Формални Езици и Езикови Процесори
ТУ, кат. КС, летен семестър 2012

Лекция 13

Тема:

Компилатори на Компилатори /Програма YACC/

Съдържание:

- YACC – генератор на парсери
- Формат на входа
- Действия
- Нееднозначности и конфликти
- Примери

Как се строят синтактични анализатори/парсери?

- Ръчно - според теорията на СА;
- Автоматизирано - utilities генерират първичния код на парсери съгласно описание на синтаксиса във формат на КСГ.

YACC е най-известният генератор на парсери (1975, Johnson, за UNIX). Използва се независимо или в комбинация с друга utility LEX (генератор на сканери). Двете програми са известни като компилатори на компилатори **compiler-compilers**.

Популярни К-К

- Непълен списък 33 compiler-compilers може да се види на www.wikipedia.org
- Някои версии, базирани на yacc/lex – виж следния слайд

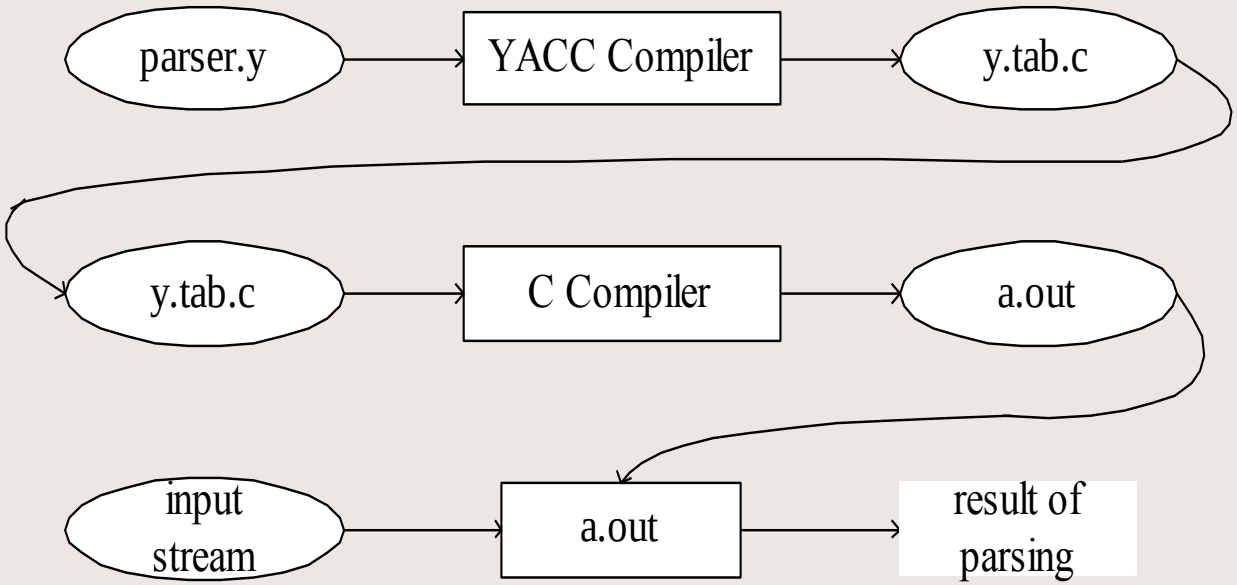
Популярни К-К

- AT&T: lex, yacc
- Berkely: yacc
- Berkely/gnu: flex
- GNU: bison
- MKS: lex, yacc
- ABRAXAS: pclex, pcyacc
- ANTLR (старо име PCCTS): Another Tool for Language Recognition, или “Anti-LR” парсер

Въведение в YACC

- YACC е генератор на синтактични анализатори /парсери/.
- Нарича се още YACC компилатор.
- Следва:
  - Физическа схема на потока данни;
  - Функционална и логическа схема.

