Compiler Design
编译器构造实验

Lab 4：YACC

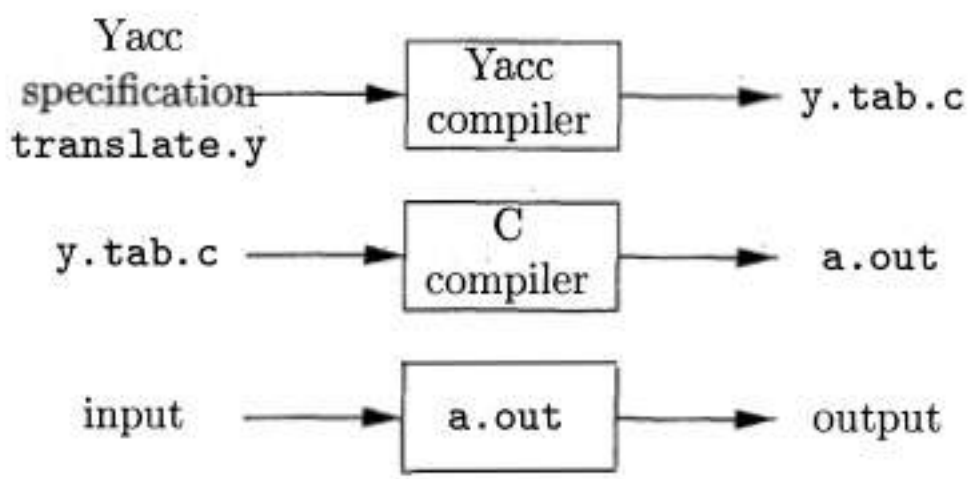
张献伟

xianweiz.github.io

DCS292, 3/10/2022

Yacc Overview

- Yacc is an LALR(1) parser generator
  - YACC: Yet Another Compiler-Compiler
  - Parse a language described by a context-free grammar (**CFG**)
  - Yacc constructs an **LALR(1)** table
- Available as a command on the UNIX system
  - Bison: free GNU project alternative to Yacc



Yacc Specification

- **Definitions** section[定义]:
  - C declarations within %{ %}
  - Token declarations
- **Rules** section[规则]:
  - Each rule consists of a grammar production and the associated semantic action
- **Subroutines** section[辅助函数]:
  - User-defined auxiliary functions

```
%{
  #include …
%}
%token NUM VAR
- - - - - - - - - - - - - - - - -
%%
production { semantic action }
…
- - - - - - - - - - - - - - - - -
%%
…
```

Write a Grammar in Yacc

- A set of productions <head> → $\langle body\rangle_1$ | … | $\langle body\rangle_n$ would be written in YACC as:

  <head> : $\langle body\rangle_1$ { $\langle semantic\ action\rangle_1$ }
  …
  : $\langle body\rangle_n$ { $\langle semantic\ action\rangle_n$ }
  ;

- Usages
  - Tokens that are single characters can be used directly within productions, e.g. '+'
  - Named tokens must be declared first in the declaration part using %token *TokenName*

Write a Grammar in Yacc (cont.)

- Semantic actions may refer to values of the synthesized attributes of terminals and non-terminals in a production:

  $X : Y_1\ Y_2\ Y_3 \ldots Y_n$ { action }

  - $$ refers to the value of the attribute of X (non-terminal)
  - $i refers to the value of the attribute of Y_i (terminal or non-terminal)
  - Normally the semantic action computes a value for $$ using $i's

- Example: $E \rightarrow E + T \mid T$

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

default action: {$$ = $1 }

Example: $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid num$

```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

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

%%

lines : 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; }
     | NUMBER
     ;
```

Can we remove those two lines?

