Crafting a Compiler with C (X)

資科系
林偉川

Build RD Parser From LL(1) Table

- Non-terminal should have the form as

```
void non_term(void) { //non_term is the name of non-terminal
  token tok=next_token(); //return the look-ahead
  switch (tok) {
   case TERMINAL_LIST:  //list of terminal symbols
     parsing_actions(); break; //call to parsing procedures
    …
    default: syntax_error(); break;
  }
}
```

Generate RDP actions

- LL(1) tables are computed from a grammar when a parser is built
- The parsing decision recorded in LL(1) tables can be hardwired into the parsing procedures used by RDP
- gen_actions() takes the grammar symbols and generates the actions (call to parsing procedures for the non-terminal and match() for the terminal) to match them in a RDP

3

Generate RDP actions

- make_id() takes the name of a grammar symbol and transforms it into a valid program identifier
- make_id() involves stripping illegal characters, like < and >, removing embedded blanks, renaming characters ➔
  make_id("<statement list>") return "statement list"
  make_id(":=") return "COLONEQUAL"

4

Grammar data structure

```
typedef int symbol, terminal, nonterminal;
#define VOCABULARY   (N_T+N_NONT)
typedef stru gram {
  int terminal[N_T], nonterminal[N_NONT];
  int start_symbol, num_productions;
  stru prod {
    int lhs, rhs_lenght, rhs[MAX_RHS];
  } productions[NUM_P];
  char *name[VOCABULARY]; // production!!
  int vocabulary[VOCABULARY];
} grammar;
typedef stru prod production;
```

5

Algorithm to generate recursive descent actions

```
extern char *make_id(char *);
void gen_actions(symbol x[], int x_length) {
  int i; char *id;
  if (x_length == 0 ) printf("; /*  null */\n");
  else {
     for (i=0; i<x_length; i++) {
        id=make_id(g.names[x[i]]); //production
        if (is_terminal(x[i])) printf("\t\tmatch(%s);\n",id);
        else printf("\t\t%s();\n",id); //non-terminal
     }
  }
}
```

6

RDP using LL(1) table

- make_parsing_proc() takes a non-terminal and an LL(1) table and generates a complete parsing procedure for the non-terminal
- prods(A) returns the set of productions with A as the left-hand side
- rhs(P) returns the string of symbols that is the right-hand side of production P

7

RDP using LL(1) table

- make_parsing_proc() can be extended to generate more efficient parsing procedures for certain special cases
- If a non-terminal can be extended one way, there is no need to generate any conditional logic
- The body of the parsing procedure is gen_actions(rhs(p)), where p is the production to be matched

8

Algorithm to generate parsing procedures

```
void make_parsing_proc(const nonterminal A, const lltable T) {
  extern grammar g;  production p; terminal x; int i,j;
  printf("void %s(void) {\n", make_id(g.names[A]));
  printf("\ttoken tok = next_token();\n");
  printf("\tswitch (tok) {\n");
  for (i=0; i<g.num_productions; i++) {
    if (g.productions[i].lhs != A) continue;
    p=g.productions[i];
```

9

Algorithm to generate parsing procedures

```
    for (j=0; j<NUM_TERMINALS; j++) {
      x=g.terminals[j];
      if (T[A][x] == i) //production number
        printf("\tcase %s:\n", make_id(g.names[x]));
    }
    gen_actions(p.rhs, p.rhs_length); printf("\t\tbreak;\n");
  }
  printf("\tdefault:\n"); printf("\t\tsyntax_error(tok);\n");
  printf("\tbreak;\n\t}\n}\n");
}
```

10

One of the production example

<statement>→ID:=<expression> | **read (<id list>)** | **write (<expr list>)**

```
void statement(void) { // the generated result by make_parsing_proc()
  token tok;
  tok=next_token();
  switch (tok) {
  case ID: match(ID); mat(ASSIGNOP); expression();
               match(SEMICOLON); break;
 case READ: match(ID); mat(LPAREN); id_list();
                     match(RPAREN); match(SEMICOLON); break;
 case WRITE: match(WRITE); mat(LPAREN); expr_list();
                      match(RPAREN); match(SEMICOLON); break;
  default: syntax_error(tok); break;
  }
}
```

