YACC

Introduction

- Yacc
  - **Y**et **A**nother **C**ompiler **C**ompiler.

- YACC Installation
  - **Ubuntu** – sudo apt-get install bison
  - **Fedora** – yum install bison
  - **Windows** – http://downloads.sourceforge.net/gnuwin32/bison-2.4.1-setup.exe

Introduction

- Yacc
  - **Y**et **A**nother **C**ompiler **C**ompiler.

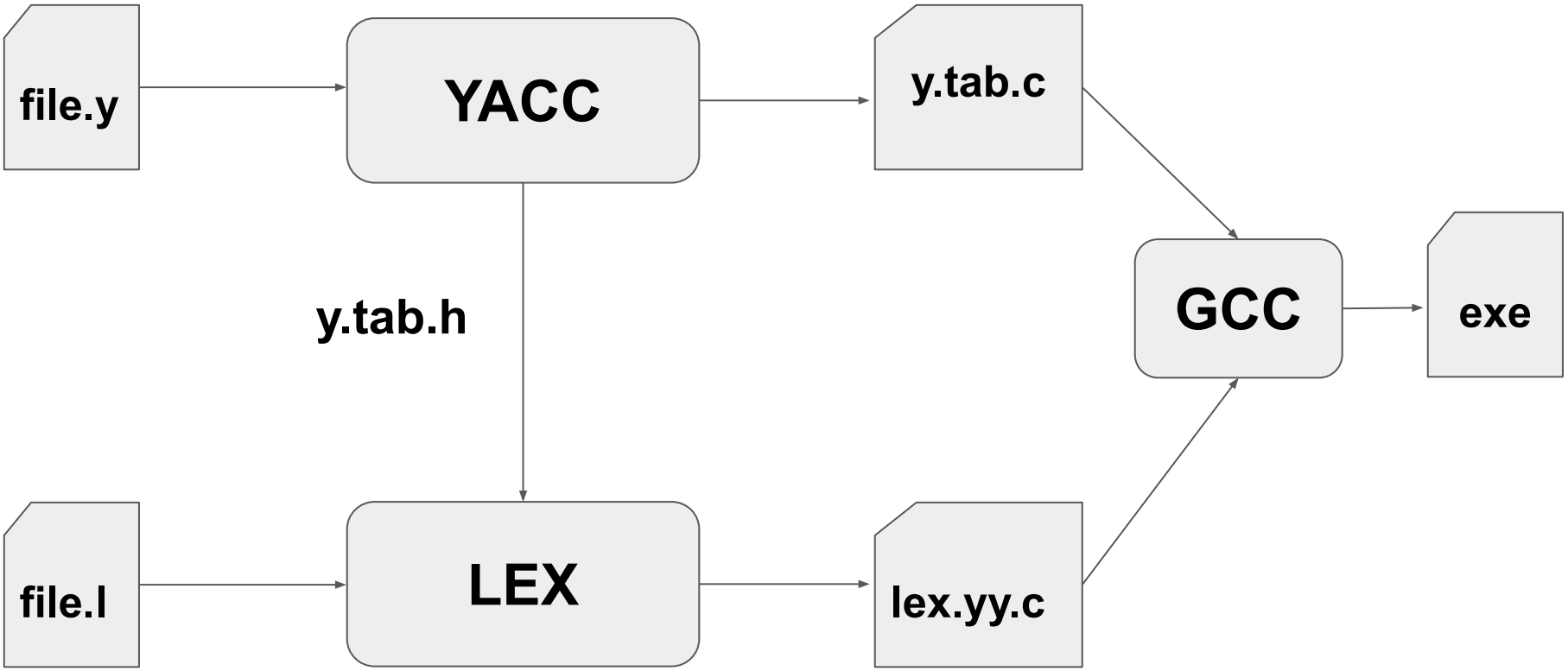
- YACC generates
  - Tables – according to the grammar rules.
  - Driver routines – in C programming language.
  - y.output – a report file.

