Introduction to YACC

Some slides borrowed from Louden
YACC

- Yet Another Compiler Compiler
- Written by Steve Johnson at Bell Labs (1975)
  - Bison: Gnu version by Corbett and Stallman (1985)
- Takes a grammar and produces a parser
- Applies tokens from lex to the grammar
  - Determines if these tokens are syntactically correct according to the grammar.
  - Semantics not done with grammar
- It creates LALR(1) parsers
- It produces a shift-reduce parser
  - Parse stack contains a state and a single value accessible in grammar through $vars
YACC

- Similar format to lex

... definitions ...
%%%
... rules ...
%%%
... user code ...

A YACC grammar is constructed of symbols
- Symbols are strings of letters, digits, periods, and underscores that do not start with a digit
- error is reserved for error recovery (only 1)

Lexer produces terminal symbols (tokens)
Non-terminals are the LHS of rules
Tokens can also be string literals "
By convention, terminals are all caps and non-terminals are lowercase
In the definition section you'll need to declare your tokens.

Use the `%token` directive

```yacc
%token PROGRAM_TOK
%token BEGIN_TOK
%token END FOR WHILE COMMA
```

These tokens will be written to `y.tab.h`

`yacc -d` will write the `#defines`

replace print “510” with return END

Don't forget to `#include “y.tab.h”` in `.l`
YACC Rules

- Rules are of the form: \texttt{LHS: RHS;}
  - Notice you replace \texttt{\Rightarrow} with:
- May have multiple rules with same LHS
- \texttt{terminals}: symbols returned by the lexer
  - Convention is \texttt{UPPER\_CASE} (since \texttt{#define} in C)
- \texttt{non-terminals}: symbols on the LHS
  - Convention is lower case, since terminals upper.
- RHS can be empty
- Should end in \\texttt{;}\texttt{,} but don't have to

Example:

\begin{verbatim}
statement : NAME '=' expression;
expression : NUMBER PLUS NUMBER
            | NUMBER '→' NUMBER;
\end{verbatim}
**YACC Rules - Actions**

- Actions-C compound statement executed when a grammar rule is matched.
  - Actions are where the semantic processing goes.

```c
goto: GOTO lab SEMI {printf("Valid goto\n");};
```

- The action can refer to values associated with the symbols.
  - The parse stack contains 1 'value' per symbol
  - $\#$, where $\#$ is order of the symbols
    - For the rule `a: b c d e;` $1 -> b, $2 -> c $4 -> e ...`
- Default action is `{$$ = $1;}`
- Note: Can also use $0, $-1, $-2 to get to other information on the parse stack.
Actions

- Actions occur at the end of the rule, if you put them elsewhere yacc will create fake rules.

```c
foo: A {printf("found A\n");} B;

foo: A fakerule B;
fakerule: /* empty */{printf("found A\n");};

- Avoid this feature, conflicts plus:
  - $1 -> A $2 -> fakerule $3 -> B
```
Recursive Rules

(expression : NUMBER
   | expression ' +' NUMBER
   | expression ' −' NUMBER;

foo : foo bar | bar | ;

- Rules can be recursive
- Rules can be empty
- Rules should end in ; but don't have to
Rules

expression : NUMBER
| expression ' +' NUMBER
| expression ' − ' NUMBER;

expression: NUMBER;
expression: expression ' +' NUMBER;
expression: expression ' − ' NUMBER;

- These are equivalent
Recursive Rules

exprlist: expr | exprlist ',' expr; /* left */
exprlist: expr | expr ',' exprlist; /* right */

- How do these differ?
- Let's expand the following

e1, e2, e3, e4, e5, e6, e7
Recursive Rules

exprlist: expr | exprlist ',' expr; /* left */

L -> exprlist
E -> expr

e1,e2,e3,e4,e5,e6,e7
E , e1
L ,
L, E
L,E e2
L
L,E e3
L
Recursive Rules

exprlist: expr | expr ' ,' exprlist; /* right */
L -> exprlist  E -> expr

e1, e2, e3, e4, e5, e6, e7
E
E,
E,E
E,E,
....
E,E,E,E,E,E,E
E,E,E,E,E,E,L
E,E,E,E,E,L
E,E,E,E,L
Recursion