Allow to evaluate a sequence of expressions, one to a line

ε

Example (cont.)

```

%%

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

int main() {
    if (yyparse() != 0)
        fprintf(stderr, "Abnormal exit\n");
    return 0;
}

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

calls yylex() to get successive tokens

Compile and Run …

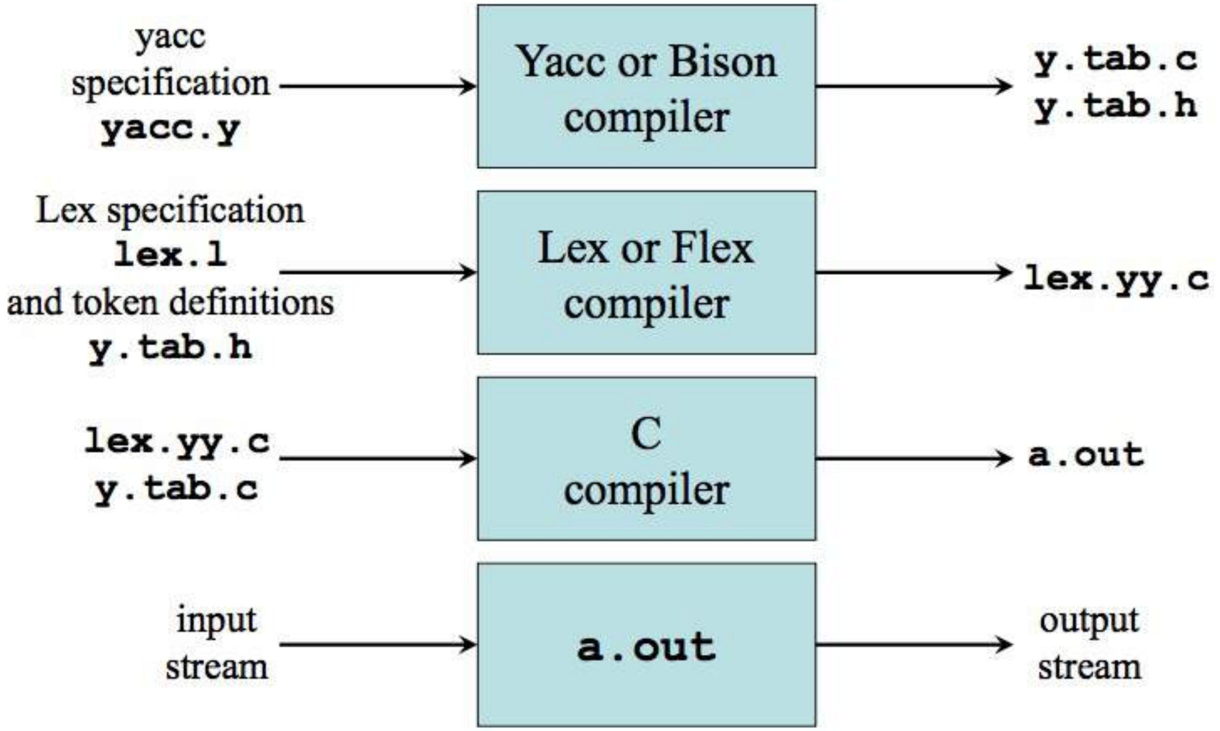
- Compile
  - $yacc -d *parser*.y
  - $gcc -o *test* y.tab.c

- Run
  - $./*test* < exprs.txt

```
1 + 5
1 * 2 + 10
10 - 2 -3
```

Yacc + Lex

- Lex was designed to produce lexical analyzers that could be used with Yacc
- Yacc generates a parser in y.tab.c and a header y.tab.h
- Lex includes the header and utilizes token definitions
- Yacc calls yylex() to obtain tokens



Example: Yacc + Lex

parser.y

```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

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

%%

lines : 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; }
     | NUMBER
     ;

%%

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

int main() {
    if (yyparse() != 0)
        fprintf(stderr, "Abnormal exit\n");
    return 0;
}

