parsing !

© 2020 **Kris Jordan** All Rights Reserved

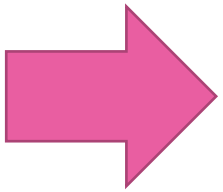
The CharPairs Language Grammar

```
Node  -> Char | Pair
Char  -> Any character except '(' ')' or ' '
Pair  -> '(' Node ' ' Node ')'
```

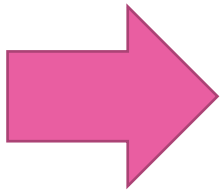
Example string: `(a (b c))`

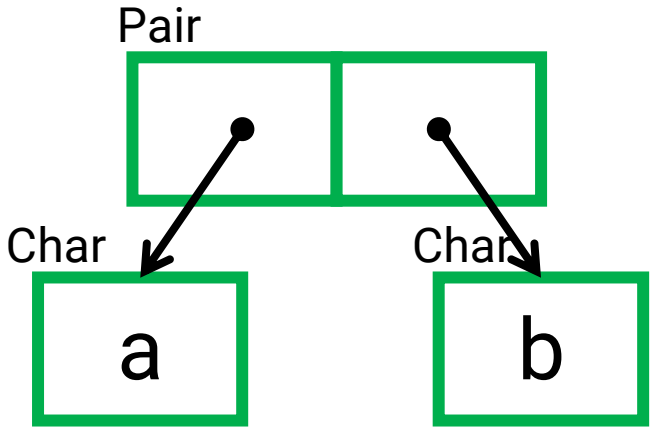
The CharPairs Parsing Pipeline

"(a b)"



LParen
Char('a')
Space
Char('b')
RParen





Scanning

Input lexemes are transformed into meaningful **tokens.**

Parsing

Parse Tree data structure is built-up to represent the relations of tokens.

```
Pair(
    Char('a'),
    Char('b')
)
```

Parsing: Given tokens and a grammar, generate a parse tree.

Example Input String: `(a (b c))`

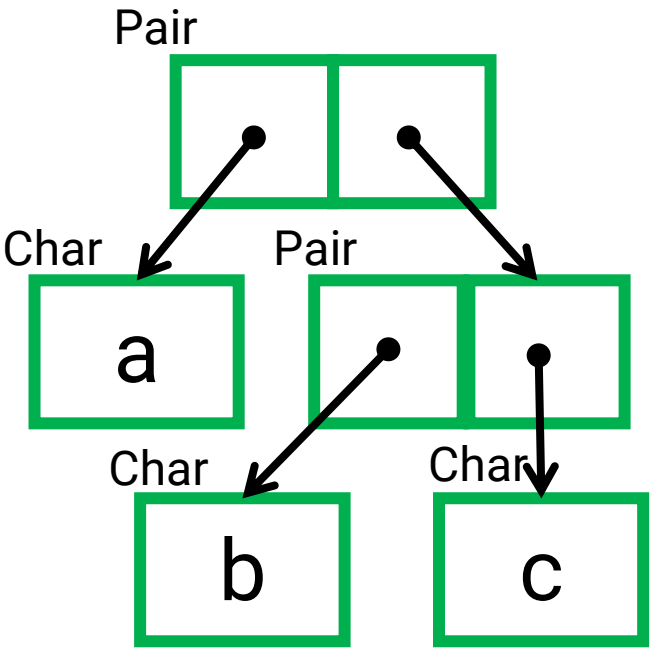
Tokens

```
LParen
Char('a')
Space
LParen
Char('b')
Space
Char('c')
RParen
RParen
```

Grammar

```
Node  -> Char | Pair
Char  -> Not ( ) or ' '
Pair  -> '('
         Node
         ' '
         Node
         ')'
```

Parse Tree



These values are variants of the example's Value enum.

A Grammar's Suitability for Parsing

- Not all grammars are straightforward to parse due to:
  - Ambiguities inherent in the grammar itself where ambiguity means given an input string and a grammar there are many valid parse trees.
  - The amount of peeking "lookahead" required to know what production rule to choose at any given point in the process.
  - The organization of recursive production rules.

- This subject gets full and proper treatment in Compilers

- The grammars you will need to parse in this course are intentionally designed to:
  1. Be unambiguous
  2. Require only one token of lookahead
  3. Not have any left recursive production rules

- This class of grammars is formally referred to as LL(1) (Lewis-Stearns 1969)
  - **L**eft-to-right, **L**eftmost derivation
  - Only **1** lookahead token required

To Parse Top-Down or Bottom-up?

- Both are possible and prevalent!

- If you're implementing a parser by hand, top-down parsing is typical.
  - Given an LL(1) grammar, there's a 1-to-1 translation from rules to code
  - You will ***feel*** the ***beautiful*** connection between theory and pragmatics

- Many real languages use *parser generators* to emit their parser's code.
  - A parser generator is given a grammar and generates the code for a parser.
  - Generated parsers are typically bottom-up parsers.
  - Generated parsers are more complex and handle broader a class of grammars.

Union Type Walk-through

- Please watch the video 20.2 which gives an introduction to union types in C.

- It also describes how we'll represent a parsed Node in today's sample code.

