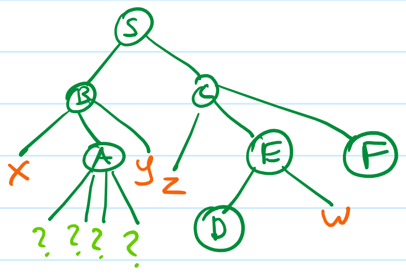
# 10 Recursive Descent Parsing

Monday, March 9, 2026 3:02 PM

- Types of parsers
  - Bottom-up
    ↳ shift-reduce
      ↳ Bison
  - Top-Down
    ↳ "Recursive Descent Parser"

- Top-Down Parser: Recursive Descent.
  - Read input from Left-to-Right.
  - Produce a Leftmost-derivation
    LL-parsers:

Intuition.



- Most common implementations:
  - Recursive Descent.- Directly encode the grammar as code functions.
  - Predictive Parse tables. Table Driven Algorithm

- Limitations
  - Not all CFG can be parsed.
  - Limited to a subclass: LL-Grammars.

- RECURSIVE DESCENT PARSER

Encoding.- One function per non-terminal symbol
One function implements a set grammar rules.

Assume.- • global variable token that stores the current symbol from the input.

Assume :- • global variable token that stores the current symbol from the input.
• function get_token() :- read a new symbol from the input and update token.

Template :-

$A \to \alpha\ \beta\ \Gamma$
$A \to x\ y\ z$

parse_A()
for each symbol x in the body of a rule
  - if x is a terminal symbol :
      compare x to token
      if it matches, consumme token : get_token()
  - if x is a non-terminal symbol :
      call function parse_X()
If a non-terminal has more than one rule
(i.e. more than one body, the token should determine which body to apply )

E.G. #0

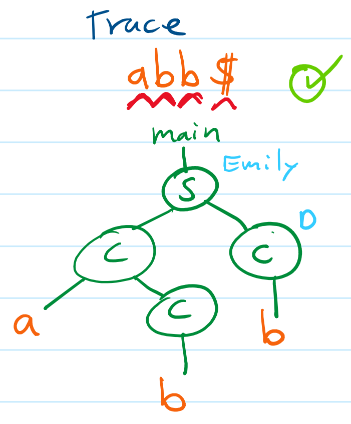
1 $S \to CC$
2 $C \to aC$
3 $C \to b$

```
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.
```

Start the Parser:

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



EG #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
```

```
FUNCTION Parse_A()
  IF token = 'b' THEN
    GetToken()
    IF token = 'a' THEN
      GetToken()
```

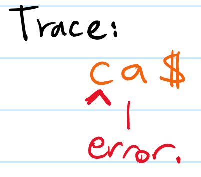
$B \to aS$

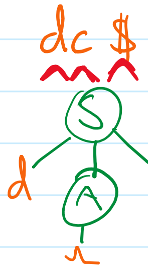
```
FUNCTION Parse_B()
  IF token = 'a' THEN
    GetToken()
    Parse_S()
  ELSE
    error()
  END
END.
```

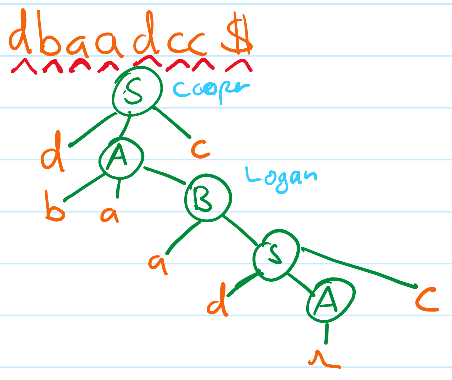
```
    GetToken()
    Parse_A()
    IF token = 'c' THEN
      GetToken()
    ELSE
      error()
    END
  ELSIF token = 'b' THEN
    GetToken()
  ELSE
    error()
  END
END.
```

```
    GetToken()
    IF token = 'a' THEN
      GetToken()
      Parse_B()
    ELSE
      error()
    END
  END
END.
```

Trace:







- EXTENDED BNF

We extend our grammar rule format with 2 shorthands.

[ ] option

{ } repetition.

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

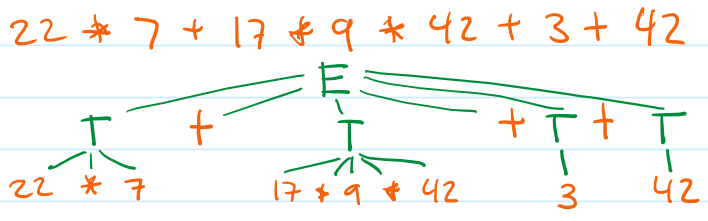
$B \to a\{b\}c \equiv B \to ac \mid abc \mid abbc \mid abbbc \mid abbbbc \mid \ldots\ldots$

zero or more repetitions of b

E.g.

$E \to T\{+T\}$ sentinel

$T \to \underline{int}\ \{* \underline{int}\}$ sentinel.



-Encoding.

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

```
FUNCTION Parse_T()
  IsTokenInteger()
  WHILE token = '*' DO
    GetToken()
    IsTokenInteger()
  END
END.
```

E.G.

$I \rightarrow$ if C then B [else B] end

```
FUNCTION Parse_I()
  IF token = 'if' THEN
    GetToken()
    Parse_C()
    IF token = 'then' THEN
      GetToken()
      Parse_B()

      IF token = 'else' THEN     } option.
        GetToken()
        Parse_B()
      END

      IF token = 'end' THEN
        GetToken()
      ELSE
        error(' end expected ')
      END.
    ELSE
      error( ' then expected ' )
  ELSE
    error(' if expected ' )
  END
END.
```

IF C THEN

if ( C )

if C :

else:

E.g

if .... elif ..... else ....

(Python)

$I \rightarrow$ if C : B {elif C : B} [else: B]