11

LL(1) parser driver

- RDP procedures are augmented with the code that performs semantic analysis and code generation
- Parsing procedures, once built and integrated into a compiler, are not easy to change
- The grammar that represents the syntax of a programming language must be update to accommodate new or modified constructs
- It is desirable to have a way of updating a parser with affecting other compiler components

12

LL(1) parser driver

- Rather than using the LL(1) table to build parsing procedures, it is possible to use the table in conjunction with a driver program to form an LL(1) parser
- LL(1) tables are computed only once when a parser is built. The tables are read-only by the LL(1) driver that uses them to control parsing

13

LL(1) Parser Driver

- The same driver is used with all LL(1) tables, changing a grammar and building a new parser is easy – new LL(1) tables are computed and substituted for the old tables
- LL(1) driver uses a stack rather then recursive procedure calls to store symbols to be matched, the resulting parser can be expected to be smaller and faster than a corresponding RDP

14

LL(1) parser driver

- The LL(1) parser driver stacks the symbols that are to be matched or expanded
- Terminal symbols on the stack must match an input symbol; non-terminal symbols are expanded using the LL(1) table
- Redo the RDP example and using the LL(1) parse table and driver
- Input data is **begin A:=BB-314+A; end$**

15

LL(1) parser driver

```
void lldriver(void) {
 push(s); // push the start symbol
 while (! stack_empty()) {
  if (is_nonterminal(x) && T[X][a]==X→Y1…Ym) {
   pop(1); push Ym … Y1 onto the stack; //expand
  }
  else if (X == a) { // X is terminal
    pop(1); scanner(&a); //get next token
  }
  else  //process syntax error
}}
```

16

Apply grammar with parsing table

Repeat (How to apply grammar with parsing table?)

let X be top stack symbol and a the next input symbol

if X is a terminal or $ then

if X = a then

pop X from the stack and remove a from the input

else Error() /* not match error */

else /* if X is a non-terminal */

if M[X][a] = $X \rightarrow Y_1 Y_2 \ldots Y_k$ then

begin pop X from the stack

push $Y_k Y_{k-1} \ldots Y_1$ onto the stack, Y_1 on top

end

else Error()

Until X=$

17

Reduced LL(1) Grammar

Start: <system goal> → <program> $

1. <program> → begin <statement list> end
2. <statement list> → <statement><statement tail>
3. <statement tail> → <statement><statement tail>
4. <statement tail> → λ
5. <statement> → ID := <expression>;
6. <statement> → read (<id list>);
7. <statement> → write (<expr list>);
8. <id list> → ID<id tail>
9. <id tail> → , ID<id tail>
10. <id tail> → λ
11. <expr list> → <expression> <expr tail>
12. <expr tail> → , <expression><expr tail>
13. <expr tail> → λ
14. <expression> → <primary><primary tail>
15. <primary tail> → <add op> <primary><primary tail>
16. <primary tail> →λ
17. <primary> → (<expression>)
18. <primary> → ID
19. <primary> → INTLITERAL
20. <add op> → PLUSOP
21. <add op> → MINUSOP
22. <system goal> → <program>$

18

Construct Parsing Table using Predict

	ID	INTLIT	:=	,	;	+	-	(	)	Begin	End	Read	Write	$
<program>										**1**				
<statement list>	**2**											**2**	**2**	
<statement>	**5**											**6**	**7**	
<statement tail>	**3**										**4**	**3**	**3**	
<expression>	**14**	**14**						**14**						
<id list>	**8**													
<expr list>	**11**	**11**						**11**						
<id tail>				**9**					**10**					
<expr tail>				**12**					**13**					
<primary>	**18**	**19**						**17**						
<primay tail>				**16**	**16**	**15**	**15**		**16**					
<add op>						**20**	**21**							
<system goal>										**22**				

19

LL(1) parser example

Begin A:=BB-314+A; end$ Start symbol is <system goal> !!!!

