Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 7

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing
  (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing (continued)
- YACC Parser generator

Closure of a Set of LR(1) Items

Itemset closure(*I*){ /* I is a set of LR(1) items */
  while (more items can be added to I) {
    for each item $[A \rightarrow \alpha.B\beta,\ a] \in I$ {
      for each production $B \rightarrow \gamma \in G$
        for each symbol $b \in \mathit{first}(\beta a)$
          if (item $[B \rightarrow .\gamma,\ b] \notin I$) add item $[B \rightarrow .\gamma,\ b]$ to *I*
  }
  return *I*
}

Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 7
S → aSb. , b

GOTO set computation

Itemset GOTO(I, X){ /* I is a set of LR(1) items
X is a grammar symbol, a terminal or a nonterminal */
Let $I' = \{[A \rightarrow \alpha X.\beta,\ a] \mid [A \rightarrow \alpha.X\beta,\ a] \in I\}$;
return (*closure*(I'))
}

Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

GOTO(0, S) = 1, GOTO(0,a) = 2, GOTO(2,a) = 4

Construction of Sets of Canonical of LR(1) Items

void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  $C = \{closure(\{S' \rightarrow .S,\ \$\})\}$;/* C is a set of LR(1) item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}

- Each set in C (above) corresponds to a state of a DFA (LR(1) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S,\ \$]$)
Parsing actions for state *i* are determined as follows
1. If $([A \rightarrow \alpha.a\beta,\ b] \in I_i)$ && $([A \rightarrow \alpha a.\beta,\ b] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.,\ a] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If $([S' \rightarrow S.,\ \$] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

4. If $([A \rightarrow \alpha.A\beta,\ a] \in I_i)$ && $([A \rightarrow \alpha A.\beta,\ a] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

LR(1) Grammar - Example 2

Grammar

S' → S
S → L=R | R
L → *R | id
R → L

State 0

S' → .S, $
S → .L=R, $
S → .R, $
L → .*R, =
L → .id, =
R → .L, $
L → .*R, $
L → .id, $

State 1

S' → S., $

State 2

S → L .=R, $
R → L ., $

State 3

S → R., $

State 4

L → *.R, =/$
R → .L, =/$
L → .*R, =/$
L → .id, =/$

State 5

L → id., =/$

State 6

S → L=.R, $
R → .L, $
L → .*R, $
L → .id, $

State 7

L → *R., =/$

State 8

R → L ., =/$

State 9

S → L=R., $

State 10

R → L ., $

State 11

L → *.R, $
R → .L, $
L → .*R, $
L → .id, $

State 12

L → id., $

State 13

L → *R., $

Grammar is not SLR(1), but is LR(1)

A non-LR(1) Grammar

Grammar

$S' \rightarrow S$

$S \rightarrow aSb$

$S \rightarrow ab$

$S \rightarrow \varepsilon$

This grammar is neither SLR(1) nor LR(1), because it is ambiguous

	a	b	$	S
0	S2		R: $S \rightarrow \varepsilon$	1
1			accept	
2	S5	**S3, R: $S \rightarrow \varepsilon$**		4
3			R: $S \rightarrow ab$	
4		S6		
5	S5	**S9, R: $S \rightarrow \varepsilon$**		7
6			R: $S \rightarrow aSb$	
7		S8		
8		R: $S \rightarrow aSb$		
9		R: $S \rightarrow ab$		

LALR(1) Parsers

- LR(1) parsers have a large number of states
  - For C, many thousand states
  - An SLR(1) parser (or LR(0) DFA) for C will have a few hundred states (with many conflicts)
- LALR(1) parsers have exactly the same number of states as SLR(1) parsers for the same grammar, and are derived from LR(1) parsers
  - SLR(1) parsers may have many conflicts, but LALR(1) parsers may have very few conflicts
  - If the LR(1) parser had no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - However, this is not true regarding R-R conflicts
- LALR(1) parsers are as compact as SLR(1) parsers and are almost as powerful as LR(1) parsers
- Most programming language grammars are also LALR(1), if they are LR(1)