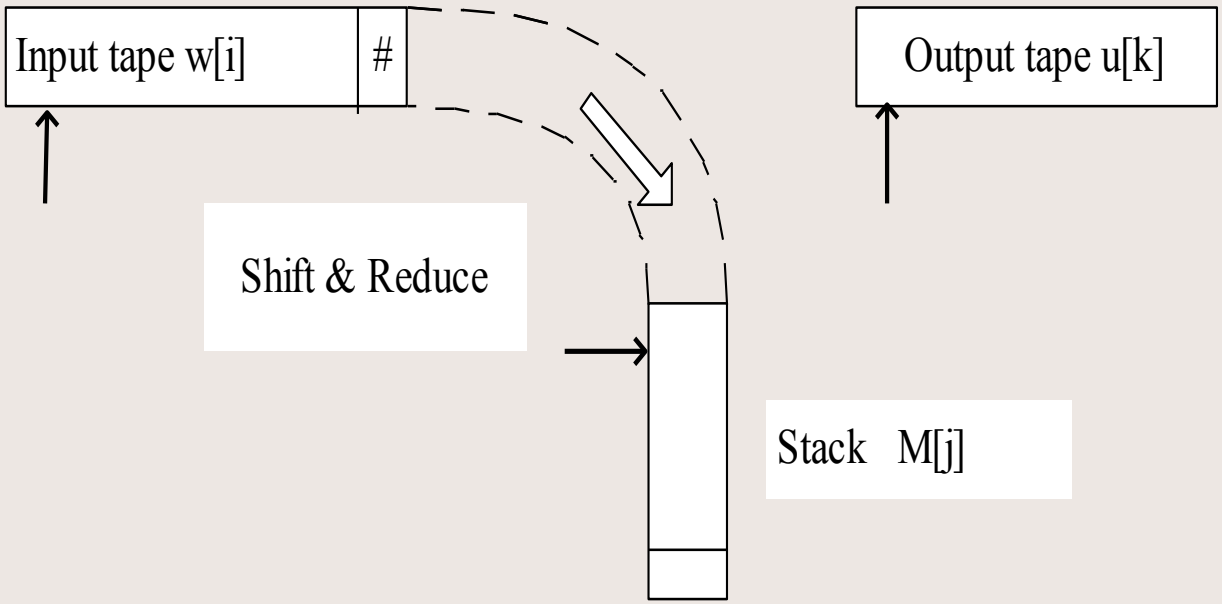
Физическа схема на потока данни



Физическа схема на потока данни

- Описание на парсер се подготвя като входни данни за YACC, например файл *parser.y*.
- На изхода си YACC създава Си програма във файл *y.tab.c*, който включва:
  - Таблица на преходите за стеков КА, според правилата на КСГ от входния файл *parser.y*.
  - Текст на функция *int yyparse( )*, която симулира стеков КА за разпознаване на входни низове.
- Файлът *y.tab.c* се компилира от C compiler до изпълнима програма *a.out*.
- *a.out* чете входен поток и разпознава (приема като валидни или отхвърля като невалидни) изречения.

Функц./Лог. схема



Теория за YACC

- YACC генерираният синтактичен анализатор оперира като LALR(1) parser. Става дума за модифицирана версия на т.н. SR Shift/Reduce алгоритми:
  - Входни символи се пренасят в стек **S**hifted/pushed
  - Top stack се проверява за локализиране на основа (най-лявата проста фраза, т.е дясна страна на продукция), която се редуцира/свива **R**educed до своя ляв нетерминал.

Теория за YACC

- Разнообразието от bottom-up SA методи се основава на подходите/начините за локализиране на основата. Всички LR(1), SLR(1), LALR(1) методи са детерминирани с управл таблица (редове – азбука на стека и колони $\Sigma \cup N \cup \{\#\}$) и съдържание:
  - Shift
  - Reduce
  - Accept – Успех, край на разбора
  - Error

Логическа схема

Както беше казано, YACC compiler генерира:

- Таблица на преходите за стеков КА
- Функция *int yyparse()* ползва горната таблица за да разпознава вх. поток.

Стековият КА е естественият модел за създаване на синтактични анализатори. Такова е и поведението на продукта, създаден от YACC.