step	Parser action	Remaining input	Parse stack
1	**Predict 22**	**Begin A:=BB-314+A; end$**	**<system goal>**
2	**Predict 1**	**Begin A:=BB-314+A; end$**	**<program>$**
3	**Match**	**Begin A:=BB-314+A; end$**	**Begin A:=BB-314+A; end$**
4	**Predict 2**	**A:=BB-314+A; end$**	**<statement list>end$**
5	**Predict 5**	**A:=BB-314+A; end$**	**<statement><statement tail>end$**
6	**Match**	**A:=BB-314+A; end$**	**ID:=<exp>; <statement tail>end$**
7	**Match**	**:=BB-314+A; end$**	**:=<exp>; <statement tail>end$**
8	**Predict 14**	**BB-314+A; end$**	**<exp>; <statement tail>end$**
9	**Predict 18**	**BB-314+A; end$**	**<primary><primary tail>; <statement tail>end$**
10	**Match**	**BB-314+A; end$**	**ID<primary tail>; <statement tail>end$**

20

LL(1) parser example

step	Parser action	Remaining input	Parse stack
11	**Predict 15**	**-314+A; end$**	**<primary tail>; <statement tail>end$**
12	**Predict 21**	**-314+A; end$**	**<add op><primary><primary tail>; <statement tail>end$**
13	**Match**	**-314+A; end$**	**- <primary><primary tail>; <statement tail>end$**
14	**Predict 19**	**314+A; end$**	**<primary><primary tail>; <statement tail>end$**
15	**Match**	**314+A; end$**	**IntLiteral<primary tail>; <statement tail>end$**
16	**Predict 15**	**+A; end$**	**<primary tail>; <statement tail>end$**
17	**Predict 20**	**+A; end$**	**<add op><primary><primary tail>; <statement tail>end$**
18	**Match**	**+A; end$**	**+ <primary><primary tail>; <statement tail>end$**

LL(1) parser example

step	Parser action	Remaining input	Parse stack
19	**Predict 18**	**A; end$**	**<primary><primary tail>; <statement tail>end$**
20	**Match**	**A; end$**	**ID<primary tail>; <statement tail>end$**
21	**Predict 16**	**; end$**	**<primary tail>; <statement tail>end$**
22	**Match**	**; end$**	**;<statement tail>end$**
23	**Predict 4**	**end$**	**<statement tail>end$**
24	**Match**	**end$**	**end$**
25	**Match**	**$**	**$**

Homework Example

Begin

read(aa, bb);

c3:=(aa+bb) - (aa-bb);

write(aa, bb, c3);

End

How about start symbol is <system goal> !!!!

23

	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		
S	9			9		

1. E→TE'
2. E' → +TE'
3. E' → λ
4. T→FT'
5. T' → *FT'
6. T' → λ
7. F→(E)
8. F→ id
9. S→ E$

First(E)=First(T)=First(F)={ (, id }

First(E')= { +, λ }, First(T')= { *, λ }

Follow(E)=Follow(E')={), $ }

Follow(T)=Follow(T')={+,), $ } → First(E') U Follow(E')

Follow(F)={+, *,), $ }→First(T')U Follow(T')

How to process "id + id * id$"?

24

Stack	Input	Production	LL(1) table
S	id+id*id$		
$E	id+id*id$		M[E][id]
$E'T	id+id*id$	E→TE'	M[T][id]
$E'T'F	id+id*id$	T→FT'	M[F][id]
$E'T'id	+id*id$	F→id	
$E'T'	+id*id$	T' → λ	M[T'][+]
$E'	+id*id$	E' → +TE'	M[E'][+]
$E'T+	+id*id$		
$E'T	id*id$	T→FT'	M[T][id]
$E'T'F	id*id$	F→id	M[F][id]
$E'T'id	id*id$		
$E'T'	*id$	T' → *FT'	M[T'][*]
$E'T'F*	*id$		
$E'T'F	id$	F→id	M[F][id]
$E'T'id	id$		
$E'T'	$	T' → λ	M[T'][$]
$E'	$	E' → λ	M[E'][$]
$	$		

	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		
S	9			9		

1. E→TE'
2. E' → +TE'
3. E' → λ
4. T→FT'
5. T' → *FT'
6. T' → λ
7. F→(E)
8. F→ id
9. S→ E$

25

Homework Example

How about id*(id+id*id)$?

How about start symbol is **S→ E$**!!!!

