YACC

CSE 310 Compiler Sessional

What is YACC?

- Yet Another Compiler Compiler
- The unix utility *yacc* parses a stream of token, typically generated by *lex*, according to a user-specified grammar.

YACC INPUT FILE FORMAT

- A *yacc* file looks much like a *lex* file:

...definitions...

%%

...rules...

%%

...code...

Definitions

- There are three things that can go in the definitions section:
- **C code** Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.
- **Definitions** The definitions section of a *lex* file was concerned with characters; in *yacc* this is tokens. These token definitions are written to a .h file when *yacc* compiles this file.
- **Associativity rules** These handle associativity and priority of operators.

RULES & CODE

- **Rules -** As with *lex*, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the case with *lex*.

```
name1 : THING something OTHERTHING {action}
      | othersomething THING {other action}
name2 : .....
```

- **Code -** This can be very elaborate, but the main ingredient is the call to **yyparse( )**, the grammatical parse.

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

TOKEN

- If *lex* is to return tokens that *yacc* will process, they have to agree on what tokens there are. This is done as follows.
- The *yacc* file will have token definitions

  %token NUM

  in the definitions section.
- When the *yacc* file is translated with yacc -d –o filename.c filename.y, a header file filename.h is created that has definitions like

  #define NUM 258

  This file can then be included in both the *lex* and *yacc* program.
- The *lex* file can then return **NUM**, and the *yacc* program can match this token.
- The return codes that are defined from **%TOKEN** definitions typically start at around **258**.

yylval

- In addition to specifying the return code, the *lex* parser can return a value that is put on top of the stack, so that *yacc* can access it. This symbol is returned in the variable **yylval**. By default, this is defined as an **int**, so the *lex* program would have

```
extern int yylval;
%%
[0-9]+  {
                yylval = atoi(yytext);
                return NUM;
        }
%%
```

yylval (Contd.)

- In *yacc* file we can use token “NUM” returned by lex

```
expr: NUM '+' NUM
```

- It more than one type of value is to be returned, the possible return values need to be stated:

```
%union {int ival; double dval;}
```

- These types need to be connected to the possible return tokens:

```
%token <ival> INDEX
%token <dval> NUM
```

- Types of non-terminals also need to be specified:

```
%type <dval> expr
```

PRECEDENCE & ASSOCIATIVITY

- **expr:**

```
expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '^' expr
| '(' expr ')'
| '-' expr
| NUM
```

-
```
%left '+' '-'
%left '*' '/'
%right '^'
%nonassoc UMINUS
```

ACCESSING VALUE STACK

- We use $ to access value returned by *lex*

```
expr:
    expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr { $$ = $1 / $3; }
    | expr '^' expr { $$ = $1 ^ $3; }
    | '(' expr ')' { $$ = $2; }
    | '-' expr { $$ = -$2; }
    | NUM {}
    ;
```

Code of a Simple Parser

- Let's see the following two files
  - simpleLexer.l
  - simpleParser.y

- Commands for Compilation and Execution:
  - **`yacc –d simpleParser.y`**
    - **`Generates y.tab.c and y.tab.h`**
  - **`g++ -c –o parser.o y.tab.c`**
    - **`Compiles y.tab.c to generate obj file parser.o`**
  - **`lex –t simpleLexer.l > lexer.c`**
    - **`-t option renames lex.yy.c to lexer.c`**
  - **`g++ -c –o lexer.o lexer.c`**
    - **`Compiles lexer.c to generate obj file lexer.o`**
  - **`g++ -o myParser lexer.o parser.o –ll –ly`**
    - **`Generate executable myParser by linking lexer.o and parser.o to lex and yacc libraries`**
  - **`./myParser`**

Assembly Code Generation

PREREQUISITES

- You have to study Chap: 6 first.
  - Specially intermediate code generation for expression and control-flow statements.
- Clarify your idea about
  - code attribute that holds the 3-address code
  - addr attribute that hold the temporary names
  - label to implement jumps and newlabel() and newtemp() functions