YACC формат на входа

Структура на вх данни:

definitions (declarations)	незадължителен
%%	**задължителен**
grammar rules	**задължителен**
%%	незадължителен
supporting (user defined)	незадължителен
C-routines	

Припомняне: LEX формат

– LEX спецификация:

Definitions (declarations)	незадължителен
%%	**задължителен**
Translation rules	незадължителен
%%	незадължителен
Auxiliary (user def) – C-routines	незадължителен

YACC правила на граматика

Най-важен вход за YACC групата ***grammar rules***. Всяко правило се представя в следния формат:

<лява част> **:** <дясна част> **;**

: е разделител.

; е краен ограничител.

YACC правила на граматика

Най-важен вход за YACC групата ***grammar rules***. Всяко правило описва допустима структура и й дава име. Примери:

date : month day ‘,’ year ;

date : month ‘/’ day ‘/’ year ;

month : ‘J’ ‘a’ ‘n’ ;

month : ‘F’ ‘e’ ‘b’ ;

. . .

month : ‘D’ ‘e’ ‘c’ ;

YACC правила на граматика

Всяко правило съдържа:

- Имена, които означават терминали /tokens/ или нетерминали
- Литерали винаги означават терминали
- : служи като метасимвол
- ; служи като метасимвол

YACC правила на граматика

Общ формат на YACC продукция:

a : body ;

- ***a*** е не терминален символ;
- ***body*** е последователност от нула, един или повече имена и/или литерали;
- **:** и **;** са мета символи;
- Имена се строят от букви, цифри, dot/./ и underscore/_/.
- Литерал значи единичен символ, заграден в апострофи.

YACC правила на граматика

Едно правило като изложеното

<left part> → <alt_1> | <alt_2> | . . . | <alt_n>

Може да се запише по два алтернативни начина:

```
<left part>  :  <alt_1>
             |  <alt_2>
                . . .
             |  <alt_n>
             ;
```

```
<left part>  :  <alt_1> ;
<left part>  :  <alt_2> ;
             . . .
<left part>  :  <alt_n> ;
```

YACC правила на граматика

Едно правило като изложеното

<left part> → <alt_1> | <alt_2> | . . . | <alt_n>

Може да се запише по два алтернативни начина:

```
<left part>  :   <alt_1>   { semantic action_1 }
             |   <alt_2>   { semantic action_2 }
                   . . .
             |   <alt_n>   { semantic action_n }
             ;
```

```
<left part>   :   <alt_1>   { semantic action_1 } ;
<left part>   :   <alt_2>   { semantic action_2 } ;
                . . .
<left part>   :   <alt_n>   { semantic action_n } ;
```

YACC правила на граматика

Примери:

```
a : B C D ;
a : E F ;
a : G ;
```

```
a : B C D
  | E F
  | G
  ;
```

Празно правило /The empty rule/:

Как се записва епсилон продукция:

$$X \rightarrow \varepsilon$$

```
X  :  ;
```

Детайли правила на граматика

- Имена, представящи терминални символи, се декларират *%token ime1 ime2 ime3 …*
- Не декларирани имена се третират като нетерминални символи.
- Всеки нетерминал трябва да се появява от лява страна на продукция.
- Стартовият нетерминал: или като първа продукция, или се декларира явно със запис *%start ime*

продължение →

Детайли правила на граматика

- Служебен токен, наречен *endmarker*, маркира края на входния поток за парсера.
- Ако всички токени от вх поток до *endmarker* формират структура, която се свива до стартовия нетерминал, парсерът разпознава входния низ като верен, приема го и го връща на главната програма след прочитане на *endmarker*, като стане текущ сканиран смвол.
- Ако *endmarker* се разпознае в кой да е друг контекст, това е признак за грешка и входният поток следва да се отхвърли.

Действия

Действия се вграждат в продукциите.

<ляв нетерминал> : <дясна_част> { семант действие } ;

С всяко правило се допуска задаване на действие, което се активира при прилагане на продукцията. YACC семантично действие е последователност от оператори на Си, заградени в скоби, т.е съставен оператор.

```
a  :  '(' B ')'   {  hello(1,"ABC");  }  ;
xx : yy  zz       {  printf("message");  flag=25;  }  ;
```

Действия

- Действията могат да връщат стойности
- Действията могат да получават стойности, върнати от предишни действия
- Сканерът може да предоставя стойности за разпознати токени (променлива *yylval*)
- Възможна е комуникация между действия и разпознавателя.