26

Homework Example

Expr→- Expr | (Expr) | Var ExprTail

Var →ID VarTail

VarTail → (Expr) | λ

ExprTail→- Expr | λ

S➔Expr$

Trace the input of "ID- -ID((ID))"

27

Grammar with 9 action symbols

<program>→ #start begin <statement list> end
<statement list> → <statement> {<statement>}
<statement> → <ident>:=<expression> #assign
<statement> →read(<id list>);
<statement> → write(<expr list>);
<id list> → <ident> #read_id {, <ident> #read_id }
<expr list> → <expression> #write_id {, <expression> #write_id }
<expression> → <primary> { <add op><primary> #gen_infix }
<primary> → (<expression>)
<primary> → <ident>
<primary> → INTLITERAL #process_literal
<add op> → PLUSOP #process_op
<add op> → MINUS #process_op
<ident> → ID #process_id
<system goal>→ <program> SCANEOF #finish

28

LL(1) Action symbols

- Action symbol of the form #action_name are added to the LL(1) grammar to mark where the semantic actions are required
- Action symbols are not part of a grammar and are ignored when a LL(1) table is computed
- During parsing, the appearance of an action symbol in a production will serve to initiate the corresponding semantic action
- In the case of RDP, we will extend the gen_actions() routine to include action symbols as well as grammar symbols

29

LL(1) Action symbols

- When an action symbol is proceeded, it will be translated to a call to the corresponding semantic routine
  Ex. gen_actions(ID:=<expression>#gen_assign ;")

→match(ID); match(colonequal); expression();
gen_assign(); match(semicolon); →
corresponding to a semantic stack

30

LL(1) Action symbols

- Semantic stack is a separate data structure from the parse stack
- In RDP, the parse stack is hidden in the procedure call stack of the running compiler
- Action symbols make it easy to include semantic actions in RDP
- For LL(1) parsers, action symbols that appear in a particular production are pushed onto the stack when the production is predicted

31

LL(1) Action symbols

- When an action symbol appears at the top of the parse stack, it popped, and the appropriate semantic routine is called

32

LL(1) parser driver

```
void lldriver(void) {
  push(s); // push the start symbol
  while (! stack_empty()) {
    if (is_nonterminal(x) && T[X][a]==X→Y1…Ym) {
      pop(1); push Ym … Y1 onto the stack; //expand
    }
    else if (X == a) { // X is terminal
       pop(1); scanner(&a); //get next token
    }
    else  //process syntax error
}}
```

33

LL(1) parser driver

```
void lldriver(void) {
  push(s); // push the start symbol
  while (! stack_empty()) {
    if (is_nonterminal(x) && T[X][a]==X→Y1…Ym) {
      Replace X with Ym … Y1 on the stack; //expand
    }
    else if (is_terminal(X) && X == a) { // X is terminal
       pop(1); scanner(&a); //get next token
    }
    else if (is_action_symbol(X)) {
       pop(1); call_semantic_routine_corresponding _to_X; }
    else  //process syntax error
}}
```

34

Make grammars LL(1)

- LL(1) requires an unique prediction for each combination of non-terminal and look-ahead symbol
- Most LL(1) prediction conflicts can be grouped into 2 categories: common prefix and left recursive
- Simple grammar transformations that eliminate common prefixes and left recursion are reworking to transform most grammars into valid LL(1) form

35

Make grammars LL(1)

- Common prefix: 2 productions with the same left-hand side often have right-hand sides that share a common prefix (pair-wise disjoint!!)
- If-then-else to eliminate common prefix
  <stmt>→if <expr> then <stmt list> end if;
  <stmt>→if <expr> then <stmt list> else <stmt list> end if;
- The solution to predict conflicts caused by common prefixes is a simple factoring transform

36

Eliminate Common prefix

void factor(grammar *G) {

while (G has 2 or more productions with the same LHS and a common prefix) {

S={$A \rightarrow \alpha\beta$, …, $A \rightarrow \alpha\zeta$} be the set of productions with the same left-hand side, A, and a common prefix, α

Create a new non-terminal, N;

Replace S with the production set SET_OF($A \rightarrow \alpha N$, $N \rightarrow \beta$, …,$N \rightarrow \zeta$)

}}

37

If-then-else to eliminate common prefix

Using the previous if-then-else example, factor() produces

<stmt>→if <expr> then <stmt list> end if;

<stmt>→if <expr> then <stmt list> else <stmt list> end if;

