Compilation Principle
编 译 原 理

第8讲：语法分析(5)

张献伟

xianweiz.github.io

DCS290, 3/25/2021

Review Questions (1)

- What are the parts of a table-driven predictive parser?

  Input buffer, stack, parse table and a driver

- What are the operations on the stack?

  Expand the non-terminal, match the terminal

- How to predict the next production to use?

  Next input symbol, current nonterminal being processed

- What does LL(k) mean?

  L: scans the input from left to right

  L: produces a leftmost derivation

  k: using k input symbols of lookahead

- How to build the LL(1) parse table?

  Two sets: FIRST, FOLLOW

Review Questions (2)

- Which one is typically used, LL(0), LL(1), LL(2) …? Why not others?

  LL(1). LL(0) is too weak, LL(k) has a too large table

- Which are the key differences between top-down and bottom-up parsing?

  Top-down is based on leftmost derivation;
  bottom-up is the reverse of rightmost derivation.

- What are the key operations of bottom-up parsing?

  Shift: pushes a terminal on the stack
  Reduce: pops RHS and pushes LHS

Types of Bottom-Up Parsers

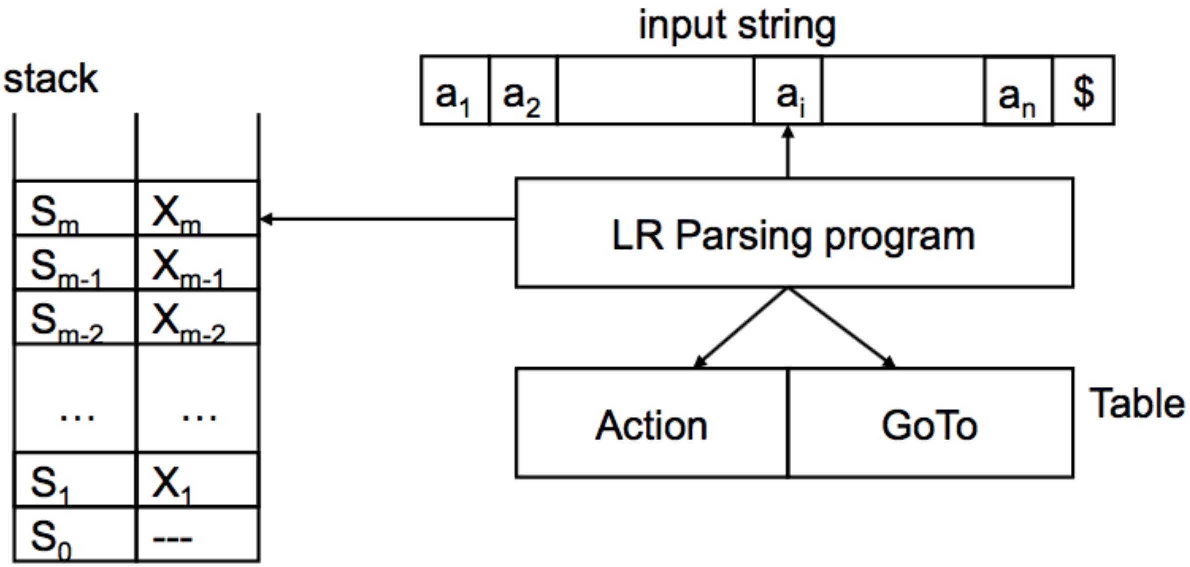
- Types of bottom up parsers
  - Simple precedence parsers
  - Operator precedence parsers
  - Recursive ascent parsers
  - LR family parsers
  - ...