Действия

За комуникация между действията и парсера се въвеждат специфични обекти. Наричат се ***pseudo variables*** или ***positional arguments*** и се именоват с **$$, $1, $2**, …

- За да върне стойност, едно действие присвоява данни на променливата $$, например { $$ = 1; }
- За да получи и ползва стойности, върнати от предишни действия, едно действие следва да се позове на променливите $1, $2, … които се отнасят до стойности, върнати от съответни компоненти в дясна страна на продукция, номерирани 1, 2, ... в посока Л > Д.

Примери с Действия

```
a : b c d ;
```

$$ е стойност, асоциирана с a

$1 е стойност, върната от b

$2 е стойност, върната от c

$3 е стойност, върната от d

Примери с Действия

```
expr : ‘(‘expr ‘)’          { $$ = $2; }  ;
expr : expr ‘+’ expr        { $$ = $1 + $3; }   ;
```

Примери с Действия

```
a : b ;
a : b  { $$ = $1; }  ;    // действие неявно
```

SA - LA (parsing - scanning)

- Всеки парсер се нуждае от сканер.
- Лексемите са отделни низове от входния поток.
- Токените са описатели, които представят отделни категории лексеми.
- Сканерът:
  - Цепи source file, като превежда лексемите в токени.
  - Конструира пореден токен и го връща/предоставя на парсера
- Парсерът:
  - Активира /Invokes/ сканера
  - Проверява входа за правилен синтаксис като ползва токените, генерирани от сканера
  - Създава parse tree, представящо входната програма

YACC и LEX

- Всеки парсер се нуждае от сканер
- Парсер, генериран от YACC, се нуждае от сканер на име *yylex( )*
- Сканерът *yylex( )* може да се създаде
  - Ръчно, By hand
  - Автоматизирано, By LEX

Сканер за YACC

- Задача: Нуждаем се от сканер, който връща токен DIGIT, и допълнително се нуждаем от числената стойност на текущата цифра. Ето текстът на сканера:

```
int yylex() { int ch; extern int yylval;
   while (     (ch=getchar()) ==‘ ‘      ) ;
   switch(ch)
   {
         case ‘0’:
         . . .
         case ‘9’:          yylval = ch-’0’; return DIGIT;
   }
}
```

YACC source дефиниции

%{

// Си глобални дефиниции и/или препроцесор директиви

%}

// декларации на терминални символи (%token)

и/или следвани от

// декларация на стартов нетерминал (%start)

и/или следвани от

// декларация за асоциативни операции (%left, %right)

и/или следвани от

// декларация за не асоциативни операции (%nonassoc)

YACC употреба

Как да изпълним парсер, синтезиран от YACC/bison с употреба на сканер функция *yylex()*, ръчно написана от потребителя

yacc example1.y	*bison example1.y*
cc y.tab.c	*gcc example1.tab.c*
./a.out	*./a.out*

YACC употреба

Как да изпълним парсер, синтезиран от YACC/bison с употреба на сканер функция *yylex()*, генерирана от LEX

yacc -d example1.y	*bison -d example1.y*
lex example1.lex	*lex example1.lex*
cc lex.yy.c y.tab.c	*cc lex.yy.c example1.tab.c*
./a.out	*./a.out*

YACC и LEX

a/ Всеки парсер, генериран от YACC, се нуждае от сканер на име *yylex( )*

b/ структура на scanner/parser модул, генериран използвайки YACC и LEX

c/ обмен на унифицирано множество токени между изхода на LEX и изхода на YACC става чрез заглавен файл ***y.tab.h,*** създаден с квалификатор ***yacc –d*** qualifier

YACC пример

Задача: Да се опише вход за YACC, който да генерира парсер, разпознаващ низове, описвани със следната КСГ:

$$A \rightarrow x \mid (B)$$

$$B \rightarrow A\ C$$

$$C \rightarrow +\ A\ C \mid \varepsilon$$

Pure Version 1 (literals и non terminals)

- YACC входът съдържа:
  - Терминални символи като литерали
  - Нетерминални символи като имена
- **Няма описани токени**
- Не се налага сканерът *yylex()* да генерира и връща токени
- Сканерът връща **само литерали**.