Construction of LALR(1) parsers

- The core part of LR(1) items (the part after leaving out the lookahead symbol) is the same for several LR(1) states (the loohahead symbols will be different)
  - Merge the states with the same core, along with the lookahead symbols, and rename them
- The ACTION and GOTO parts of the parser table will be modified
  - Merge the rows of the parser table corresponding to the merged states, replacing the old names of states by the corresponding new names for the merged states
  - For example, if states 2 and 4 are merged into a new state 24, and states 3 and 6 are merged into a new state 36, all references to states 2,4,3, and 6 will be replaced by 24,24,36, and 36, respectively
- LALR(1) parsers may perform a few more reductions (but not shifts) than an LR(1) parser before detecting an error

LALR(1) Parser Construction - Example 1

Grammar
S' → S, S → aSb, S → ε

Grammar is LALR(1)

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

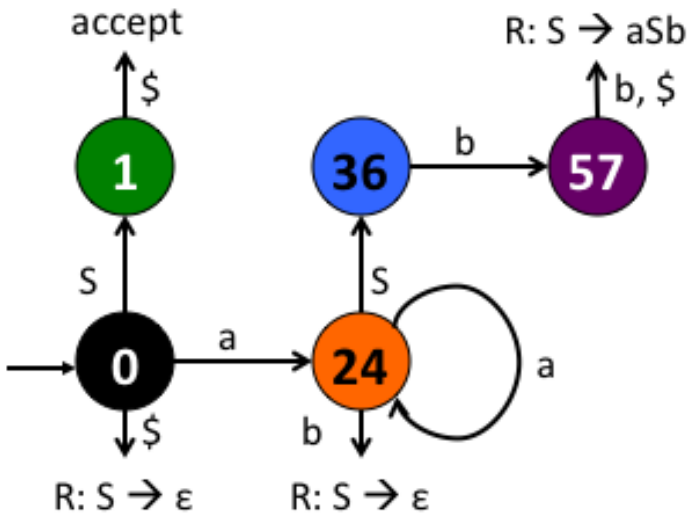
State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	



LALR(1) Parser Construction - Example 1 (contd.)

LR(1) Parser Table

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

LALR(1) Parser Table

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

LALR(1) Parser Error Detection

LR(1) Parser

0	ab$	shift
0 a 2	b$	S → ε
0 a 2 S 3	b$	shift
0 a 2 S 3 b 5	$	S → aSb
0 S 1	$	**accept**

0	aa$	shift
0 a 2	a$	shift
0 a 2 a 4	$	**error**

0	aab$	shift
0 a 2	ab$	shift
0 a 2 a 4	b$	S → ε
0 a 2 a 4 S 6	b$	shift
0 a 2 a 4 S 6 b 7	$	**error**

LALR(1) Parser

0	ab$	shift
0 a 24	b$	S → ε
0 a 24 S 36	b$	shift
0 a 24 S 36 b 57	$	S → aSb
0 S 1	$	**accept**

0	aa$	shift
0 a 24	a$	shift
0 a 24 a 24	$	**error**

0	aab$	shift
0 a 24	ab$	shift
0 a 24 a 24	b$	S → ε
0 a 24 a 24 S 36	b$	shift
0 a 24 a 24 S 36 b 57	$	S → aSb
0 a 24 S 36	$	**error**

Characteristics of LALR(1) Parsers

- If an LR(1) parser has no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - LR(1) and LALR(1) parser states have the same core items (lookaheads may not be the same)
  - If an LALR(1) parser state $s1$ has an S-R conflict, it must have two items $[A \rightarrow \alpha., a]$ and $[B \rightarrow \beta.a\gamma, b]$
  - One of the states $s1'$, from which $s1$ is generated, must have the same core items as $s1$
  - If the item $[A \rightarrow \alpha., a]$ is in $s1'$, then $s1'$ must also have the item $[B \rightarrow \beta.a\gamma, c]$ (the lookahead need not be b in $s1'$ - it may be b in some other state, but that is not of interest to us)
  - These two items in $s1'$ still create an S-R conflict in the LR(1) parser
  - Thus, merging of states with common core can never introduce a new S-R conflict, because shift depends only on core, not on lookahead

Characteristics of LALR(1) Parsers (contd.)