- https://www.youtube.com/watch?v=BrJoKr1mtbw&list=PLKUb7MEve0TgkIM6eK6EUl9n-ClmVATND&index=33

Recursive Descent Parsing on LL(1) Grammars

- Input: A Stream of Peekable Tokens
- Output: A Parse Tree
- General Implementation Strategy:

1. Write a function for each non-terminal production rule
   - Each function returns a parse tree node to represent its production rule (i.e. parse_char returns a Char Node, parse_pair returns a Pair Node)

2. Each non-terminal function's body translates its grammar definition:
   - Alternation (OR) | - peek ahead to know what step to take next
   - Terminal - take that token and move forward.
   - Non-terminal - call the non-terminal function responsible for parsing it.

3. Parse an input string by calling the starting symbol's production rule

Pseudo-code for Recursive Descent Parsing (1/5)

CharPair Grammar:

```
Value -> Char | Pair
Char  -> Characters except '(' ')' or ' '
Pair  -> '(' Value ' ' Value ')'
```

1. **Write a function for each non-terminal production rule**
2. Each non-terminal function's body translates its grammar definition.
3. Parse an input string by calling the initial non-terminal production rule

```
func parse      -> Node



func parse_char -> Node (Char)



func parse_pair -> Node (Pair)
```

Pseudo-code for Recursive Descent Parsing (2/5)

CharPair Grammar:

```
Node -> Char | Pair
Char  -> Characters except '(' ')' or ' '
Pair  -> '(' Node ' ' Node ')'
```

1. Write a function for each non-terminal production rule
2. **Each non-terminal function's body translates its grammar definition.**
3. Parse an input string by calling the initial non-terminal

Notice because of Value's alternation of either Char or Pair we need to peek ahead. We look at the first tokens of the rules we're alternating between to decide what to do next.

```
func parse       -> Node
  if peek == '('
    return parse_pair()
  else
    return parse_char()

func parse_char -> Node (Char)

func parse_pair -> Node (Pair)
```

Pseudo-code for Recursive Descent Parsing (3/5)

CharPair Grammar:

```
Node -> Char | Pair
Char  -> Characters except '(' ')' or ' '
Pair  -> '(' Node ' ' Node ')'
```

1. Write a function for each non-terminal production rule
2. **Each non-terminal function's body translates its grammar definition.**
3. Parse an input string by calling the initial non-terminal

Parsing a Char is straightforward, we're simply converting a Char token into a Char value.

```
func parse       -> Node
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

func parse_char -> Node (Char)
  ret Node::Char(take_char())

func parse_pair -> Node (Pair)
```

Pseudo-code for Recursive Descent Parsing (4/5)

CharPair Grammar:

```
Node  -> Char | Pair
Char  -> Characters except '(' ')' or ' '
Pair  -> '(' Node ' ' Node ')'
```

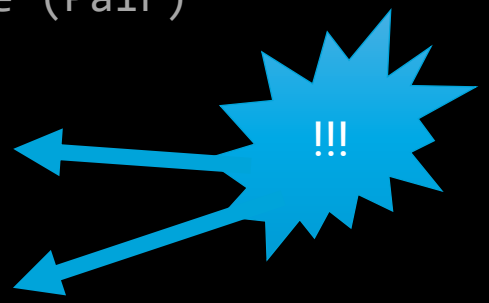
1. Write a function for each non-terminal production rule
2. **Each non-terminal function's body translates its grammar definition.**
3. Parse an input string by calling the initial non-terminal

!!! This is where you realize recursive descent !!!
Notice there's *mutual recursion* here. The function parse calls parse_pair and parse_pair calls parse. The base cases here are found in parse_char (valid) or parse_pair (in an error state).

```
func parse      -> Node
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

func parse_char -> Node (Char)
  ret Node::Char(take_char())

func parse_pair -> Node (Pair)
  take_lparen()
  left = parse()
  take_space()
  right = parse()
  take_rparen()
  ret Node::Pair(left, right)
```



Pseudo-code for Recursive Descent Parsing (5/5)

CharPair Grammar:

```
Node  -> Char | Pair
Char  -> Characters except '(' ')' or ' '
Pair  -> '(' Node ' ' Node ')'
```

1. Write a function for each non-terminal production rule
2. Each non-terminal function's body translates its grammar definition.
3. **Parse an input string by calling the initial non-terminal**

Finally, to parse an input string you would establish the connection between your scanner and parser and then call **`parse`** which is the start rule.

```
func parse      -> Node
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

func parse_char -> Node (Char)
  ret Node::Char(take_char())

func parse_pair -> Node (Pair)
  take_lparen()
  left = parse()
  take_space()
  right = parse()
  take_rparen()
  ret Node::Pair(left, right)
```

Parser Hands-on

- Please watch the video 20. 4 which walks through implementing a simple recursive descent Parser.

- https://www.youtube.com/watch?v=sUxFE32tXF0&list=PLKUb7MEve0TgkIM6eK6EUl9n-ClmVATND&index=35&t=0s

Traversing a Parse Tree Hands-on

- Please watch the video 20.5 which walks through processing the tree of Node values produced by the Parser.

- https://www.youtube.com/watch?v=rMp_zOdvDUw&list=PLKUb7MEve0TgkIM6eK6EUl9n-ClmVATND&index=36&t=0s