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

lexer.l

```
%{
 #define YYSTYPE double
 #include "y.tab.h"
 extern double yylval;
%}
number [0-9]+\.?|[0-9]*\.[0-9]+

%%

[ ]         { /* skip blanks */ }
{number}    { sscanf(yytext, "%lf", &yylval);
                return NUMBER; }
\n|.        { return yytext[0]; }

%%

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

Generated by Yacc

Defined in y.tab.c

Compile and Run …

- Compile
  - $yacc -d *parser*.y
  - $lex *lexer*.l
  - $gcc -o *test* y.tab.c lex.yy.c

- Run
  - $./*test* < exprs.txt

```
1 + 5
1 * 2 + 10
10 - 2 -3
```

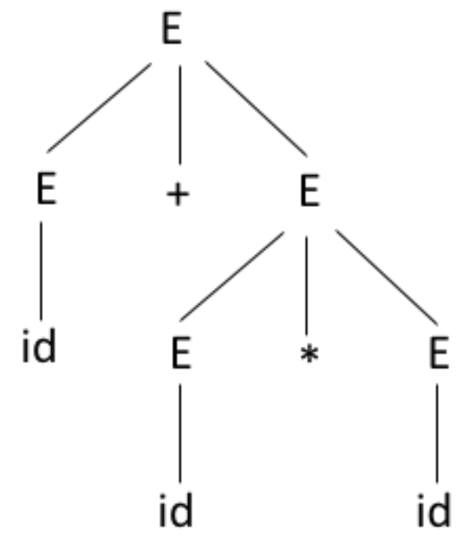
References

- 编译原理（第2版），章节4.9
- Yacc/Bison - Parser Generators, https://tldp.org/LDP/LG/issue87/ramankutty.html
- Lex and Yacc – A Quick Tour, https://courses.cs.washington.edu/courses/cse322/07au/slides/lec25.pdf
- ANTLR, Yacc, and Bison, https://www.cs.csustan.edu/~xliang/Courses/CS4300-20F/Notes/Ch4c.pdf
- Yacc Practice, https://epaperpress.com/lexandyacc/pry1.html
- The Lex & Yacc Page, http://dinosaur.compilertools.net/

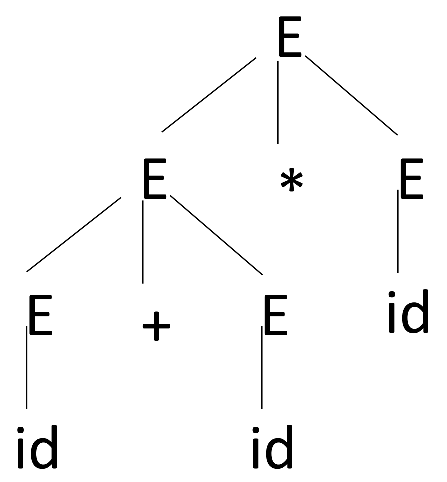
Backup …

Example

- Grammar: $E \rightarrow E*E \mid E+E \mid (E) \mid id$



- Two distinct leftmost derivations for the sentence id + id * id
  - Above: id + (id * id)
  - Below: (id + id) * id
- How to evaluate a + b * c ?
  - a + (b * c) ?
  - (a + b) * c ?
- Grammar $E \rightarrow E*E \mid E+E \mid (E) \mid id$ is ambiguous



The deepest sub-tree is traversed first, thus higher precedence

Precedence and Associativity[优先级/结合性]

- Two characteristics of operators that determine the evaluation order of sub-expressions w/o brackets
- Operator **precedence**[优先级]
  - Determines which operator is performed first in an expression with more than one operators with different precedence
  - 10 + 20 * 30
- Operator **associativity**[结合性]
  - Is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left
  - 100 / 10 * 10 -> (100 / 10) * 10 //left associativity
  - a = b = 1 -> a = (b = 1) //right associativity