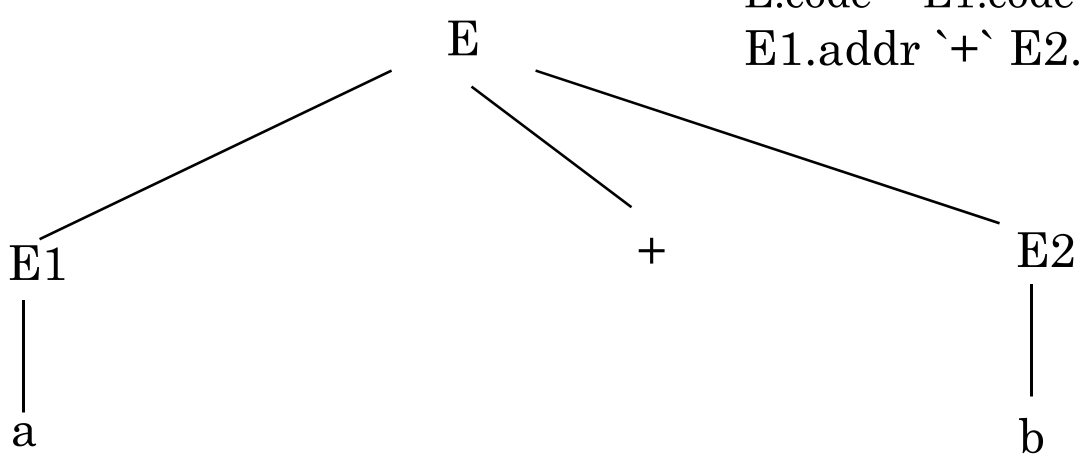
$E \rightarrow E1+E2$ E.addr = new.Temp()

E.code = E1.code || E2.code ||

gen(E.addr `=` E1.addr `+` E2.addr);

E.addr = new.Temp() = t1

E.code = E1.code || E2.code || gen(E.addr `=` E1.addr `+` E2.addr); => E.code = t1 = a + b



- x := a + 1
- Code for addition:
  - MOV AX, [a]
  - ADD AX, 1
  - MOV word [t1], AX
- Code for assignment:
  - MOV AX, [t1]
  - MOV word [x], AX

```
simple_expression      :    simple_expression PLUS term
{
        SymbolInfo temp;
        temp.code[0]= 0; // code attribute
        temp.key[0] = 0; // addr attribute
        temp.key = newTemp(true); // newTemp function

        temp.code += $1->code;
        temp.code += $3->code;
        temp.code += "MOV AX, " + $1->key + "\n";
        temp.code += "ADD AX, " + $3->key + "\n";
        temp.code += "MOV word [" + temp.key + “], AX" + "\n";
        st.insert(temp);
        $$ = (YYSTYPE) st.search(temp.key);
}
```

```
expression      :       simple_expression       {$$ = $1;}


statement       :       variable ASSIGNOP expression
{
        SymbolInfo temp;
        temp.code[0]= 0;
        temp.key[0] = 0;

        temp.code = $3->code + "MOV AX, [" + $3->key + “]\n";
        temp.code += "MOV word [" + $1->key + “], AX\n";
        $1->code = temp.code;
        $$ = $1;

}
```

```
statement        :        IF expression THEN statement ELSE
   statement
{
            SymbolInfo temp;
            string lab_1 = newlabel();
            string lab_2 = newlabel();
            temp.code[0]= 0;
            temp.key[0] = 0;
            temp.code += $2->code;
            temp.code += "CMP word [" + $2->key + “], 0" + "\n";
            temp.code += "JE NEAR " + lab_1 + "\n";
            temp.code += $4->code;
            temp.code += "JMP NEAR " + lab_2 + "\n";
            temp.code += lab_1 + ":\n";
            temp.code += $6->code ;
            temp.code += lab_2 + ":\n";
            $2->code = temp.code;
            $2->key[0] = 0;
            $$ = $2;
}
```

```
simple_expression : simple_expression OR term
{
                SymbolInfo temp;
                string lab_1 = newlabel();
                string lab_2 = newlabel();
                temp.key = newtemp(true);
                temp.code[0]= 0;
                temp.key[0] = 0;
                temp.code += $1->code;
                temp.code += $3->code;
                temp.code += "CMP word [" + $1->key + “], 0" + "\n";
                temp.code += "JNE NEAR " + lab_1 + "\n";
                temp.code += "CMP word [" + $3->key + “], 0" + "\n";
                temp.code += "JNE NEAR " + lab_1 + "\n";
                temp.code += "MOV word [" + temp.key + “], 0" + "\n";
                temp.code += "JMP NEAR " + lab_2 + "\n";
                temp.code += lab_1 + ":\n";
                temp.code += "MOV word [" + temp.key + “], 1" + "\n";
                temp.code += lab_2 + ":\n";
                st.insert(temp);
                $$ = (YYSTYPE) st.search(temp.key);
  };
```

Special Thanks to

- Rajkumar Das

  Assistant Professor

  Department of CSE, BUET