UNIVERSITY OF NICOSIA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΛΕΥΚΩΣΙΑΣ

COMP-421 Compiler Design

Presented by
Dr Ioanna Dionysiou

Lecture Outline

- Introduction to yacc

Introduction to YACC

- Yet Another Compiler-compiler
  - It is a tool for building syntax analyzers
- Yacc takes a grammar (series of rules) that you specify and writes a parser that recognizes valid strings in that grammar
  - Non-recursive rules
    - statement → NAME = expression
    - expression → NUMBER + NUMBER
    - S → 0 S | 1

Yacc

- LR parser
  - Shift/reduce parsing
    - When yacc reduces a rule, it executes user-code associated with it
  - What cannot be parsed
    - Ambiguous grammar
    - Lookahead symbol > 1
      - (if-the-else situation , need to left factor the grammar)

Yacc specification sections

- A yacc specification consists of three sections
  - Definition section
  - Rules section
  - User subroutines section

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

- The parts are separated by lines consisting of %%
- The first two parts are required, although a part may be empty
- The third part and the preceding %% may be omitted

Definition Section

- It includes declarations
  - %token
  - %type
  - %left
  - %right
  - %start
  - %union
  - %nonassoc

Symbols

- A grammar is constructed from symbols which are strings of letters, digits, _
  - Every symbol has a value
    - Error is reserved for error recovery
  - It gives additional information about a particular instance of a symbol
    - Token is a number ==> value would be a particular number
    - Token is a literal string ==> value would be a pointer to a copy of the string
    - Token is variable ==> value would be a pointer to a symbol table entry describing the variable

Symbols and tokens

- Symbols produced by the lexer are called tokens
  - These are the ones that the lexer passes to the parser
  - When a yacc parser needs another token, it calls yylex() which returns the next token from the input

Tokens and %token

- All tokens must be defined in the declaration sections

```
%token NAME NUMBER
```

- You can also user single quoted characters as tokens without declaring them
  - ‘+’ ‘=‘, etc

Tokens, %union and %type

- The %union declaration identifies all of the possible C types that a symbol value can have
- The field declarations are copied into a C union declaration of the type YYSTYPE in the output file
  - In the absence of a %union declaration, yacc defines YYSTYPE to be int, so all of the symbol values are integers
  - You associate the types declared in the %union with particular symbols using the %type declaration

%union and %type

```
%union {
   double dval;
   char *sval;
}

%token <dval> NUMBER
%token <sval> NAME
```

Precedence Specification

- Yacc lets you specify precedence explicitly
  - Operators are declared in increasing order of precedence
  - Operators on the same line are the same precedence level

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

%start

- Normally, the start rule is the one named in the first rule.
- If you want to start with some other rule, you can write

```
%start rulename
```

- In most cases, the clearest way to present the grammar is top-down, with the start rule first
  - No %start needed

Rules Section

- It includes
  - Rules
  - Actions
- Whenever a parser reduces a rule, it executes user C code associated with the rule, known as rule's action.

Rule Format

- ASCII keyboards don't have a → key
  - we use a colon : between the left-hand and right-hand sides of a rule
  - We put a semicolon ; at the end of each rule

```
%%
statement : NAME '=' expression
          | expression
          ;
```

  - Unlike lexx, yacc pays no attention to line boundaries in the rules section

Action

- The action appears in braces after the end of the rule, before the ; or the |
  - Action code can refer to the values of the right-hand side symbols as $1, $2, …
  - Action code can set value of the left-hand side symbol by setting $$

Action

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

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

Lex Example

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+       {
                 yylval = atoi(yytext);
                 return INTEGER;
             }

[-+\n]       return *yytext;

[ \t]        ; /* skip whitespace */

.            yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Yacc Example

```
%{
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'          { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                    { $$ = $1; }
        | expr '+' expr            { $$ = $1 + $3; }
        | expr '-' expr            { $$ = $1 - $3; }
        ;

%%

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

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

Yacc Example

```
%{
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

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

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

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

Add expr '*' expr
Add expr '/' expr