➔

<stmt>→if <expr> then <stmt list> <if suffix>

<if suffix>→ end if;

<if suffix>→ else <stmt list> end if;

38

Remove left recursive

- The second category of conflicts, a production is left recursive if its left hand side symbol is also the first symbol of its right hand side
- Assume some look-ahead symbol t causes the prediction of some left recursive production $A \rightarrow A\beta$. After the prediction, A will again be the top stack symbol, and the same production would be predicted forever
- The algorithm remove_left_recursion() removes left recursion from a factored grammar

39

Eliminate left recursion

void remove_left_recursion(grammar *G) {

while (G contains a left-recursive non-terminal) {

$S=\{A \rightarrow A\alpha, A \rightarrow \beta, \ldots, A \rightarrow \zeta\}$ be the set of productions with the same left-hand side, A, where A is left recursive

Create 2 non-terminals, T and N;

Replace S with the production set
SET_OF($A \rightarrow$ **NT**, **N** $\rightarrow \beta, \ldots,$ **N** $\rightarrow \zeta$,
$T \rightarrow \alpha T$, $T \rightarrow \lambda$)

}}

40

Eliminate left recursive

E→E+T
E→T
T→T*P
T→P
P→ID

→

E→E_1 Etail
E_1→T
Etail→+T Etail
Etail→λ
T→ T_1 Ttail
T_1→P
Ttail→*P Ttail
Ttail→λ
P→ID

→

E→ T Etail
Etail→+T Etail
Etail→λ
T→ P Ttail
Ttail→*P Ttail
Ttail→λ
P→ID

41

Make grammar to be LL(1)

- Factoring and left recursion removal are the primary transforms used to make grammars LL(1)
- In rare cases, other transformations may be needed
- Consider the following grammar fragment that might appear in a language that allows identifiers as labels

1. <stmt>→<label><unlabeled stmt> **{ID}**
2. <label>→ID: **{ID}**
3. <label>→λ **{ID}**
4. <unlabel stmt>→ID:=<expr>; **{ID}**

42

Make grammar to be LL(1)

- There are no common prefixes in this fragment, and there is no left recursion, and the grammar is not LL(1)
- The problem is that ID predicts both <label> 2 and 3 productions
- The solution is to factor ID from the 2 and 4 productions, obtaining the following equivalent grammar, which is LL(1)

43

Implied common prefix convert to LL(1)

1. <stmt>→<label><unlabeled stmt> **{ID}**
2. <label>→ID: **{ID}**
3. <label>→λ **{ID}**
4. <unlabel stmt>→ID:=<expr>; **{ID}**

➔

<stmt>→ID<id suffix> **{ID}**

<id suffix>→:<unlabeled stmt> **{:}**

<id suffix>→:=<expr>; **{:=}**

<unlabel stmt>→ID:=<expr>; **{ID}**

44

Make grammar to be LL(1)

- In some cases more elaborate factoring may be needed
- In an Ada array declaration, array bounds can be declared as an explicit range pair or as the name of discreet type or subtype → A: array(I..J, Boolean).
- In defining an array bound we might write
  <array bound>→<expr>..<expr> {ID}
  <array bound>→ ID {ID}

45

Eliminate left recursive

<expr>→<expr>+<term>
<expr>→<term>
<term>→<term>*<factor>
<term>→<factor>
<factor>→ID

→

<expr>→<term><etail>
<etail>→+ <term><etail>
<etail>→λ
<term>→<factor><Ttail>
<Ttail>→*<factor><Ttail>
<Ttail>→λ
<factor>→ID

46

Make grammar to be LL(1)

- Because ID can be generated from <expr>, we have a prediction conflict
- Factoring ID from <expr> would be very tedious, giving the variety of expressions possible in Ada
- An alternative way is to write
  <array bound>→<expr><bound tail>
  <bound tail>→ ..<expr>
  <bound tail>→ λ
  <expr> → ID

47

Make grammar to be LL(1)

- If only a single <expr> appears, it must generate an ID. This can be checked during semantic processing for only an ID can name a type or subtype
- All grammars that include an endmarker can be rewritten into a form in which all right-hand sides begin with a terminal symbol; this form is called Greibach normal form
- Once a grammar is in Greibach normal form, factoring of common prefixes is easy
- There do exist language constructs that have no LL(1) grammar

48

Homework Which is LL(1)

E→Prefix (E) | V Tail
Prefix→F | λ
Tail→+ E | λ

S→ABc
A→a | λ
B→b | λ

S→aSe | B
B→bBe | C
C→cCe | d

S→Ab
A→a | B | λ
B→b | λ

S→ABBA
A→a | λ
B→b | λ

49

Homework to transform into LL(1)

DeclList→ DeclList ; Decl | Decl
Decl→IdList : Type
IdList→IdList , ID | ID
Type → ScalarType
Type → array (ScalarTypeList) of Type
ScalarType→ ID | Bound..Bound
Bound→Sign INTLIT | ID
Sign → + | - | λ
ScalarTypeList→ ScalarTypeList , ScalarType
ScalarTypeList→ ScalarType

50

If-then-else problem in LL(1)

- Dangling else problem → else can be optional
- The problem is that there may be more then parts than else parts, which means the matching of thens to elses may not be unique
- The same problem is similar to match [] which is not LL(1) and also not LL(K)
- BL={$[^i\ ]^j \mid i \geq j \geq 0$} the corresponding grammar is right
- CL generates an optional close bracket

S→ [S CL
S→λ
CL→] {]}
CL→λ {]}

51

Matching problem

- This grammar is ambiguous and can be modified to be the following grammar
- This grammar generates 0 or more unmatched ['s followed by 0 or more pairs of matched brackets

S→ [S {[} ∈ $first_1(S)$ → {[[} ∈ $first_2(S)$
S→ S1
S1→[S1] {[} ∈ $first_1(S1)$ → {[[} ∈ $first_2(S1)$
S1→λ
[or [[or [[[is accepted

52

Matching problem

- Matching [] grammar is pars-able using most bottom-up techniques such as SLR(1) is not LL(1)
- LL parsers cannot decide whether to predict a matched or unmatched open bracket
- Bottom-up parsers can delay announcing a production until an entire right-hand side is matched

53

Matching problem

- Top-down parser cannot delay – they must predict a production by having seen only the first (or first k) symbols derived from a right-hand side
- A technique used to handle the dangling else problem in LL(1) parsers is to use an ambiguous grammar along with some special case rules to resolve any non-unique predictions that arise

54

If-then-else LL(1) grammar

1. G→S;
2. S→if S E
3. S→Other
4. E→else S
5. E→λ

	if	else	Other	;
S	Predict 2		Predict 3	
E		Predict 4 Predict 5		Predict 5
G	Predict 1		Predict 2	

LL(1) table is not single-valued because it is ambiguous.
We must Make T[E][else]=predict 4.
Production 4 has priority over production 5!!

55

Matching problem

- To solve the dangling-else problem is not a grammar or parsing issue but rather a language design issue
- If all if statements are terminated with an endif or some equivalent symbol, the problem disappears

1.S→if S E
2.S→Other
3.E→else S endif
4.E→endif

	if	else	Other	endif
S	1		2	
E		3		4

56

EX. Grammar as followed:

S→iCtS | iCtSeS | a (if Conditional then Statement else)

C→ b

Can be converted into

S→iCtSS' | a — **First(S)={i,a}, First(S')={e, λ}**

S' → eS | λ — **Follow(S')={e, $}**

C→ b — **First(C)={b}**

	a	b	e	i	t	$
S	**S→a**			**S→iCtSS'**		
S'			**S'→ eS** **S' → λ**			**S' → λ**
C		**C→ b**				

omitted

57

Properties of LL(1) parser

- A correct, leftmost parse is guaranteed
  - Predict set is unique
- All grammars in LL(1) class are unambiguous
  - Predict set is unique
- All LL(1) parsers operate in linear time and, at most, linear space (relative to the length of the input being parsed)

58

Bottom-up parser

- LL parser cannot decide whether to predict a matched or unmatched open bracket!!!
- Bottom-up parser have an advantage– they can delay announcing a production until an entire right-hand side is matched
- Top-down methods cannot delay – they must predict a production by having seen only the first (or first k) symbols derived from a right-hand side
- A technique is used to handle the dangling else problem is to use an ambiguous LL(1) grammar along with some special case rules to resolve any non-unique predictions that arises