YACC вход (1/2)

```
%{
//  file examp7.y
#include <stdio.h>
%}
%%
a   :  ‘x’            { printf(“\n1     Rule  A → x          applied”); }
    |  ‘(‘  b  ‘)’    { printf(“\n2     Rule  A → ( B )      applied”); }
    ;
b   :      a  c       { printf(“\n3     Rule  B → A  C       applied”); }
    ;
c   :  ‘+’  a  c     { printf(“\n4     Rule  C → + A C      applied”); }
    |  /*empty*/     { printf(“\n5     Rule  C → ε          applied”); }
    ;
%%
```

YACC вход (2/2)

```
int yylex()      {
                 int ch;  ch = getchar();
                 while  ( ch==’ ’ )  { ch = getchar(); }
                 return ch;
                 }

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

int yyerror(char *param)  {
                 printf(“\n%s”, param);    return 0;      }
```

Pure Version 2 (tokens и non terminals)

- YACC входът съдържа:
  - Терминални символи като токени
  - Нетерминални символи като имена
- **Токени за всички литерали** са описани явно
- Налага се сканерът *yylex()* да генерира и връща токени
- Сканерът връща **само токени**

YACC вход (1/2)

```
%{
//  file examp7.y
#include <stdio.h>
%}
%token  XCHAR   LP  RP  PLUS
%%
a   :   XCHAR       { printf("\n1      Rule  A → x          applied"); }
    |   LP   b   RP     { printf("\n2      Rule  A → ( B )      applied"); }
    ;
b   :       a   c       { printf("\n3      Rule  B → A  C      applied"); }
    ;
c   :   PLUS   a   c   { printf("\n4      Rule  C → + A C    applied"); }
    |   /*empty*/       { printf("\n5      Rule  C → ε          applied"); }
    ;
%%
```

YACC вход (2/2)

```
int yylex()          {
                     int ch;  ch = getchar();
                     while  ( ch == ’ ’ )  { ch = getchar(); }
                     if  ( ch == ’x’ )     return XCHAR;
                     if  ( ch == ’(’ )     return LP;
                     if  ( ch == ’)’ )     return RP;
                     if  ( ch == ’+’ )     return PLUS;
                     return ch;
                     }
int main()           {          yyparse();          return 0; }
int yyerror(char *param)  {
                     printf(“\n%s”, param);   return 0;     }
```

Mixed Version 3 (literals, tokens и non terminals)

- YACC входът съдържа:
  - Терминални символи като литерали
  - Терминални символи като токени
  - Нетерминални символи като имена
- Токени (не за всички литерали) са описани явно
- Налага се сканерът *yylex()* да генерира и връща токени и литерали.
- Сканерът връща **литерали и токени**

YACC вход (1/2)

```
%{
//  file examp7.y
#include <stdio.h>
%}
%token  XCHAR
%%
a   :   XCHAR     { printf("\n1      Rule  A → x          applied"); }
    |   '('   b   ')'   { printf("\n2      Rule  A → ( B )      applied"); }
    ;
b   :       a   c       { printf("\n3      Rule  B → A  C      applied"); }
    ;
c   :   '+'   a   c    { printf("\n4      Rule  C → + A C    applied"); }
    |   /*empty*/   { printf("\n5      Rule  C → ε          applied"); }
    ;
%%
```

YACC вход (2/2)

```
int yylex()      {
                 int ch;  ch = getchar();
                 while  ( ch==’ ’ )  { ch = getchar(); }
                 if  ( ch==’x’ )  return XCHAR;
                 return ch;
                 }
int main()       {        yyparse();      return 0;        }
int yyerror(char *param)  {
                 printf(“\n%s”, param);   return 0;       }
```

YACC пример

Задача: Да се опише вход за YACC, който да генерира калкулатор на Аритм Изрази съгласно следната КСГ:

line → expr ‘\n’

expr → term | expr + term

term → factor | term * factor

factor → digit | (expr)