- Left recursive is more efficient
  - Most rules should be left recursive
- Right recursive can be useful
  - Good for making linked lists

```
thinglist: THING {$$ = $1;}
  | THING thinglist {$1->next = $2; $$ = $1;}
  - For small lists, this is OK
  - For large lists, like statements, it is bad
```
Grammars

- All grammars have a start symbol
  - First nonterminal in rules section
  - %start

- As input is turned into tokens, the tokens are applied to the grammar.
## Grammars

### a: B C D E

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCDE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDE</td>
<td>B</td>
<td>shift</td>
</tr>
<tr>
<td>DE</td>
<td>BC</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>BCD</td>
<td>shift</td>
</tr>
<tr>
<td></td>
<td>BCDE</td>
<td>shift</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>reduce</td>
</tr>
</tbody>
</table>
Grammars

a: B b
b: C D E

<table>
<thead>
<tr>
<th>input</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCDE</td>
<td></td>
</tr>
<tr>
<td>CDE</td>
<td>B</td>
</tr>
<tr>
<td>DE</td>
<td>BC</td>
</tr>
<tr>
<td>E</td>
<td>BCD</td>
</tr>
<tr>
<td></td>
<td>BCDE</td>
</tr>
<tr>
<td></td>
<td>Bb</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>


Compiling

yacc -d part3.y       # make y.tab.h y.tab.c
lex part3.l           # make lex.yy.c
cc -o part3 y.tab.c lex.yy.c -ly -ll # compile
./part3 < test.sil
Errors

- When an error occurs `yyerror()` is called
  - Default `yyerror()` is
    ```c
    yyerror(const char *msg) {
        printf("%s\n", msg);
    }
    
    You may want to redefine it to give more information such as:
    ```
    ```c
    yyerror(const char *s) {
        printf("%d: %s at '%s'\n", yylineno, s, yytext);
    }
    ```
  - You may have to define and/or set `yylineno`
    - Maybe a rule for \n in `lex`?
Error state

- Only one reserved symbol, `error`.
- This is a special symbol that can be used for error recovery.
- For instance

```plaintext
while: WHILE cond statements END WHILE SEMI
    | WHILE error SEMI {printf("Invalid While\n");};
```

- Placement of error token is difficult to get right, try putting it before a statement terminal, i.e. `;`
Error Recovery in Yacc

- Yacc uses a form of error productions
  - \[ A \rightarrow \text{error} \alpha \]
  - `%%`
    - \[ \text{line : lines expr 'n'} \]
      - \[ \mid \text{lines 'n'} \]
      - \[ \mid /* \text{empty} */ \]
      - \[ \mid \text{error 'n'} \]
    - `{printf("%g
", $2); }`  
    - `{yyerror("reenter previous line:’");`  
    - `yyerrok; }`  
    - `;`
  - `yyerrok`: resets the parser to normal mode of operation
Passing Information

D [0-9]

yylval.ival = atoi(yytext); return I_CONST;

yylval.fval = atof(yytext); return F_CONST;
Passing Information

%union{
    float fval;
    int  ival;
}
%token  <ival> I_CONST
%token  <fval> F_CONST
%%
expr: I_CONST {printf("c:%d\n", $1);}
    | F_CONST {printf("c:%f\n", $1);}  
    ;

- Will use correct type by default
Passing Information

```c
%union{
  float fval;
  int  ival;
}
%token  I_CONST
%token  F_CONST

expr: I_CONST {printf("c:%d\n", $1.ival);}
    | F_CONST  {printf("c:%f\n", $1.fval);}
    ;
```

- Less effort setting up the types
- Explicit typing may make actions easier to read
Passing Information

%union{
    float fval;
    int  ival;
}
%token   I_CONST
%token  F_CONST
%
expr: I_CONST {printf("c:%d\n", <$ival>1);}
    | F_CONST  {printf("c:%f\n", <$fval>1);}
    ;

- Use this form if you need/want to override a default type
Symbol Types

- Symbols can have types
- Use %union to declare all possible types
- Can give tokens type using %token
  - Also using %left, %right, and %nonassoc
- Can give non-terminals type using %type
- Once a symbol is given a type, the $ vars use the correct field in the %union
  - You can override this: $<dval>1
The union is declared as YYSTYPE
And yylval is declared with that type
You can enter the symbol table information either in the parser or the scanner.

