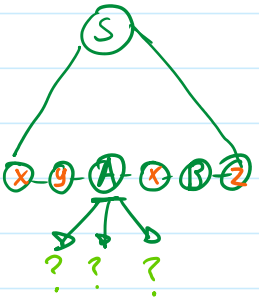
10 Recursive Descent Parsing

Monday, October 7, 2024 12:01 PM

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

- 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 context-free grammars can be parses
    - Limited to a subclass: LL-Grammars.

- Recursive Descent Parsing:
  assume: • global variable token
    stores current symbol

assume :- • global variable token
stores current symbol
• function getToken() that reads the next terminal symbol from the input.

encoding :-
one function per non terminal symbol

e.g
$A \to \alpha \beta \Gamma$
$A \to a\ b\ c$

parse_A()

- for each symbol X in the body
  - if X is a terminal symbol
    compare 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 rule to apply

E.G. #8

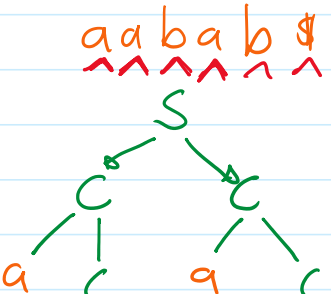
$S \to CC$
$C \to aC$
$C \to b$

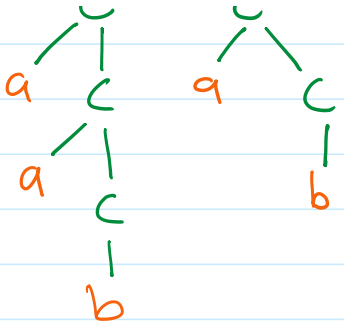
```
FUNCTION parse_S()
  parse_C()
  parse_C()
END.
```

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

Starting the parser:

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





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.
```

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

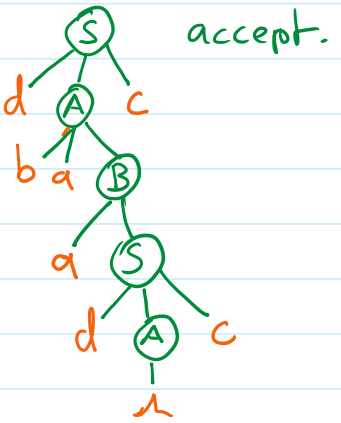
Trace:

ca$
^
reject.

dba$
^^^^
reject.

dc$
^^^
accept.

dbaadcc$
^^^^^^^^
accept.



- Extended BNF

  We extend our grammar format with 2 new shorthands

  [ ] option
  { } repetition

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

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

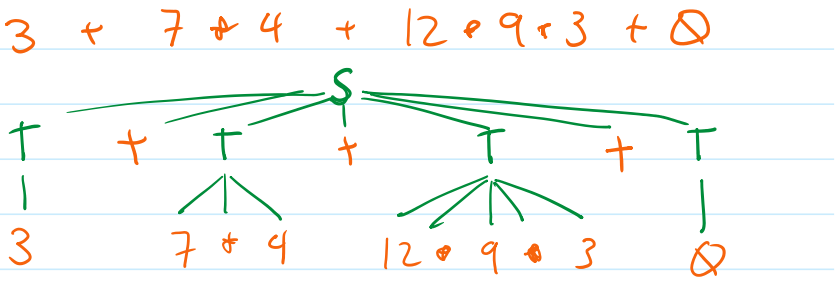
$B \rightarrow a\{b\}c \equiv B \rightarrow ac \mid abc \mid abbc \mid abbbc \mid abbbbc \mid \ldots$

zero or more repetitions of b

E.g.

$S \rightarrow T\{+T\}$

$T \rightarrow int\{*int\}$



Encoding:

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

sentinel.

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

EG.:

$S \rightarrow$ 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()

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

Parses option.

```
        ELSE
            error("If expected")
        END
    END.
```

E.G.

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