8 Shift-Reduce Parsing

Monday, September 23, 2024 12:16 PM

Specification → Context Free Grammar.

Recognition → Parsing

$w \in L$?

- THE "PARSING" PROBLEM:

recognize a language specified by a C.F. Grammar.

$w \in L$? How? → build a derivation for w
→ construct the parse tree for w.

- TYPES OF PARSERS:
  - Top-Down Parsers
    - Construct the parse tree from root to leafs.

    e.g. "Recursive Descent Parser"

  - Bottom-up Parsers.
    - Construct the parse tree from leafs to root

    e.g. "Shift-Reduce Parser"

- THE SHIFT-REDUCE PARSER D. Knuth

  - Reads input from Left to Right
  - Produces Rightmost Derivation.

    Called LR-Parser

    classification; LR(K) : K is a number K is the number of symbols from the input the algorithm needs to work.

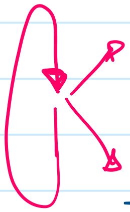
  - LR(1)
  - Not a general parser for C.F. Grammars

LR(1)

- Not a general parser for C.F. Grammars
  Limited to a subclass :- LR-Grammars.

• Intuition:

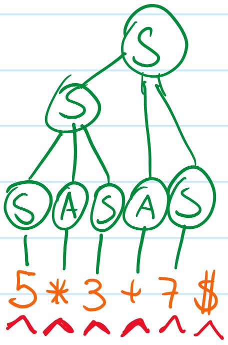
Iteratively; do one of two things:



- Shift :- read the next symbol from the input
- Reduce :- build a branch of the parse tree. build an intermediate node i.e. take nodes that match the body of a rule in the grammar, and connect them to their "parent", the head of the rule.

E.G. $S \to SAS \mid 5 \mid 3 \mid 7$

$A \to + \mid *$



• CONFLICTS IN A SHIFT-REDUCE PARSER

- Reduce-Reduce Conflict

$S \to A \mid B$

$A \to a$

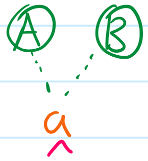
$B \to a$

Derivation

S

A

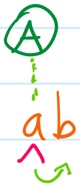
a



- Shift-Reduce Conflict

$S \to ab \mid Ab$

$A \to a$



- SHIFT-REDUCE TRACE
    - An implementation will simulate a push-down automata
    - The algorithm Constructs Tables from the grammar.
        - ↳ how to manage the stack.
        - ↳ when to "shift", when to "Reduce"

Pseudocode

```
FUNCTION shift-reduce ()
  stack.push(0)  // 0 is the start state
  input := w$    // $ marks end of input
  x := first symbol in w

  WHILE ~stop DO
    s := top of stack
    IF action[s,x] = shift t
      x := next input symbol
      stack.push(t)

    ELSIF action[s,x] = reduce t   // A → β
      pop len(β) symbols from stack
      t := stack.top()
      stack.push( goto[t,A] )
      print( A → β )

    ELSIF action[s,x] = accept
      stop := True
```

E.G

1 $S \to CC$
2 $C \to aC$
3 $C \to b$

Action:

symbols

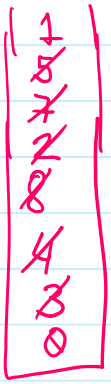
states of p.d.a

	a	b	$
0	S3	S4	
1			acc
2	S6	S7	
3	S3	S4	
4	R3	R3	
5			R1
6	S6	S7	
7			R3
8	R2	R2	
9			R2

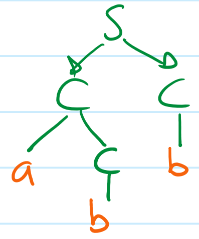
non-terminal symbols.

	S	C
0	1	2
1		
2		5
3		8
4		
5		
6		9
7		
8		
9		

word = abb$



$C \to b$
$C \to aC$
$C \to b$
$S \to CC$



—o— EOF.