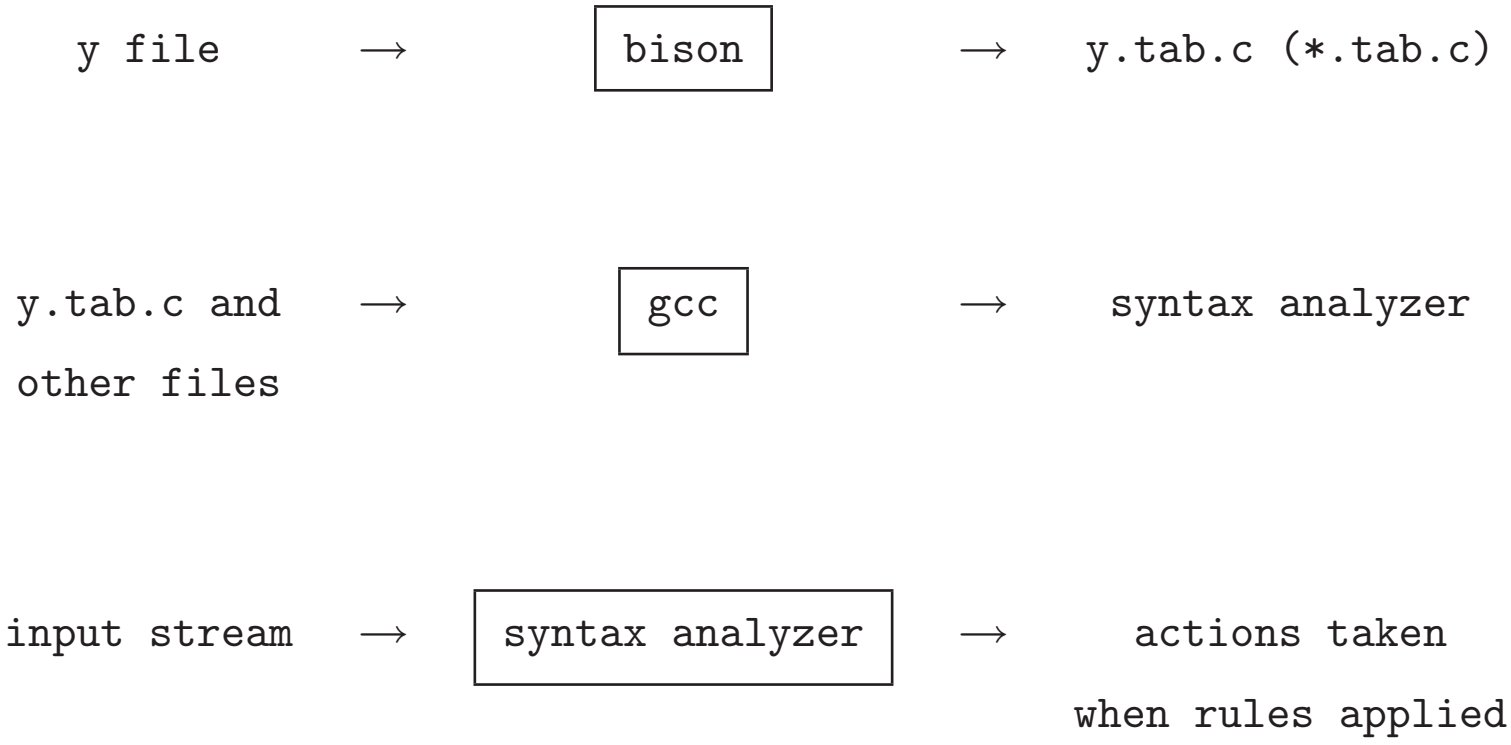
Write a one-off recursive descent parser.

Write a one-off parser suited to your program.

Bison - our lexical analyzer generator

Can be called as `yyparse()`.

It is easy to interface with `flex/lex`.


y file
bison
y.tab.c (*.tab.c)
y.tab.c and
other files
gcc
syntax analyzer
input stream
syntax analyzer
actions taken
when rules applied

Calling Bison

Here's an example of calling Bison (which will be very useful when compiling `assign6`):

```
Assign6-solution.out: Assign6-solution.y Assign6-solution.l
        bison -d --debug --verbose Assign6-solution.y
        flex Assign6-solution.l
        cc -c lex.yy.c
        cc -c Assign6-solution.tab.c
        cc -o Assign6-solution.out Assign6-solution.tab.o lex.yy.o
```

The `-d` option specifies to output an explicit

`y.tab.h/*.tab.h` file for flex. Specifying `--debug` and `--verbose` (combined with enabling `yydebug`) make it much easier to debug your parser!