59

Bottom-up Parsing

•The parsing problem is finding the correct RHS re-witten as its corresponding LHS to get the previous sentential form in the derivation (rewritten shown in underline)

E→E+T | T , T→T*F | F, F→(E) | id ➔id+id*id

E => **E+T** => E+**T*F** => E+T***id** => E+**F***id =>

E+**id***id => **T**+id*id => **F**+id*id => **id**+id*id

60

Bottom-up Parsing

- The Bottom-up parser starts with the last sentential form and produces the sequence of sentential forms from there until all that remains is the start symbol
- The correct RHS in a given right sentential form to rewrite to get the previous right sentential form is called the handle
  - The Handle of a right sentential form is unique!!!
  - The bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar

61

Bottom-up Parsing

- Def: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S =>^*_{rm} \alpha A w =>_{rm} \alpha\beta w$ ➔ $=>^*_{rm}$ means zero or more rightmost derivation steps
- Def: β is a *phrase* of the right sentential form γ if and only if $S =>^* \gamma = \alpha 1 A \alpha 2 =>+ \alpha 1 \beta \alpha 2$ ➔ $=>_+$ means one or more rightmost derivation steps
- Def: β is a *simple phrase* of the right sentential form γ if and only if $S =>^* \gamma = \alpha 1 A \alpha 2 => \alpha 1 \beta \alpha 2$

62

Bottom-up Parsing

- In term of a parse tree, a phrase can be derived from a single non-terminal in one or more tree levels. But a simple phrase can be derived in just a single tree level. Sentential form : E → E + T * id 3 internal nodes→ 3 phrases→3 sub-tree's leaves
- phase : E + T * id(by E) , T * id(by T), id(by F)

simple phrase : id(by F)→ The simple phrases are a subset of the phrases

- The handle of a right sentential form is its leftmost simple phrase (id) → handle ⊆ simple phrase

63

Bottom-up Parsing

- Given a parse tree, it is now easy to find the handle (Have a grammar and draw a parse tree)

BUP v. s. a parse tree → the purpose of a parse is to produce a parse tree →find the handle to rewrite the sentence

- Parsing can be thought of as handle pruning (reduce)

- Handle can be pruned from the parse tree and the process repeated until to the root of the parse tree.

Then the entire rightmost derivation can be constructed

64

Shift-Reduce Algorithms

- **Shift is the action of moving the next input token to the top of the parser's stack ➔ like the PushDown**
  Automata as a language recognizer
  [use a pushdown stack as its memory]
- **Reduce is the action of replacing the handle(RHS) on the top of the parser's stack by its corresponding LHS**

65

Bottom-up Parsing (continued)

•The BUP examines one symbol of a input string at a time and left to right.
The input is treated as if it were stored in another stack
because the PDA never sees more than the leftmost symbol of the input string.
LR(Left-to-Right & Rightmost derivation) parser
- use a relative small amount of code and a parsing table
•LR parsers must be constructed with a tool (yacc in Linux) because the producing parsing table requires large amount of computer time and memory not suitable by hand-writing ➔ 2 variations on the (canonical LR table)

1. ***Require much less computer resources to produce parsing table***
2. ***Work on smaller classes of* grammars *than the canonical LR algorithm***

66

Advantages of LR parsers

- **They will work for nearly all grammars that describe programming languages.**
- **They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other BUP.**
- **They can detect syntax errors as soon as it is possible.**
- **The LR class of grammars is a superset of the class pars-able by LL parsers.**

67

Bottom-up Parsing (continued)

•*Knuth's insight in 1965*: A bottom-up parser could use the entire history of the parse(states), up to the current point, to make parsing decisions(canonical LR table)

•There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

68

LR parsing stack

Parse stack is shown as followed:

- **$S_0X_1S_1X_2S_2 \ldots X_mS_m$ (top)**
- **(S_i → state symbols, X_j → processed input symbols(terminals), a_j → waiting processed input symbols)**

69

LR parsing stack

- **The input string has been processing $a_0 \ldots a_{i-1}$**
- **($S_0X_1S_1X_2S_2 \ldots X_mS_m$, $a_ia_{i+1} \ldots a_n\$$)**
- **An LR configuration stores the state of an LR parser ($S_0X_1S_1X_2S_2 \ldots X_mS_m$, $a_ia_{i+1} \ldots a_n\$$) → $ at the end of input!!! ➔ A pair of string (stack, input)**