- In this course, we will only discuss **LR family parsers**
  - Efficient, table-driven shift-reduce parsers
  - Most automated tools for bottom-up parsing generate LR family

LR(k) Parser

- **LR(k)**: member of LR family of parsers
  - L: scan input from left to right
  - R: construct a rightmost derivation in reverse
  - k: number of input symbols of lookahead to make decisions
    - k = 0 or 1 are of particular interests, is assumed to be 1 when omitted
- Comparison with LL(k) parser
  - Efficient as LL(k)
    - Linear in time and space to length of input (same as LL(k))
  - Convenient as LL(k)
    - Can generate automatically from grammar – YACC, Bison
  - More complex than LL(k)
    - Harder to debug parser when grammar causes conflicting predictions
  - More powerful than LL(k)
    - Handles more grammars: no left recursion removal, left factoring needed
    - Handles more (and most practical) languages: LL(1) ⊂ LR(1)

LR Parser



- The stack holds a sequence of states, $s_0s_1...s_m$ (s_m is the top)
  - States are to track where we are in a parse
  - Each grammar symbol X_i is associated with a state s_m
- Contents of stack + input ($X_1X_2...X_ma_i...a_n$) is a right sentential form
  - If the input string is a member of the language
- Uses $[S_m, a_i]$ to index into parsing table to determine action

Parse Table

- LR parsers use two tables: **action table** and **goto table**
  - The two tables are usually combined
  - Action table specifies entries for terminals
  - Goto table specifies entries for non-terminals

- Action table[动作表]
  - Action[*s*, *a*] tells the parser what to do when the state on top of the stack is *s* and terminal *a* is the next input token
  - Possible actions: **shift**, **reduce**, **accept**, **error**
- Goto table[跳转表]
  - Goto[*s*, *X*] indicates the new state to place on top of the stack after a reduction of the non-terminal *X* while state *s* is on top of the stack

Possible Actions[可能动作]

- **Shift**
  - Transfer the next input symbol onto the top of the stack
- **Reduce**
  - If there's a rule A → w, and if the contents of stack are qw for some q (q may be empty), then we can reduce the stack to qA
- **Accept**
  - The special case of reduce: reducing the entire contents of stack to the start symbol with no remaining input
  - Last step in a successful parse: have recognized input as a valid sentence
- **Error**
  - Cannot reduce, and shifting would create a sequence on the stack that cannot eventually be reduced to the start symbol

Possible Actions (cont.)

- Grammar

  S → E
  E → T | E + T
  T → id | (E)

- Input: (id + id)
  - #(id + id)$ => (id#+ id)$ => (T#+ id)$ => (E#+ id)$ => (E+id#)$ => (E+T#)$ => (E#)$ => (E)#$ => T#$ => E#$ => S#$

- Input: id+)
  - #id+)$ => id#+)$ => T#+)$ => E#+)$ => E+#)$ …

Example: Parse Table

Grammar:

(1) S → BB

(2) B → aB

(3) B → b

String: bab

State	**ACTION**			**GOTO**	
	a	**b**	**$**	**S**	**B**
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Table entry:
  - s*i*: shifts the input symbol and moves to state *i* (i.e., push state on stack)
  - r*j*: reduce by production numbered *j*
  - acc: accept
  - blank: error

Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	**b**	**$**	**S**	**B**
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

state → 0 4

b a b

symbol → $ b

b a b $

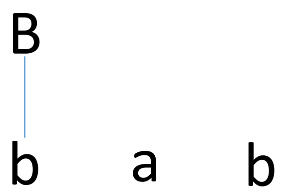
Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	**b**	**$**	**S**	**B**
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



state → 0 4

symbol → $ bB

a b $

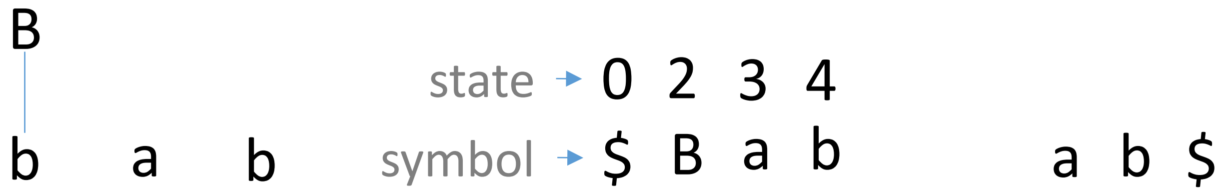
Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