Bison specifications

Bison source:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Definitions

- Declarations of ordinary C variables and constants.
- `bison` declarations.

Rules

The general form for production rules is:

```
<non-terminal> :  <sequence of terminals and non-terminals> {action} | ... ;
```

The actions are C/C++ code. Actions can appear in the middle of the sequence of terminals and non-termianls.

Bison declarations

```
%token TOKEN        create a TOKEN type

%union { }          create a Union for llvals.

%right TOKEN        create a TOKEN type that has right associativity

%left TOKEN         create a TOKEN type that has left associativity
```

Bison actions

Actions are C source fragments.

Example rules:

```
variableDeclaration : ID COLON ID SEMICOLON {
                                        printf("emitting var %s of type %s\n",$3,$1
                                            } ;
```

The $3 and $1 refer to the values of the items 3 and 1 in the righthand side of the production rule.

An example of Bison: first, its matching flex file

```
%{
#include <stdlib.h>
#include <string.h>
#include "Assign6-solution.tab.h"
extern int linecount;
%}
%%
program             return PROGRAM;
end                 return END;
variables           return VARIABLES;
var                 return VAR;
functions           return FUNCTIONS;
define              return DEFINE;
```

```
statements          return STATEMENTS;
if                  return IF;
then                return THEN;
else                return ELSE;
while               return WHILE;
,                   return COMMA;
"("                 return LPARENTHESIS;
")"                 return RPARENTHESIS;
"{"                 return LBRACE;
"}"                 return RBRACE;
:                   return COLON;
;                   return SEMICOLON;
[a-zA-Z0-9]+        yylval = (int)strdup(yytext); return ID;
[\n]                linecount++;
[ \t]+
```

An example Bison program

```
%{
#include <stdlib.h>
#include <stdio.h>
int linecount = 0;
void yyerror(char *s)
{
  fprintf(stderr,"file is not okay -- problem at line %d\n",linecount);
  exit(1);
}
int yywrap()
{
  return 1;
}
%}
%token ID
```

```
%token PROGRAM
%token END
%token VARIABLES
%token VAR
%token STATEMENTS
%token IF
%token THEN
%token ELSE
%token WHILE
%token LBRACE
%token RBRACE
%token COLON
%token SEMICOLON
%token FUNCTIONS
%token COMMA
%token DEFINE
%token LPARENTHESIS
%token RPARENTHESIS
%%
program : PROGRAM ID variablesSection functionsSection statementsSection END ;
variablesSection : VARIABLES LBRACE variableDeclarations RBRACE ;
```

```
variableDeclarations :  | variableDeclarations variableDeclaration ;
variableDeclaration : ID COLON ID SEMICOLON {printf("emitting var %s of type %s\n",$
functionsSection : FUNCTIONS LBRACE functionDeclarations RBRACE ;
functionDeclarations : | functionDeclarations functionDeclaration ;
functionDeclaration : DEFINE ID COLON ID LPARENTHESIS argsList RPARENTHESIS LBRACE $
statementsSection : STATEMENTS LBRACE statements RBRACE ;
statements : | statements statement ;
statement : VAR variableDeclaration | whileLoop | ifStruct | subroutineCall SEMICOL
whileLoop : WHILE LPARENTHESIS subroutineCall RPARENTHESIS LBRACE statements RBRACE
ifStruct : IF LPARENTHESIS subroutineCall RPARENTHESIS LBRACE statements RBRACE  ;
           |
           IF LPARENTHESIS subroutineCall RPARENTHESIS LBRACE statements RBRACE ELSE
subroutineCall : ID LPARENTHESIS callArgsList RPARENTHESIS ;
argsList : | argPair | argsList COMMA argPair ;
argPair : ID ID ;
callArgsList : | ID |  callArgsList COMMA ID ;
%%
int main(int argc, char **argv)
{
  //  yydebug = 1;
  yyparse();
```

```
  printf("input is okay\n");
}
```