If you use the scanner you must pass a pointer to the symbol table entry to the parser.

If you use the parser you must pass the identifier string or use yytext in .y

- Remember that yytext may change
- May need to store own copy, strdup()
Ambiguity

expr: expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')' | NUMBER
;

- How should 2+3*4 be parsed?
Ambiguity

For this example E is short for expr

2  shift  NUMBER
E  reduce  E  ->  NUMBER
E+  shift  +
E+3  shift  NUMBER
E+E  reduce  E  ->  NUMBER

- Now what?
- Parser sees '*', so it could reduce 2+3 using expr->expr '+'
  expr or shift '*' expecting to reduce expr '*' expr later on:
- A shift/reduce conflict
Precedence & Associativity

%left '+' '-'
%left '*' '/'

- Here '*' and '/' have higher precedence since they come after '+' and '-'. And '+' and '-' have the same precedence.
- Also have %right and %nonassoc
- Rules get precedence of rightmost on right hand side.
Definitions Review

- Use `%token` to define your terminals, `yacc -d` will create `y.tab.h` and define the token for you (as `#define`).

- Along with the token, you can have exactly one piece of information passed onto the stack. That piece of information can change depending upon the token (or rule matched). Use `%union` to define the possible values. This is defined as `YYTYPE`.

- Remember that one piece of information can be a point to a structure that holds lots of information.

- Can give non-terminals type using `%type`.

- Define the start symbol with `%start`, will default to the first rule (lhs).

- To define precedence you `%left`, `%right`, or `%nonassoc`. 
Conflicts

- Conflicts are caused when yacc has more than one choice for matching a rule
- Usually caused by a bad grammar
- Possibly because of YACC's 1 lookahead
- Sometimes by bad language design
Reduce/Reduce Conflicts

start: a Y
   | b Y;

a:  X;
b:  X;

- Input XY what rule should fire?
  - start:a Y or start:b Y
Reduce/Reduce Conflicts

start: a Z
    | b Z;

a: X y;
b: X y;
y: Y;

- Input XYZ what rules should fire?
- The Y gets reduced to a y
- But the y can complete either a or b
Reduce/Reduce Conflicts

start: A B x Z
  y Z;
x: C;
y: A B C;

- Input ABCZ - what rules should fire?
- On the C should rule x or y get reduced?
Shift/Reduce Conflicts

start: x
    | y R;

x: A R;
y: A;

- Input AR - what rules should fire?
- For the R should y be reduced, or should we shift to the end of x?
Shift/Reduce Conflicts

expr: TERMINAL
    | expr '-' expr;

given expr – expr – expr

- How should this be grouped
  (expr – expr) – expr
  expr – (expr – expr)
Shift/Reduce Conflicts

stmt: IF cond stmt
    | IF cond stmt ELSE stmt
    | TERMINAL;

given IF cond IF cond stmt ELSE stmt

- How should this be grouped
  IF cond (IF cond stmt ELSE stmt)
  IF cond (IF cond stmt) ELSE stmt
- This is called the dangling else
Fixing Conflicts

- Redesign grammar
- Redesign language
- Give precedence
Fixing Conflicts

stmt: matched
  | unmatched
  ;
matched: other_stmt
  | IF expr THEN matched ELSE matched
  ;
unmatched: IF expr THEN stmt
  | IF expr THEN matched ELSE unmatched
  ;
Fixing Conflicts

%%%%

stmt: IF expr stmt  %prec LOWER_THAN_ELSE
    | IF expr stmt ELSE stmt
    ;
Fixing Conflicts

- Conflicts due to limited lookahead
- We can flatten the rules

```
rule: cmd opt_kw '地看着 plist )'

; opt_kw: /* empty */
  | 'phans keyword ')' ;

rule: cmd 'phans keyword ')' 'phans plist ')' |
  | cmd 'phans plist ')' ;
```
Yacc etc.