B

b a b

state → 0 2 3 4

symbol → $ B a b

a b $

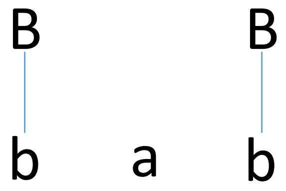
Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



state → 0 2 3 4 6

symbol → $ B a b B $

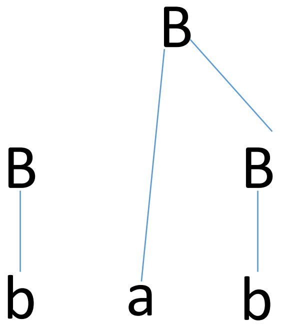
Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	**b**	**$**	**S**	**B**
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



state → 0 2 3 6

symbol → $ B B B $

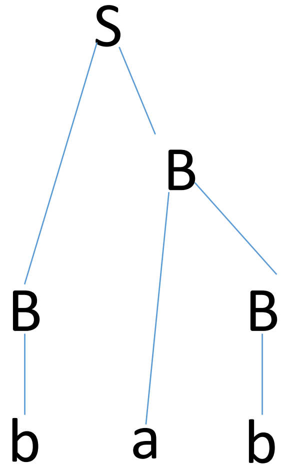
Example: Parse Table (cont.)

Grammar:

(1) S → BB
(2) B → aB
(3) B → b

String: bab

State	ACTION			GOTO	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



state → 0 2 1 5

symbol → $ B S B $

Parser Actions

Initial	s_0 $\$$	$a_1a_2 \ldots a_n\$$
General	$s_0s_1 \ldots s_m$ $\$X_1 \ldots X_m$	$a_ia_{i+1} \ldots a_n\$$

- If ACTION[s_m, a_i] = sx, then do shift
  - Pushes a_i on stack
    - a_i is removed from input
  - Enters state x
    - i.e., pushes state x on stack

$s_0s_1 \ldots s_mx$ $\$X_1 \ldots X_ma_i$	$a_{i+1} \ldots a_n\$$

Parser Actions (cont.)

Initial

s_0

$\$$ $a_1a_2\ldots a_n\$$

General

$s_0s_1 \ldots s_m$

$\$X_1\ldots X_m$ $a_ia_{i+1}\ldots a_n\$$

- If ACTION[s_m, a_i] = rx, (i.e., the x^{th} production: $A \rightarrow X_{m-(k-1)}\ldots X_m$), then do reduce
  - Pops *k* symbols from stack
  - Pushes *A* on stack
  - No change on input
  - GOTO[S_{m-k}, A] = y, then

$s_0s_1 \ldots s_{m-k}$

$\$X_1\ldots X_{m-k}A$ $a_ia_{i+1}\ldots a_n\$$

$s_0s_1 \ldots s_{m-k}y$

$\$X_1\ldots X_{m-k}A$ $a_ia_{i+1}\ldots a_n\$$

Parser Actions (cont.)

Initial	s_0 $\$$	$a_1a_2\ldots a_n\$$
General	$s_0s_1 \ldots s_m$ $\$X_1\ldots X_m$	$a_ia_{i+1}\ldots a_n\$$

- If ACTION[s_m, a_i] = acc, then parsing is complete
- If ACTION[s_m, a_i] = <empty>, then report error and stop