Правилото **line → expr ‘\n’** е за връзка калкулатор-user. Смисъл: Входът за калкулатора е израз, който завършва с упр символ за нов ред

YACC пример

```
%{
#include <stdio.h>
%}
%token DIGIT
%%
 line     :  expr   '\n'  { printf("%d\n", $1); }
          ;
 expr     :  term
          |  expr  '+'  term  { $$ = $1 + $3; }
          ;
 term     :  factor
          |  term  '*'  factor  { $$ = $1 * $3; }
          ;
 factor   :  DIGIT
          |  '('  expr  ')'      {  $$ = $2;  }
%%
```

YACC пример

```
int yylex()
 { int ch; extern int yylval;
    ch = getchar();
    while (ch ==‘ ‘)   ch = getchar();
    if (isdigit(ch)
    {
          yylval = ch-’0’; return DIGIT;
    }
   return ch;
 }
```

YACC пример – practical hints

```
line    :  expr   ‘\n’  { printf(“%d\n”, $1); }
        ;
```

Входът за калкулатора е израз, който завършва с упр символ за нов ред

```
lines   :  lines  expr  ‘\n’ { printf(“%d\n”, $2); }
        |  lines  ‘\n’
        |  /* empty rule */
        ;
```

Горните правила позволяват на калкулатора:

a/ да обработи последователност от изрази по един на ред

b/ да допуска празни редове между изразите

YACC и неоднозначни граматики

1. Неоднозначни продукции водят до конфликтни ситуации:

Shift/Reduce конфликт: Парсерът може да извърши две легални действия.

Reduce/Reduce конфликт: Парсерът може да извърши две легални редукции.

Shift/Shift конфликт: Никога не е възможен

Илюстрация на S/R конфликт

Неоднозначна КСГ на АИ

E → a | E+E | E*E | (E)

Статус на процеса: expr + expr + expr

^

Стек: |expr| Конфликт:

| + | shift на входа ‘+’ или

|expr| reduce на top stack

Илюстрация на R/R конфликт

Дадена е граматика в нотация YACC:

```
start  :  a   Y
       |   b  Y
       ;
a      |  X  ;
b      |  X  ;
```

две продукции с
една и съща дясна част.

Кое правило да се приложи в случай, че X се локализира на върха на стека за редукция?

Разрешаване на конфликти in general

Прилагат се следните правила:

a/ При shift/reduce конфликт, приема се действие shift.

b/ При reduce/reduce конфликт, приема се да се редуцира съгласно правилото, което по-рано /първо/ се среща в YACC входната последователност.

Разрешаване на конфликти на практика

Припомняне неоднозначна КСГ на АИ

expr → a | expr + expr | expr * expr | (expr)

Два подхода за избягване на неоднозначността:

- Замяна на неоднозначната граматика с еквивалентна еднозначна граматика
- При неоднозначна граматика явно да се декларира асоциативност и приоритет с цел разрешаване на неоднозначността

Замяна на неоднозначна граматика с еднозначна граматика

```
expr   : ‘a’
       | expr ‘+’ expr
       | expr ‘*’ expr
       | ‘(‘ expr ‘)’
       ;
```

```
expr   : term
       | expr ‘+’ term
       ;
term   : factor
       | term ‘*’ factor
       ;
factor : ‘a’
       | ‘(‘ expr ‘)’
       ;
```

Работа с неоднозначна граматика и явно деклариране на асоциативност и приоритет на операциите с цел разрешаване на конфликти от неоднозначност

```
%right  '='
%left  '+'  '-'
%left  '*'  '/'  '%'
%right  '^'
%%
expr    :  'a'
        | expr  '+'  expr
        | expr  '*'  expr
        | '('  expr  ')'
        ;
```

Обработка на грешки и възстановяване след грешка

Служебият токен **error** е запазен и предназначен за обработка на грешки.

```
lines :     lines  expr  '\n'
    |       lines  '\n'
    |       /*  empty  */
    |       error '\n' {  yyerror("reenter last line:");
                          yyerrok;  }
    ;
```

Демо програми

symstr1.y symstr2.y

examp6.y examp7.y examp8.y examp9.y

calc5.y calc6.y

Благодаря
За
Вниманието