Compiler Design

Bottom Up Parsing

Amey Karkare
Department of Computer Science and Engineering
IIT Kanpur
karkare@iitk.ac.in

Parsing

- Process of determination whether a string can be generated by a grammar
- Parsing falls in two categories:
  - Top-down parsing:
    Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals)
  - Bottom-up parsing:
    Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol)

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root OR
- Reduce a string w of input to start symbol of grammar

Consider a grammar

S → aABe
A → Abc | b
B → d

And reduction of a string

a b b c d e
a A b c d e
a A d e
a A B e
S

The sentential forms happen to be a *right most derivation in the reverse order.*

S → a A B e
→ a A d e
→ a A b c d e
→ a b b c d e

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is .w

Shift reduce parsing …

- Bottom up parsing has two actions
- Shift: move terminal symbol from right string to left string

  if string before shift is α.pqr

  then string after shift is αp.qr

Shift reduce parsing …

- Reduce: immediately on the left of "." identify a string same as RHS of a production and replace it by LHS

if string before reduce action is $\alpha\beta$.pqr

and $A \rightarrow \beta$ is a production

then string after reduction is αA.pqr

Example

Assume grammar is E → E+E | E*E | id

Parse id*id+id

Assume an oracle tells you when to shift / when to reduce

String	**action (by oracle)**
.id*id+id	shift
id.*id+id	reduce E→id
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce E→id
E*E.+id	reduce E→E*E
E.+id	shift
E+.id	shift
E+id.	Reduce E→id
E+E.	Reduce E→E+E
E.	ACCEPT

Shift reduce parsing …

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - Which production to use for reduction?
- Sometimes parser can reduce but it should not:
  - $X \rightarrow \epsilon$ can always be used for reduction!

Issues in bottom up parsing

- Sometimes parser can reduce in different ways!
- Given stack δ and input symbol a, should the parser
  - Shift a onto stack (making it δa)
  - Reduce by some production $A \rightarrow \beta$ assuming that stack has form $\alpha\beta$ (making it αA)
  - Stack can have many combinations of $\alpha\beta$
  - How to keep track of length of β?

Handles

- The basic steps of a bottom-up parser are
  - to identify a *substring* within a *rightmost sentential* form which matches the RHS of a rule.
  - when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost-sentential form.
- Such a substring is called a *handle*

Handle

- A *handle* of a right sentential form γ is
  - a production rule $A \rightarrow \beta$, and
  - an occurrence of a sub-string β in γ

such that

- when the occurrence of β is replaced by A in γ, we get the previous right sentential form in a rightmost derivation of γ.

Handle

Formally, if

$S \Rightarrow^{rm*} \alpha A w \Rightarrow^{rm} \alpha\beta w$,

then

- β in the position following α,
- and the corresponding production $A \rightarrow \beta$

is a handle of $\alpha\beta w$.

- The string w consists of only terminal symbols

13

Handle

- We only want to reduce handle and not any RHS
- Handle pruning: If β is a handle and A → β is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning.

14

Handle: Observation

- *Only terminal symbols can appear to the right of a handle in a rightmost sentential form.*
- Why?

15

Handle: Observation

Is this scenario possible:

- $\alpha\beta\gamma$ is the content of the stack
- $A \rightarrow \gamma$ is a handle
- The stack content reduces to $\alpha\beta A$
- Now $\mathrm{B} \rightarrow \beta$ is the handle

In other words, handle is not on top, but buried *inside* stack

Not Possible! Why?

16

Handles …

- Consider two cases of right most derivation to understand the fact that handle appears on the top of the stack

$$S \to \alpha Az \to \alpha\beta Byz \to \alpha\beta\gamma yz$$
$$S \to \alpha BxAz \to \alpha Bxyz \to \alpha\gamma xyz$$

17

Handle always appears on the top

Case I: $S \to \alpha Az \to \alpha\beta Byz \to \alpha\beta\gamma yz$

stack	input	action
αβγ	yz	reduce by B→γ
αβB	yz	shift y
αβBy	z	reduce by A→ βBy
αA	z	

Case II: $S \to \alpha BxAz \to \alpha Bxyz \to \alpha\gamma xyz$

stack	input	action
αγ	xyz	reduce by B→γ
αB	xyz	shift x
αBx	yz	shift y
αBxy	z	reduce A→y
αBxA	z	

18

Shift Reduce Parsers

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- Bottom up parsing is essentially the process of detecting handles and reducing them.
- Different bottom-up parsers differ in the way they detect handles.