- However, merger of states may introduce a new R-R conflict in the LALR(1) parser even though the original LR(1) parser had none
- Such grammars are rare in practice
- Here is one from ALSU's book. Please construct the complete sets of LR(1) items as home work:
  $S' \rightarrow S\$,\ S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
  $A \rightarrow c,\ B \rightarrow c$
- Two states contain the items:
  $\{[A \rightarrow c.,\ d], [B \rightarrow c.,\ e]\}$ and
  $\{[A \rightarrow c.,\ e], [B \rightarrow c.,\ d]\}$
- Merging these two states produces the LALR(1) state:
  $\{[A \rightarrow c.,\ d/e], [B \rightarrow c.,\ d/e]\}$
- This LALR(1) state has a reduce-reduce conflict

Error Recovery in LR Parsers - Parser Construction

- Compiler writer identifies *major* non-terminals such as those for *program, statement, block, expression*, etc.
- Adds to the grammar, *error productions* of the form $A \rightarrow$ *error* α, where *A* is a major non-terminal and α is a suitable string of grammar symbols (usually terminal symbols), possibly empty
- Associates an error message routine with each error production
- Builds an LALR(1) parser for the new grammar with error productions

Error Recovery in LR Parsers - Parser Operation

- When the parser encounters an error, it scans the stack to find the topmost state containing an *error item* of the form $A \rightarrow .error\ \alpha$
- The parser then shifts a token *error* as though it occurred in the input
- If $\alpha = \epsilon$, reduces by $A \rightarrow \epsilon$ and invokes the error message routine associated with it
- If $\alpha \neq \epsilon$, discards input symbols until it finds a symbol with which the parser can proceed
- Reduction by $A \rightarrow .error\ \alpha$ happens at the appropriate time *Example*: If the error production is $A \rightarrow .error$;, then the parser skips input symbols until ';' is found, performs reduction by $A \rightarrow .error$;, and proceeds as above
- Error recovery is not perfect and parser may abort on end of input

LR(1) Parser Error Recovery

State 0
S -> .rhyme, $
rhyme -> .sound place, $
rhyme -> .error DELL, $
sound -> .DING DONG, $

State 1
S -> rhyme. , $

State 2
rhyme -> sound.place, $
place -> .DELL, $
place -> .error DELL, $

State 3
rhyme -> error.DELL, $

State 4
rhyme -> error DELL. , $

State 5
sound -> DING.DONG, $

State 6
sound -> DING DONG. , $

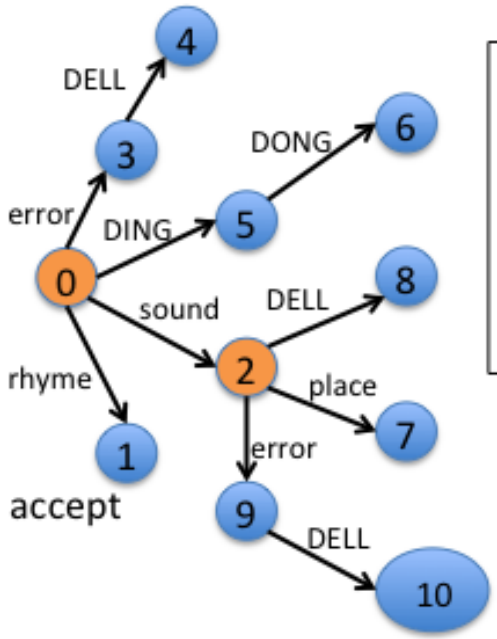
State 7
rhyme -> sound place. , $

State 8
place -> DELL. , $

State 9
place -> error.DELL, $

State 10
place -> error DELL. , $

S -> rhyme
rhyme -> sound place | error DELL
sound -> DING DONG
place -> DELL | error DELL



