8 Shift-Reduce Parsing

Wednesday, February 25, 2026 3:26 PM

Specification :- Context Free Grammars

Recognition :- Parser*

$w \in L$?

- THE "PARSING" PROBLEM:

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

$w \in L$? How → build a derivation of w
↳ build the parse tree of w

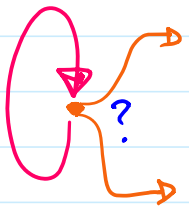
- TYPES OF PARSERS:
  - Top-Down Parsers
    - Construct Parse Tree from root to leaves.
    e.g. Recursive Descent Parser
  - Bottom-Up Parsers.
    - Construct Parse tree from leaves to root
    e.g "Shift-Reduce" Parser.

- THE SHIFT-REDUCE PARSES: D. Knuth
  - Read input from Left-to-Right
  - Produces Rightmost-Derivation
  Called LR-parsers.
  classification: LR(K): K is a number
  K is the number of symbols from the input the parser needs to operate.
  - LR(1)

the parser needs to operate.

- LR(1)
- Not a general solution for C.F. Grammars.
  Limited to a sublass of grammars:- LR-Grammars.

- Intuition.
  Iterative algorithm

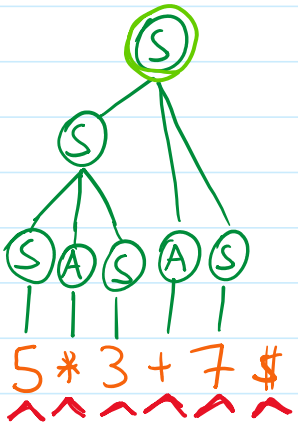


Shift:- read the next symbol from the input

Reduce:- build a branch of the parse tree
build an intermediate node
take nodes that match the body of a rule in the grammar and connect them to a new "parent" node labeled by the head.

E.g.

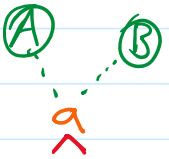
$S \to SAS \mid 5 \mid 3 \mid 7$
$A \to + \mid *$



- 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

$S \to ab \mid Ab$
$A \to a$

S
ab

S
Ab

S → ab | Ab
A → a

S
ab

S
Ab
ab

A
ab

- SHIFT-REDUCE TRACE:
  - An implementation of a shift-reduce parser corresponds to a push-down automata.
  - The algorithm generates 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 → CC
2 C → aC
3 C → b

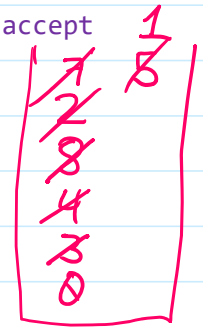
S
CC
aCC
abb

Action.

states	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

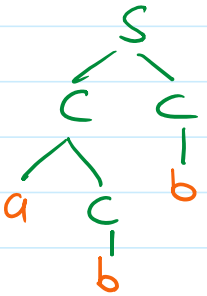
Goto

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



w = abb $

C → b
C → aC
C → b
S → CC



EOF