19

Conflicts

- What happens when there is a choice
  - What action to take in case both shift and reduce are valid?

    shift-reduce conflict
  - Which rule to use for reduction if reduction is possible by more than one rule?

    reduce-reduce conflict

20

Conflicts

- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

Shift reduce conflict

Consider the grammar E → E+E | E*E | id

and the input id+id*id

stack	input	action
E+E	*id	reduce by E→E+E
E	*id	shift
E*	id	shift
E*id		reduce by E→id
E*E		reduce byE→E*E
E		

stack	input	action
E+E	*id	shift
E+E*	id	shift
E+E*id		reduce by E→id
E+E*E		reduce by E→E*E
E+E		reduce by E→E+E
E		

Reduce reduce conflict

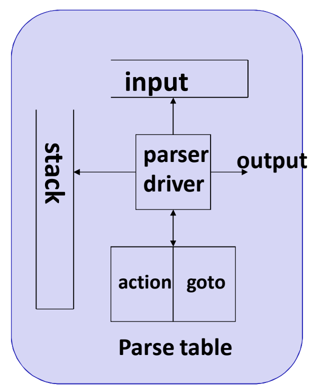
Consider the grammar M → R+R | R+c | R
R → c

and the input c+c

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by R→c
R+R		reduce by M→R+R
M		

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by M→R+c
M		

LR parsing



- Input buffer contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2 \ldots\ldots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.
- Table contains action and goto parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

Example

Consider a grammar and its parse table

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

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

action

goto

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_i is current input symbol

- Action $[S_i, a_i]$ can have four values
  1. sj: shift a_i to the stack, goto state S_j
  2. rk: reduce by rule number k
  3. acc: Accept
  4. err: Error (empty cells in the table)

Driving the LR parser

Stack: $S_0X_1S_1X_2 \ldots X_mS_m$ Input: $a_ia_{i+1} \ldots a_n\$$

- If action$[S_m, a_i]$ = shift S
  Then the configuration becomes
  Stack: $S_0X_1S_1 \ldots \ldots X_mS_ma_iS$ Input: $a_{i+1} \ldots a_n\$$

- If action$[S_m, a_i]$ = reduce $A \rightarrow \beta$
  Then the configuration becomes
  Stack: $S_0X_1S_1 \ldots X_{m-r}S_{m-r}AS$ Input: $a_ia_{i+1} \ldots a_n\$$
  Where r = $|\beta|$ and S = goto$[S_{m-r}, A]$

Driving the LR parser

Stack: $S_0X_1S_1X_2 \ldots X_mS_m$ Input: $a_ia_{i+1} \ldots a_n\$$

- If action$[S_m, a_i]$ = accept
  Then parsing is completed. HALT

- If action$[S_m, a_i]$ = error (or empty cell)
  Then invoke error recovery routine.

Parse: id + id * id

1. E→ E + T
2. E→ T
3. T→ T * F
4. T→ F
5. F→ (E)
6. F→ id

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

Parse id + id * id

Stack	Input	Action
0	id+id*id$	shift 5
0 id 5	+id*id$	reduce by F→id
0 F 3	+id*id$	reduce by T→F
0 T 2	+id*id$	reduce by E→T
0 E 1	+id*id$	shift 6
0 E 1 + 6	id*id$	shift 5
0 E 1 + 6 id 5	*id$	reduce by F→id
0 E 1 + 6 F 3	*id$	reduce by T→F
0 E 1 + 6 T 9	*id$	shift 7
0 E 1 + 6 T 9 * 7	id$	shift 5
0 E 1 + 6 T 9 * 7 id 5	$	reduce by F→id
0 E 1 + 6 T 9 * 7 F 10	$	reduce by T→T*F
0 E 1 + 6 T 9	$	reduce by E→E+T
0 E 1	$	ACCEPT

Configuration of a LR parser

- The tuple

  <Stack Contents, Remaining Input>

  defines a *configuration* of a LR parser
- Initially the configuration is

  $< S_0 , \quad a_0 a_1 \ldots a_n \$ >$
- Typical final configuration on a successful parse is

  $< S_0 X_1 S_i , \quad \$ >$

LR parsing Algorithm

Initial state: Stack: S_0 Input: w$

```
while (1) {
   if (action[S,a] = shift S') {
      push(a); push(S'); ip++
   } else if (action[S,a] = reduce A→β) {
      pop (2*|β|) symbols;
      push(A); push (goto[S'',A])
       (S'' is the state at stack top after popping symbols)
   } else if (action[S,a] = accept) {
       exit
   } else { error }
```