DING DELL $
0 -> 5 -> error; pops 5;
0 contains error item;
shifts error, reads DELL, enters 4;
reduces by *rhyme -> error DELL*;
reduces by *S -> rhyme*; accepts

DING DONG DING DELL $
0 -> 5 -> 6 -> reduce -> 2 -> error;
2 contains error item;
skips DING; shifts error, reads DELL;
enters 10; reduces by *place -> error DELL*;
enters 7; reduces by *rhyme -> sound place*;
reduces by *S -> rhyme*; accepts

DING $; 0 -> 5 -> error; pops 5;
0 contains error item;
hits $; aborts; solution: add
rhyme -> error instead of
rhyme -> error DELL

YACC:

Yet Another Compiler Compiler
A Tool for generating Parsers

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

YACC Example

```
%token DING DONG DELL
%start rhyme
%%
rhyme : sound place '\n'
          {printf("string valid\n"); exit(0);};
sound : DING DONG ;
place : DELL ;
%%
#include "lex.yy.c"

int yywrap(){return 1;}
yyerror( char* s)
{ printf("%s\n",s);}
main() {yyparse(); }
```

LEX Specification for the YACC Example

```
%%
ding return DING;
dong return DONG;
dell return DELL;
[ ]* ;
\n|. return yytext[0];
```

Compiling and running the parser

```
lex ding-dong.l
yacc ding-dong.y
gcc -o ding-dong.o y.tab.c
ding-dong.o
```

Sample inputs		Sample outputs
ding dong dell	||	string valid
ding dell	||	syntax error
ding dong dell$	||	syntax error

Form of a YACC file

- YACC has a language for describing context-free grammars
- It generates an LALR(1) parser for the CFG described
- Form of a YACC program
  %{ declarations – optional
  %}
  %%
  rules – compulsory
  %%

  programs – optional
- YACC uses the lexical analyzer generated by LEX to match the **terminal symbols** of the CFG
- YACC generates a file named **y.tab.c**

Declarations and Rules

- **Tokens**: %token name1 name2 name3, $\cdots$
- **Start Symbol**: %start name
- **names** in rules: *letter*(*letter* | *digit* | . | _)*
  letter is either a lower case or an upper case character
- **Values of symbols** and **actions**: Example

```
A   :   B
        {$$ = 1;}
        C
        {x = $2; y = $3; $$ = x+y;}
        ;
```

  - Now, value of A is stored in `$$` (second one), that of B in `$1`, that of action 1 in `$2`, and that of C in `$3`.

Declarations and Rules (contd.)

- Intermediate action in the above example is translated into an ϵ-production as follows:

```
$ACT1 : /* empty */
        {$$ =1;}
      ;
   A  : B $ACT1 C
        {x = $2; y = $3; $$ = x+y;}
      ;
```

- Intermediate actions can return values
  For example, the first `$$` in the previous example is available as `$2`
- However, intermediate actions cannot refer to values of symbols to the left of the action
- Actions are translated into C-code which are executed just before a reduction is performed by the parser

Lexical Analysis

- LA returns integers as token numbers
- Token numbers are assigned automatically by YACC, starting from 257, for all the tokens declared using **%token** declaration
- Tokens can return not only token numbers but also other information (e.g., value of a number, character string of a name, pointer to symbol table, etc.)
- Extra values are returned in the variable, **yylval**, known to YACC generated parsers

