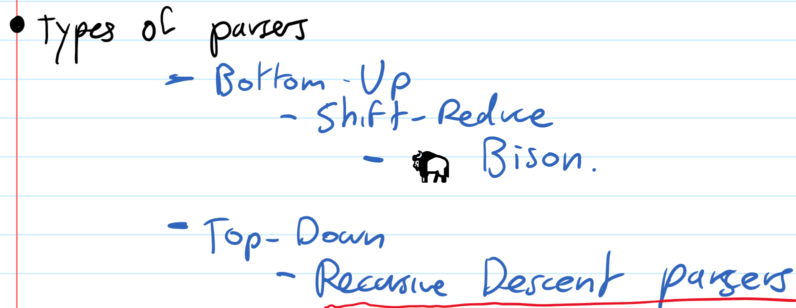
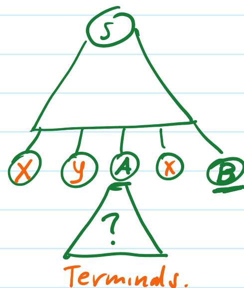
10 Recursive Descent Parsing

Monday, March 4, 2024 12:21 PM



- Types of parsers
  - Bottom-Up
    - Shift-Reduce
      - Bison.
  - Top-Down
    - Recursive Descent parsers

- Top-Down Parsers
  - Read input from Left-to-Right
  - Produce a Leftmost-Derivation
    - called: LL-Parsers.



  - Most common types:
    - Recursive Descent - Directly encodes grammar rules into code functions
    - "Predictive parse tables" - table driven Algorithm.
  - Limitations.
    - Not all Contex-Free-Grammars can be Parsed.
    - Limited to a subclass of grammars. called LL-grammars.

- RECURSIVE DESCENT PARSING:

assume: - global variable token
- Function gettoken() reads a new terminal from input and stores it in token.

Recipe: (to encode a grammar)
- one function per non-terminal symbol

$A \to \alpha \beta \Gamma$
$A \to xyz$

parse_A()

- If a non-terminal has more than one body the token should determine which rule to apply
- For each symbol α in the body
  - if α is a terminal symbol compare to token if the same, consume token
  - if α is a non-terminal symbol call function parse_α()

E.G. #0

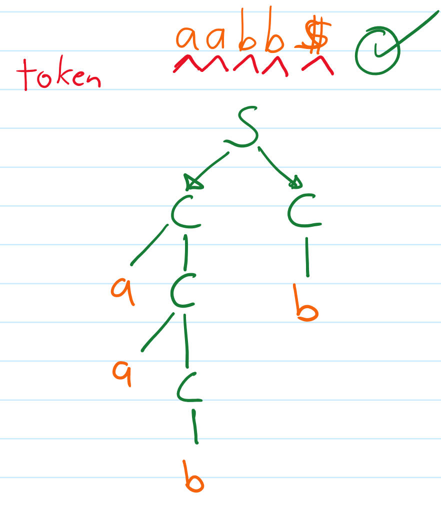
$S \to CC$
$C \to b$
$C \to aC$

```
FUNCTION parse_C()
  IF  token = 'b'  THEN
    getToken()
  ELSIF  token = 'a'  THEN
    getToken()
    parse_C()
  ELSE
    error()
  END
END.
```

```
FUNCTION parse_S()
  parse_C()
  parse_C()
END.
```

- Starting the parser.

```
FUNCTION main()
  getToken()
  parse_S()
  IF  token # '$'  THEN
    error("expected end of input")
  END
END.
```



E.G. #1

$> S \to dAc \mid b$
$A \to baB \mid \lambda$
$B \to aS$

```
FUNCTION parse_S()
  IF  token = 'd' THEN
    getToken()
    parse_A()
    IF  token = 'c'  THEN
      getToken()
    ELSE
       error("Expecting 'c'")
  ELSIF  token = 'b' THEN
    getToken()
  ELSE
    error("Expecting 'd' or 'b'")
END.
```

```
FUNCTION parse_A()
  IF  token = 'b'  THEN
    getToken()
    IF  token = 'a'  THEN
      getToken()
      parse_B()
    ELSE
      error("Expecting 'a'")
END.
```

```
FUNCTION parse_B()
  IF  token = 'a' THEN
    getToken()
    parse_S()
  ELSE
```

```
    getToken()
    parse_S()
  ELSE
    error()
END.
```

```
  ELSE
    error("Expecting 'd' or 'b'")
END.
```

Trace:

cba $ — reject.

dc $ — accept.

dbaac $
parse_S()
  parse_A()
    parse_B()
      parse_S()
reject.

dbaadcc $
parse_S()
  parse_A()
    parse_B()
      parse_S()
        parse_A()
accept.

bbb $ — reject.

- EXTENDED BNF

We extend now our grammar format with 2 new shorthands

[ ] option

{ } repetition

Eg:

A → a [b] c ≡ A → abc
A → ac

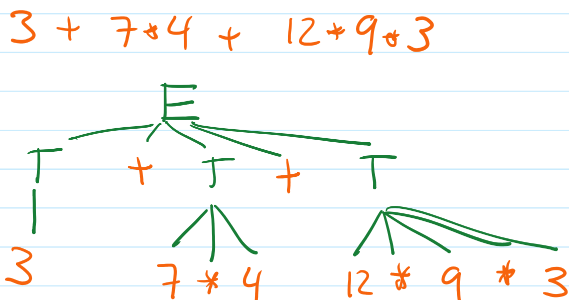
B → a {b} c ≡ B → ac
B → abc
B → abbc
B → abbbc
⋮

zero or more repetitions of b

E.g:

E → T {+ T}
T → int {* int}

3 + 7*4 + 12*9*3



Encoding:

```
FUNCTION parse_E()
  parse_T()
  WHILE  token = '+'  DO
```

sentinel

```
FUNCTION parse_T()
  parse_T_INT()
  WHILE  token = '*'  DO
```

```
FUNCTION parse_E()
  parse_T()
  WHILE  token = '+'  DO
    getToken()
    parse_T()
  END
END.
```

sentinel ('+')

{+T}

```
FUNCTION parse_T()
  parse_T_INT()
  WHILE  token = '*'  DO
    getToken()
    parse_T_INT()
  END
END.
```

E.G

S → if C then B [else B] end.

sentinel ([else)

```
FUNCTION parse_S()
  IF  token = "if" THEN
    getToken()
    parse_C()
    IF  token = "then"  THEN
      getToken()
      parse_B()

      IF  token = "else" THEN
        getToken()
        parse_B()
      END

      IF  token = "end" THEN
        getToken()
      ELSE
        error("unterminated if")
      END
    ELSE
      error("then expected")
  ELSE
    error("if expected")
  END
END.
```

E.G.

S → if C then B {elsif C then B} [else B] end.

EOF