LR Parsing Program

- **Input**: input string ω and parse table with ACTION/GOTO
- **Output**: reduction steps ω's bottom-up parse, or error
- **Initial**: s_0 on the stack, ω$ in the input buffer

```
let a be the first symbol of ω$
while (1) { /* repeat forever */
    let s be the state on top of the stack;
    if (ACTION[s,a] = shift t) {
        push t onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A -> β) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A-> β;
    } else if (ACTION[s,a] = accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

Construct Parse Table

- Construct parsing table: identify the possible states and arrange the transitions among them
- **LR(0)** parsing
  - Simplest LR parsing, only considers stack to decide shift/reduce
  - Weakest, not used much in practice because of its limitations
- **LR(1)** parsing
  - LR parser that considers next token (lookahead of 1)
  - Compared to LR(0), more complex alg and much bigger table
- **SLR(1)** parsing
  - Simple LR, lookahead from first/follow rules derived from LR(0)
  - Keeps table as small as LR(0)
- **LALR(1)** parsing
  - Lookahead LR(1): fancier lookahead analysis using the same LR(0) automaton as SLR(1)

Item[项目]

- An **item** is a production with a "·" somewhere on the RHS
  - Dot indicates extent of RHS already seen in the parsing process
  - The only item for X → ε is X → ·
  - Items are often called "**LR(0) items**" (a.k.a., **configuration**)
- The items for A → XYZ are
  - A → ·XYZ
    - Indicates that we hope to see a string derivable from XYZ next on input
  - A → X·YZ
    - Indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ
  - A → XY·Z
  - A → XYZ·
    - Indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A

State[状态]

- Example:
  - Suppose we are currently in this position

    A → X·YZ
  - We have just recognized X and expect the upcoming input to contain a sequence derivable from YZ (say, Y → u|w)
    - Y is further derivable from either u or w

      A → X·YZ

      Y → ·u

      Y → ·w
  - The above three items can be placed into a set, called as **configuration set** of the LR parser
- Parsing tables have one **state** corresponding to each set
  - The states can be modeled as a finite automaton where we move from one state to another via transitions marked with a symbol of the CFG

Augmented Grammar[增广文法]

- We want to start with an item with a dot before the start symbol *S* and move to an item with a dot after *S*
  - Represents shifting and reducing an entire sentence of the grammar
  - Thus, we need *S* to appear on the right side of a production
  - Only one 'acc' in the table
- Modify the grammar by adding the production

  S' → ·S

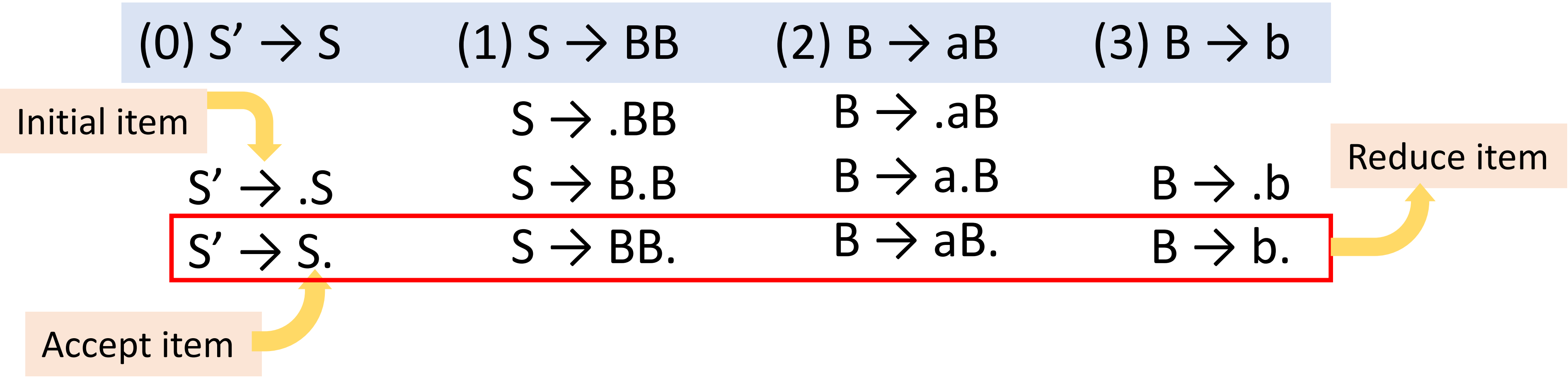
Grammar:

(1) E → E + T
(2) E → T
(3) T → T * F
… …

Augmented grammar:

(0) E' → E
(1) E → E + T
(2) E → T
(3) T → T * F
… …

Example



(0) S’ → S	(1) S → BB	(2) B → aB	(3) B → b
	S → .BB	B → .aB	
S’ → .S	S → B.B	B → a.B	B → .b
S’ → S.	S → BB.	B → aB.	B → b.

Initial item

Reduce item

Accept item

- **Closure**: the action of adding equivalent items to a set
  - Example: S’ → .S S → .BB B → .aB B → .b
- Intuitively, A → α.Bβ means that we might next see a <u>substring derivable from Bβ</u> (_*sub*) as input. The _*sub* will have a prefix derivable from B by applying one of the B-productions.
  - Thus, we add items for all the B-productions, i.e., if B → γ is a production, we add B → .γ in the closure