```
int a,b=1,c=2;
a=b=c;
```

a = 2 (not 1)

How to Remove Ambiguity?[消除二义性]

- Step I: **Specify precedence**[指定优先级]
  - Build precedence into grammar by recursion on a different non-terminal for each precedence level. Insight:
    - Lower precedence — tend to be higher in tree (close to root)
    - Higher precedence — tend to be lower in tree (far from root)
  - Place lower precedence non-terminals higher up in the tree

- Rewrite $E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$ to:

$E \rightarrow E + E \mid T$ // lowest precedence +

$T \rightarrow T * T \mid F$ // middle precedence *

$F \rightarrow (E) \mid \text{id}$ // highest precedence ()

https://people.cs.pitt.edu/~jmisurda/teaching/cs1622/slides/cs1622-parsing-part2.pdf

How to Remove Ambiguity (cont.)

- Step II: **Specify associativity**[指定结合性]
  - Allow recursion only on either left or right non-terminal
    - Left associative — recursion on left non-terminal
    - Right associative — recursion on right non-terminal
- Even after step 1, ambiguous due to associativity
  - $E \rightarrow E + E$ .... ; allows both left/right associativity
- Rewrite:

  $E \rightarrow E + E \mid T$ // lowest precedence +

  $T \rightarrow T * T \mid F$ // middle precedence *

  $F \rightarrow (E) \mid \text{id}$ // highest precedence ()

  to

  $E \rightarrow E + T \mid T$ // + is left-associative

  $T \rightarrow T * F \mid \text{F}$ // * is left-associative

  $F \rightarrow (E) \mid \text{id}$

https://people.cs.pitt.edu/~jmisurda/teaching/cs1622/slides/cs1622-parsing-part2.pdf

The Example

- Grammar $E \rightarrow E*E \mid E+E \mid (E) \mid \mathrm{id}$ was ambiguous
  - Rewrite to

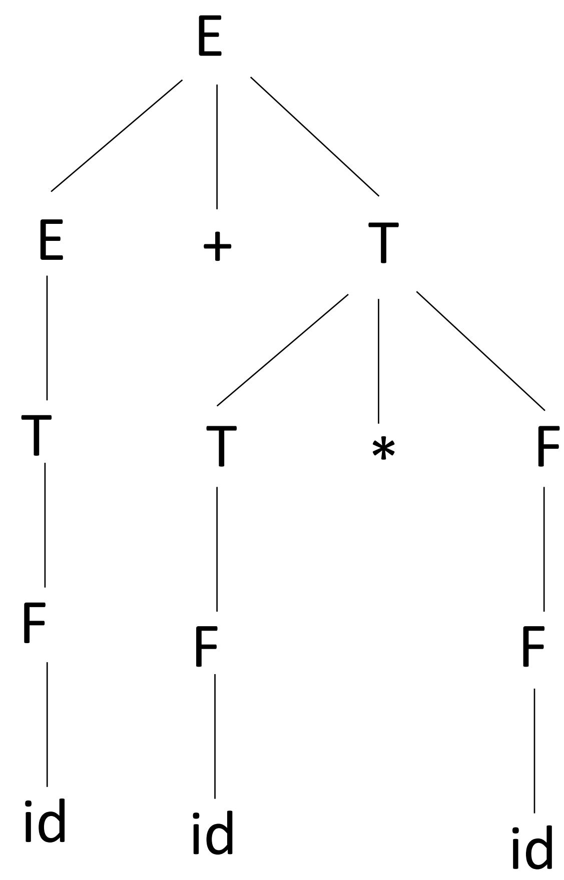
    $E \rightarrow E + T \mid T$

    $T \rightarrow T * F \mid F$

    $F \rightarrow (E) \mid \mathrm{id}$

- The id + id * id has only one parse tree now

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow \mathrm{id} + T * F \\
&\Rightarrow \mathrm{id} + F * F \\
&\Rightarrow \mathrm{id} + \mathrm{id} * F \\
&\Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}
\end{aligned}
$$



Summary: Ambiguity Removal

- How to remove the ambiguity?
- Specify precedence
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

⇒

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

still possible to get
id + (id + id)
and
(id + id) + id
what if ‘-’ (minus)?

⇒

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

Now, can only have more ‘+’ on left

SUN YAT-SEN UNIVERSITY