Introduction

Input: A Grammar

Output: A parser for the grammar

Introduction



Introduction

- Invokes yylex() automatically.
- Generate *y.tab.h* file through the -d option.
- The lex input file must contains y.tab.h
- For each token that lex recognized, a number is returned (from *yylex()* function.)

Introduction

- Steps
  - yacc -d file.y # create y.tab.h, y.tab.c
  - lex file.l # create lex.yy.c
  - gcc lex.yy.c y.tab.c -o exe # execute file
  - ./exe

YACC File

- A Yacc input file consists of three sections:
  - Definition
  - Rules
  - User code
- Separate by %%
- Similar to lex (actually, lex imitates yacc.)

YACC File

%{

C declarations

%}

yacc declarations

%%

Grammar rules

%%

Additional C code

Rules Section

S -> n = E
S -> E

S: n = E | E;

- Each rule contains LHS and RHS, separated by a colon and end by a semicolon.
- White spaces or tabs are allowed.
- Ex:

```
statement: name EUQALSIGN expression
        | expression ;

expression: number PLUSSIGN number
        | number MINUSSIGN number ;
```

Semantic Routines

- The action in semantic routines are executed for the production rule.
- The action is actually C source code.
- LHS: `$$` RHS: `$1 $2 ……`
- Default action: { `$$ = $1;` }
- Action between a rule is allowed. For ex:

```
expression : simple_expression
  | simple_expression {somefunc($1);} relop simple_expression;
```

YACC File

```
%{
#include <stdio.h>
%}
```

```
%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression    { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
                  |    expression '-' NUMBER { $$ = $1 - $3; }
                           |     NUMBER{ $$ = $1; }
        ;
%%
```

```
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

int main(void)
{
   yyparse();
   return 0;
}
```

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

Start Symbol

- The first non-terminal specified in the grammar specification section.
- To overwrite it with %start declaraction.

```
%startnon-terminal
```

Rules Section

- This section defines grammar
- Example

```
expr : expr '+' term | term;  term : term '*' factor
| factor;  factor : '(' expr ')' | ID | NUM;
```

Rules Section

Normally written like this

Example:

```
expr    : expr '+' term
        | term
        ;
term    : term '*' factor
        | factor
        ;
factor  : '(' expr ')'
        | ID
        | NUM
        ;
```

The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

The Position of Rules

```
expr : expr '+' term      { $$ = $1 + $3; }
     | term               { $$ = $1; }
     ;
term : term '*' factor    { $$ = $1 * $3; }
     | factor             { $$ = $1; }
     ;
factor : '(' expr ')'     { $$ = $2; }
       | ID

       | NUM
       ;
```

The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
                         { $$ = $2; }

factor : '(' expr ')'
       | ID
       | NUM
       ;
```

Example grammar:

```
expr -> ( expr )
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | - expr
      | INT
      ;
```

The yacc code:

```
expr : '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | - expr
     | INT
     ;
```

Example

```
%left '+', '-'
%left '*', '/'
%left UMINUS
%%

expr : '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr          %prec UMINUS
     | INT
     ;
```

Actions

```
%left '+', '-'
%left '*', '/'
%left UMINUS
%%

expr : '(' expr ')'          {$$ = $2;}
     | expr '+' expr         {$$ = $1 + $3;}
     | expr '-' expr         {$$ = $1 - $3;}
     | expr '*' expr         {$$ = $1 * $3;}
     | expr '/' expr         {$$ = $1 / $3;}
     | '-' expr        %prec UMINUS    {$$ = -$2;}
     | INT                    {$$ = $1;}
     ;
```

YACC Declaration

`%start'

Specify the grammar's start symbol

`%union'

Declare the collection of data types that semantic values may have

`%token'

Declare a terminal symbol (token type name) with no precedence or associativity specified

`%type'

Declare the type of semantic values for a nonterminal symbol

YACC Declaration

`%right'

Declare a terminal symbol (token type name) that is right-associative

`%left'

Declare a terminal symbol (token type name) that is left-associative

`%nonassoc'

Declare a terminal symbol (token type name) that is nonassociative
(using it in a way that would be associative is a syntax error,
ex: x *op*. y *op*. z is syntax error)

Resources

- See course website.