- Start rule is the first rule in grammar
  - Use `%start` to change it
- Periods are allowed in symbols
  - Don't use them for token names
- Can have multiple grammars:
  - `%token ASTRT PPSTART`
  - combined: `ASTRT agrmr | BSTRT bgrmr`
  - Make sure lexer sends correct start token
- Macro `YYERROR` will call `yyerror()`
- `Yacc -v` will produce `y.output`
  - Good for finding/removing conflicts
Yacc etc.

- For debugging – slow but useful
  - To add debugging code use -t flag or
    \%
    \%
    \#define YYDEBUG 1
    \%
  - Then set yydebug to nonzero to start
- Complete examples in the yacc links
An example

```%{
#include <ctype.h>
%
%token DIGIT

line : expr \n    { printf("%d\n", $1); }
    ;
expr : expr \+ term
    { $$=$1+$3; }
    | term
    ;
term : term \* factor
    { $$=$1*$3; }
    | factor
    ;
factor :'(' expr ')'
    { $$ = $2; }
    | DIGIT
    ;
}%

yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```
Another Example

{%
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%
%token NUMBER
%left '+’ ‘-'
%left ‘*’ ‘/
%right UMINUS
%
line : lines expr ‘\n’ {printf("%g\n", $2);}
    | lines ‘\n’
    | /* empty */
    |
expr : expr ‘+’ expr { $$=1+$3;}
    | expr ‘-’ expr { $$=1-$3;}
    | expr ‘*’ expr { $$=1*$3;}
    | expr ‘/’ expr { $$=1/$3;}
    | ‘(‘ expr ‘)’ { $$ = $2;}
    | ‘-’ expr %prec UMINUS { $$ = -$2;}
    | NUMBER
%
}

%%
yylex() {
    int c;
    while ((c = getchar()) == ‘ ‘);
    if ((c==‘.’) || (isdigit(c))) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}


Yacc insists on defining tokens itself (except single chars can be matched directly).

Actions can use a “value” stack to compute results (yylval); number is position.

The value of a token must be assigned to yylval by the scanner.
Yacc Example, continued

```c
main()
{  return yyparse();  }
int yylex(void)
{  int c;  while((c = getchar()) == ' ');
   if ( isdigit(c) ) {
      ungetc(c,stdin);  scanf("%d",&yylval);
      return(NUMBER);
   }
   if (c == '\n') return 0;/* makes the parse stop */
   return(c);
}
void yyerror(char * s) /* prints an error message */
{  fprintf(stderr,"%s\n",s);}
```
Interfacing Yacc/Bison

- Yacc generates a C file named y.tab.c (Bison: `<filename>.tab.c`)
- Yacc/Bison will generate a header file with token information for a scanner with the `-d` option: `bison -d tiny.y produces tiny.tab.c and tiny.tab.h`
- The `.tab.h` file for the above grammar looks as follows:

  ```c
  #ifndef YYSTYPE
  #define YYSTYPE int
  #endif
  #define NUMBER 258
  extern YYSTYPE yylval;
  ```
With the -v option ("verbose") Yacc generates a file `y.output` (Bison: `<filename>.output`) describing its parsing actions. For example, for the grammar

\[
S \rightarrow A \ B \\
A \rightarrow x \\
B \rightarrow x
\]

the output file looks as on the next slide.
y.output file:

state 0
'x' shift, and go to state 1
S go to state 5
A go to state 2
state 1
A -> 'x'. (rule 2)
$default reduce using rule 2 (A)
state 2
S -> A . B (rule 1)
'x' shift, and go to state 3
B go to state 4

state 3
B -> 'x'. (rule 3)
$default reduce using rule 3 (B)
state 4
S -> A B . (rule 1)
$default reduce using rule 1 (S)
state 5
$ go to state 6
state 6
$ go to state 7
state 7
$default accept
#ifndef _SCAN_H_
define _SCAN_H_

/* MAXTOKENLEN is the maximum size of a token */
define MAXTOKENLEN 40

/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];

/* function getToken returns the
 * next token in source file
 */
TokenType getToken(void);

#endif
#ifndef YYPARSER
/* the name of the following file may change */
#include "tiny.tab.h"
/* ENDFILE is implicitly defined by Yacc/Bison,
 * and not included in the tab.h file
 */
#define ENDFILE 0
#endif

/* Yacc/Bison generates its own integer values
 * for tokens
 */
typedef int TokenType;
```c
#ifndef BISON_TINY_TAB_H
#define BISON_TINY_TAB_H
#ifndef YYSTYPE
#define YYSTYPE int
#define YYSTYPE_IS_TRIVIAL 1
#endif
#define IF 257
#define THEN 258
#define ELSE 259
#define END 260
...
#define RPAREN 275
#define SEMI 276
#define ERROR 277
extern YYSTYPE yylval;
#endif /* not BISON_TINY_TAB_H */
```
%{
#define YYPARSER /* distinguishes Yacc output
   from other code files */

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

#define YYSTYPE TreeNode *
static char * savedName; /* for use in assignments */
static int savedLineNo; /* ditto */
static TreeNode * savedTree; /* stores syntax tree
   for later return */

}%
tiny.y (part 2)

%token IF THEN ELSE END REPEAT UNTIL READ WRITE
%token ID NUM
%token ASSIGN EQ LT PLUS MINUS TIMES OVER
%token LPAREN RPARENS SEMI ERROR
%
/* Grammar for TINY */
program    : stmt_seq { savedTree = $1;} 
            ;
 stmt_seq   : stmt_seq SEMI stmt
            { YYSTYPE t = $1;
              if (t != NULL)
              { while (t->sibling != NULL) t = t->sibling;
                t->sibling = $3;
                $$ = $1; }
              else $$ = $3;
            }
            | stmt   { $$$ = $1; }
            ;
stmt : if_stmt { $$ = $1; } 
  | repeat_stmt { $$ = $1; } 
  | assign_stmt { $$ = $1; } 
  | read_stmt { $$ = $1; } 
  | write_stmt { $$ = $1; } 
  | error { $$ = NULL; } 

if_stmt : IF exp THEN stmt_seq END 
    { $$ = newStmtNode(IfK); 
    $$->child[0] = $2;  
    $$->child[1] = $4; } 
  | IF exp THEN stmt_seq ELSE stmt_seq END 
    { $$ = newStmtNode(IfK); 
    $$->child[0] = $2;  
    $$->child[1] = $4;  
    $$->child[2] = $6; }
assign_stmt : ID
    { savedName = copyString(tokenString);
      savedLineNo = lineno;
    }
ASSIGN exp
    { $$ = newStmtNode(AssignK);
      $$->child[0] = $4;
      $$->attr.name = savedName;
      $$->lineno = savedLineNo;
    }
;
...

factor : ... | NUM
    { $$ = newExpNode(ConstK);
      $$->attr.val = atoi(tokenString);
    } ... /* also an error production */
%%

int yyerror(char * message)
{ fprintf(listing,"Syntax error at line %d: %s\n",
     lineno, message);
  fprintf(listing,"Current token: ");
  printToken(yychar, tokenString);
  Error = TRUE;
  return 0;
}

int yylex(void)
{ return getToken(); }

TreeNode * parse(void)
{ yyparse();
  return savedTree;
}
### Yacc/Bison internal names

<table>
<thead>
<tr>
<th>Yacc internal name</th>
<th>Meaning/Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>y.tab.c</td>
<td>Yacc output file name</td>
</tr>
<tr>
<td>y.tab.h</td>
<td>Yacc-generated header file containing token definitions</td>
</tr>
<tr>
<td>yyparse</td>
<td>Yacc parsing routine</td>
</tr>
<tr>
<td>yylval</td>
<td>value of current token in stack</td>
</tr>
<tr>
<td>yyerror</td>
<td>user-defined error message printer used by Yacc</td>
</tr>
<tr>
<td>error</td>
<td>Yacc error pseudotoken</td>
</tr>
<tr>
<td>yyerrok</td>
<td>procedure that resets parser after error</td>
</tr>
<tr>
<td>yychar</td>
<td>contains the lookahead token that caused an error</td>
</tr>
<tr>
<td>YYSTYPE</td>
<td>preprocessor symbol that defines the value type of the parsing stack</td>
</tr>
<tr>
<td>yydebug</td>
<td>variable which, if set by the user to 1, causes the generation of runtime information on parsing actions</td>
</tr>
</tbody>
</table>