Ambiguity, Conflicts, and Disambiguation

- $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid id$
- Ambiguity with left or right associativity of '-' and '/'
- This causes shift-reduce conflicts in YACC: (E-E-E) – shift or reduce on -?
- Disambiguating rule in YACC:
  - Default is **shift** action in S-R conflicts
  - Reduce by **earlier** rule in R-R conflicts
  - Associativity can be specified explicitely
- Similarly, precedence of operators causes S-R conflicts. Precedence can also be specified
- Example

```
%right '='
%left '+' '-'        --- same precedence for +, -
%left '*' '/'        --- same precedence for *, /
%right ^             --- highest precedence
```

Symbol Values

- Tokens and nonterminals are both stack symbols
- Stack symbols can be associated with values whose **types** are declared in a **%union** declaration in the YACC specification file
- YACC turns this into a union type called YYSTYPE
- With **%token** and **%type** declarations, we inform YACC about the types of values the tokens and nonterminals take
- Automatically, references to `$1`, `$2`, `yylval`, etc., refer to the appropriate member of the union (see example below)

YACC Example : YACC Specification (desk-3.y)

```
%{
#define NSYMS 20
struct symtab {
        char *name; double value;
     }symboltab[NSYMS];
struct symtab *symlook();
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}
```

YACC Example : YACC Specification (contd.)

```
%union {
    double dval;
    struct symtab *symp;
    }
%token <symp> NAME
%token <dval> NUMBER
%token POSTPLUS
%token POSTMINUS
%left '='
%left '+' '-'
%left '*' '/'
%left POSTPLUS
%left POSTMINUS
%right UMINUS
%type <dval> expr
```

YACC Example : YACC Specification (contd.)

```
%%
lines: lines expr '\n' {printf("%g\n",$2);}
     | lines '\n'      | /* empty */
     | error '\n'
       {yyerror("reenter last line:"); yyerrok; }
     ;
expr : NAME '=' expr {$1 -> value = $3; $$ = $3;}
     | NAME {$$ = $1 -> value;}
     | expr '+' expr {$$ = $1 + $3;}
     | expr '-' expr {$$ = $1 - $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | expr '/' expr {$$ = $1 / $3;}
     | '(' expr ')' {$$ = $2;}
     | '-' expr   %prec UMINUS {$$ = - $2;}
     | expr POSTPLUS {$$ = $1 + 1;}
     | expr POSTMINUS {$$ = $1 - 1;}
     | NUMBER
```

YACC Example : LEX Specification (desk-3.l)

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
            return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
            yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

YACC Example : Support Routines

```
%%
void initsymtab()
{int i = 0;
 for(i=0; i<NSYMS; i++) symboltab[i].name = NULL;
}
int yywrap(){return 1;}
yyerror( char* s) { printf("%s\n",s);}
main() {initsymtab(); yyparse(); }

#include "lex.yy.c"
```

YACC Example : Support Routines (contd.)

```
struct symtab* symlook(char* s)
{struct symtab* sp = symboltab; int i = 0;
 while ((i < NSYMS) && (sp -> name != NULL))
  { if(strcmp(s,sp -> name) == 0) return sp;
    sp++; i++;
  }
 if(i == NSYMS) {
   yyerror("too many symbols"); exit(1);
 }
 else { sp -> name = strdup(s);
        return sp;
      }
}
```

Error Recovery in YACC

- In order to prevent a cascade of error messages, the parser remains in error state (after entering it) until three tokens have been successfully shifted onto the stack
- In case an error happens before this, no further messages are given and the input symbol (causing the error) is quietly deleted
- The user may identify **major** nonterminals such as those for **program**, **statement**, or **block**, and add error productions for these to the grammar
- Examples
  statement $\rightarrow$ **error** {action1}
  statement $\rightarrow$ **error** **';'** {action2}

YACC Error Recovery Example

```
%token DING DONG DELL
%start S
%%
S     :  rhyme{printf("string valid\n"); exit(0);}
rhyme : sound place
rhyme : error DELL{yyerror("msg1:token skipped");}
sound : DING DONG ;
place : DELL ;
place : error DELL{yyerror("msg2:token skipped");}
%%
```