70

LR parsing table

- **LR parsers are table driven, where the table has two components, an ACTION table M and a GOTO table G**
  - **The ACTION table M specifies the action of the parser, given the parser state and the next token ➔Shift action**
  - **Rows are state names; columns are terminals(input symbol)**
  - **The parser Shifts the next input symbol onto the parse stack M[state, symbol]**

71

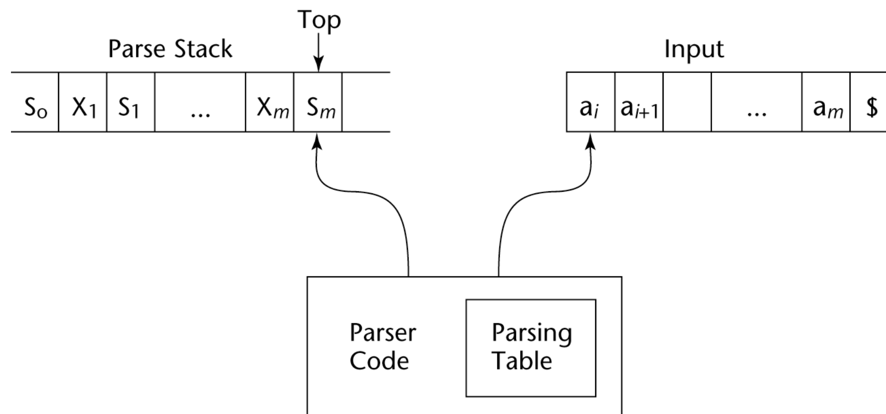
LR parsing table

- **The GOTO table G specifies which state to put on top of the parse stack after a reduction action is done ➔Reduce action**
- **Rows are state names; columns are non-terminals → Handle has been removed from the parse stack and the new non-terminal has been pushed onto the parse stack**

72

The structure of an LR parser

(S0X1S1X2S2…XmSm , aiai+1…an$)



73

Bottom-up Parsing (continued)

Initial configuration: (S_0, $a_1…a_n$$) *Parser actions*:

1. If ACTION[S_m, a_i] = Shift S, the next configuration is:($S_0X_1S_1X_2S_2…X_mS_ma_iS$, $a_{i+1}…a_n$$) ➔input symbol move into parse stack and move to next input

2. If ACTION[S_m, a_i] = Reduce A → β and S = GOTO[S_{m-r}, A], where r = the length of β, the next configuration is ($S_0X_1S_1X_2S_2…X_{m-r}S_{m-r}AS$, $a_ia_{i+1}…a_n$$) ➔ input symbol no change!!!

3. If ACTION[S_m, a_i] = Accept, the parse is comple and no errors were found.

4. If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine.

- A parser table can be generated from a given grammar with a tool, e.g., yacc in UNIX

E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

74

The LR parsing table for an arithmetic expression grammar

Shift and change to state 5

Use production 2 to reduce

state

State	id	+	*	(	)	$	E	T	F
	M Action						Goto **G**		
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Given a string id + id * id for parsing, the step of the LR Algorithm to apply the parsing table (include shift action table M and reduce GOTO G table) and is followed:

E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

Stack	Input	Output	
$0	id + id * id$	Shift 5	M[0,id]
$0id5	+ id * id$	Reduce 6	G[0, F]
$0F3	+ id * id$	Reduce 4	G[0, T]
$0T2	+ id * id$	Reduce 2	G[0, E]
$0E1	+ id * id$	Shift 6	M[1, +]
$0E1+6	id * id$	Shift 5	M[6, id]
$0E1+6id5	* id$	Reduce 6	G[6, F]
$0E1+6F3	* id$	Reduce 4	G[6, T]
$0E1+6T9	* id$	Shift 7	M[9, *]
$0E1+6T9*7	id$	Shift 5	M[7,id]
$0E1+6T9*7id 5	$	Reduce 6	G[7,F]
$0E1+6T9*7F10	$	Reduce 3	G[6,T]
$0E1+6T9	$	Reduce 1	G[0,E]
$0E1	$	Accept	

76