8 Shift-Reduce Parsing

Wednesday, February 26, 2025 12:30 PM

Specification:- Context Free Grammar.

Recognition:- Parsers

$w \in L$?

- THE "PARSING" PROBLEM:

  How to recognize a language specified by a CFG?

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

- TYPES OF PARSERS:
  - Top-Down Parsers:
    - Constructs parse-tree from root to leafs.
      e.g. "Recursive Descent Parser"
  - Bottom-Up Parsers:
    - Construct 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 read to work.

input the algorithm needs to read to work.

- LR(1)
- Not a general parser for C.F. Grammars
  Limited to a subclass:- LR-Grammars.

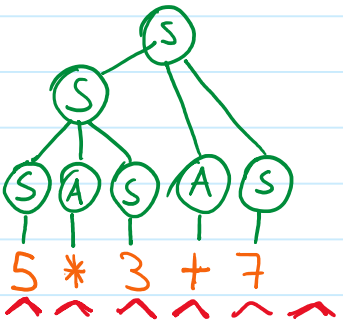
- Intuition:
  Iteratively, do one of two things
  - Shift :- read next symbol from the input
  - Reduce :- build a branch in the parse tree
    build an intermediate node
    take the branches that match a body of a grammar rule, and connect them to their "parent": the head of the rule

E.G.

$S \to SAS \mid 5 \mid 3 \mid 7$

$A \to + \mid *$

$w = 5 * 3 + 7$

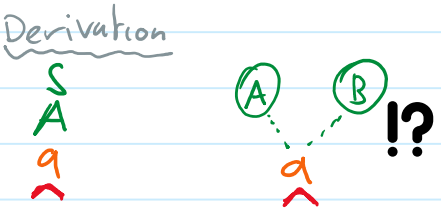


- "CONFLICTS" IN A SHIFT-REDUCE PARSER
  - "Reduce-Reduce" Conflict

    $S \to A \mid B$

    $A \to a$

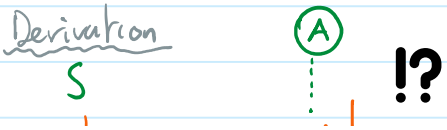
    $B \to a$



  - "Shift-Reduce" Conflict.

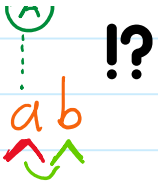
    $S \to ab \mid Ab$

    $A \to a$



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

derivation
S
ab



!?

- SHIFT-REDUCE TRACE
  - An implementation will simulate a push-down automata.
    stack set of states.
  - The Algorithm construct 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) from stack
      t := stack.pop()
      stack.push( goto[t,A] )
      print( A → β )

    ELSIF action[s.x] = accept
      stop := True

   RETURN accept
```

E.G.

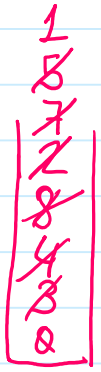
1.- $S \to CC$
2.- $C \to aC$
3.- $C \to b$

ACTION (symbols)

state	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

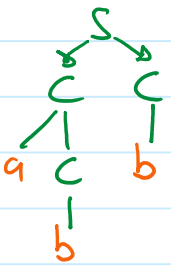
GOTO (non-terminal symbol)

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

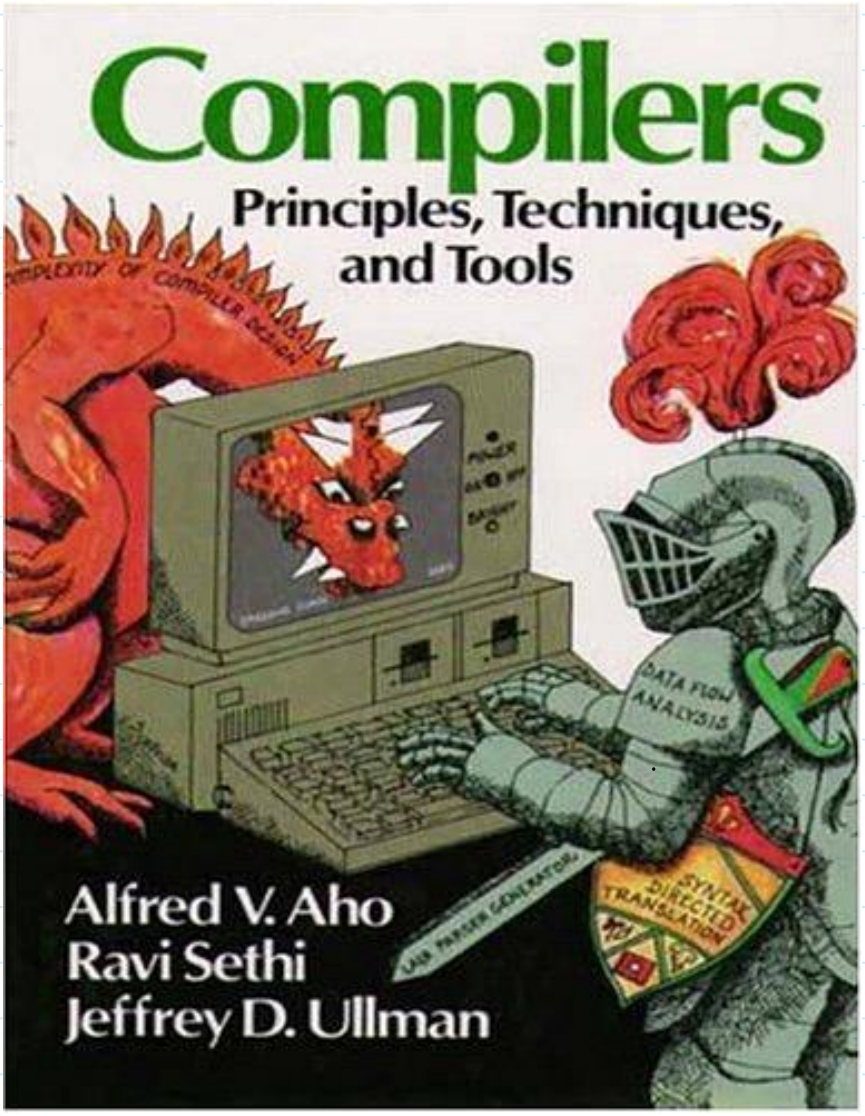


Word: abb$

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



—o— EOF.


Compilers
Principles, Techniques, and Tools
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman