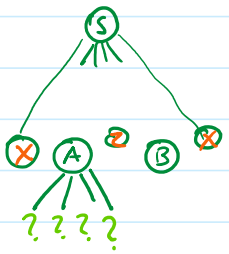
10 Recursive Descent Parser

Friday, March 7, 2025 12:33 PM

- types of parsers.
  - Bottom-up
    - shift-reduce
      - Bison.
  - Top Down
    - Recursive Descent Parser

- Top-Down Parser:
  - Read input Left-to-Right
  - Produces a Leftmost Derivation
    - called: LL-parsers.



  - Most Common types:
    - Recursive Descent :- directly encodes grammar rules as code functions.
    - Predictive Parse tables :- Table-Driven Algorithm.
  - Limitations
    - Not al Context-Free Grammars can be Parsed.
    - Limited to a subclass LL-Grammars.

- Recursive Descent Parser:-

Assume :
- global variable token.
  stores the current symbol
- function gettoken()
  reads the next terminal symbol from input and updates token.

Encoding :- One code function per non-terminal symbol.

Template :-

$A \rightarrow \alpha \beta \Gamma$

$A \rightarrow a\ b\ c$

parse_A()

for each symbol x in the body of the rule
- if x is a terminal symbol
  compare x to token
  if the same, consume token
- if X is a non-terminal symbol
  call function parse_X()

if a non-terminal has more than one body the token should determine which body to apply.

E.G #0

1.- $S \rightarrow CC$

2.- $C \rightarrow aC$

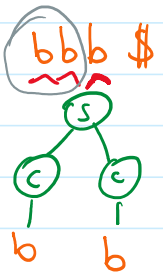
3.- $C \rightarrow b$

```
FUNCTION Parse_S ( )
  Parse_C()
  Parse_C()
END.
```

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

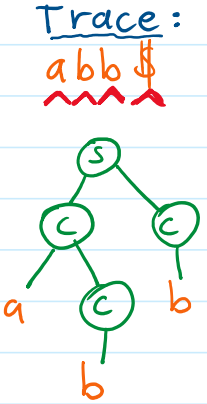
Starting the Parser:

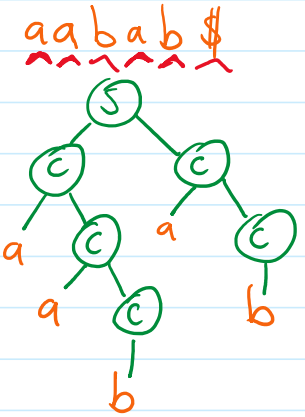
```
FUNCTION main ( )
  getToken()
  Parse_S()
  IF token != '$' THEN
    error()
  END
END.
```



error()!

Trace:





E.G #1

- $S \rightarrow dAc \mid b$
  $A \rightarrow baB \mid \lambda$
  $B \rightarrow aS$

```
FUNCTION Parse_S ( )
  IF token = 'd' THEN
    getToken()
    parse_A()
    IF token = 'c' THEN
      getToken()
    ELSE
      error()
    END
  ELSIF token = 'b' THEN
    getToken()
  ELSE
    error()
  END
END.
```

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

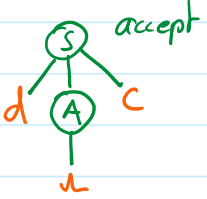
```
FUNCTION Parse_B ( )
  IF token = 'a' THEN
    getToken()
    parse_S()
  ELSE
    error()
  END
END.
```

Trace:

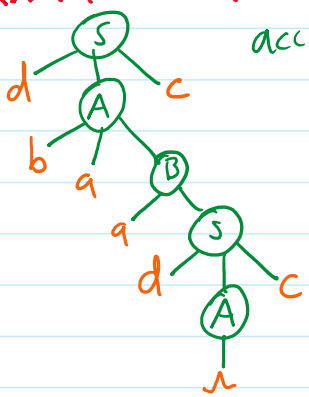
ca $
^
reject.

dba $
^^^^
reject.

dc $
^^^
accept



dbaadcc $
^^^^^^^^
accept



- Extended BNF

  We extend our grammar format with 2 more shorthands

  [ ] option
  { } repetition

  $A \rightarrow a[b]c \equiv A \rightarrow ac \mid abc$

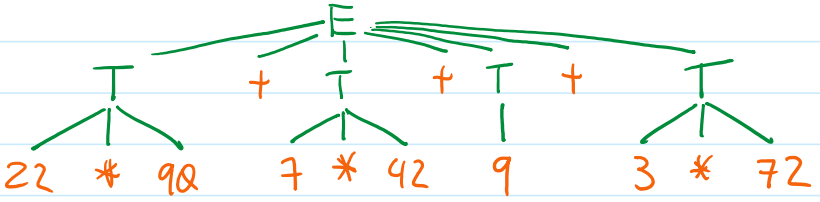
  $B \rightarrow a\{b\}c \equiv B \rightarrow ac \mid abc \mid abbc \mid abbbc \mid abbbbc \ldots$

  zero or more repetitions of b

E.g

$E \rightarrow T\{+T\}$
$T \rightarrow \underline{int}\{*\ \underline{int}\}$

22 * 90 + 7 * 42 + 9 + 3 * 72



Encoding.

Encoding.

```
FUNCTION Parse_E ( )
  Parse_T()
  WHILE token = '+' DO
    getToken()
    Parse_T()
  END
END.
```

```
FUNCTION Parse_T ( )
  IsInt()
  WHILE token = '*' DO
    getToken()
    IsInt()
  END
END.
```

E.G.

S → if C then B [else B] end

sentinel

```
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("end expected")
      END
    ELSE
      error("then expected")
    END
  END
END.
```

option

E.G.

if .. elsif ... else.

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

—o— EOF