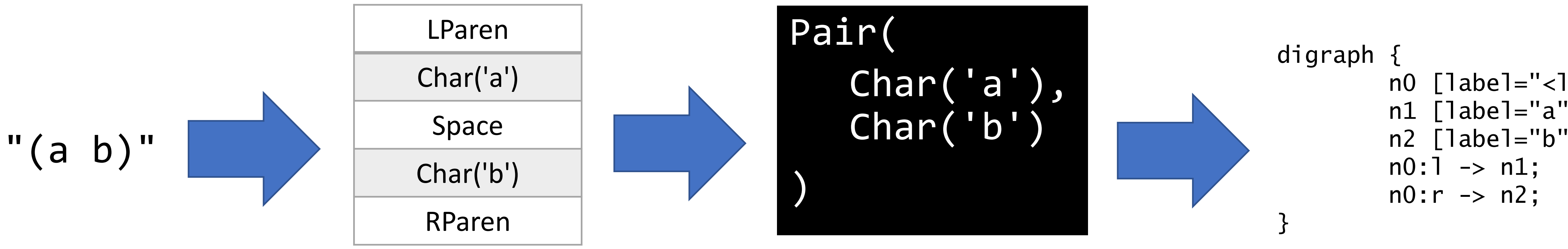
Recursive Descent Parsing 101

The CharPairs Language Grammar

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

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

The CharPairs to DOT Compilation Pipeline



```
"(a b)"
```

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

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

```
digraph {
        n0 [label="<l
        n1 [label="a"
        n2 [label="b"
        n0:l -> n1;
        n0:r -> n2;
}
```

Tokenization

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

Parsing

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

Code Generation

Finally, an algorithm visits the hierarchy to generate a **target** representation.

We can then take our

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

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

Parse Tree

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

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

A Grammar's Suitability for Parsing

- Not all grammars are equally 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 arrangement of recursive production rules.

- We *may* get more into these details on grammar properties later on
    - However, this subject gets full and proper treatment in COMP520 - 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

- Most 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 classes of grammars.

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_value returns a Value, parse_char returns a Value::Char, and so on)

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 initial non-terminal 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_value -> Value




func parse_char -> Value::Char




func parse_pair -> Value::Pair
```

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

```
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

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_value -> Value
  if peek == '('
    return parse_pair()
  else
    return parse_char()


func parse_char -> Value::Char


func parse_pair -> Value::Pair
```

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

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

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

```
func parse_value -> Value
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

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

func parse_pair -> Value::Pair
```

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

CharPair Grammar:

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

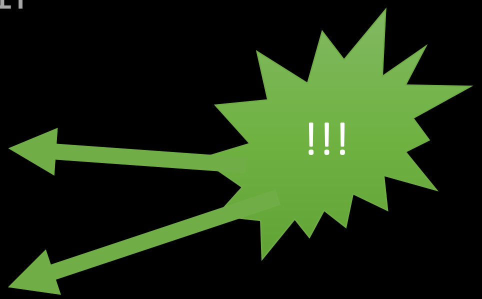
1. Write a function for each non-terminal production rule
2. **Each non-terminal function's body translates its grammar definition.**
3. Parse an input string by calling the initial non-terminal production rule

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

```
func parse_value -> Value
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

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

func parse_pair -> Value::Pair
  take_lparen()
  lhs = parse_value()
  take_space()
  rhs = parse_value()
  take_rparen()
  ret Value::Pair(lhs, rhs)
```



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

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**

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

```
func parse_value -> Value
  if peek == '('
    ret parse_pair()
  else
    ret parse_char()

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

func parse_pair -> Value::Pair
  take_lparen()
  lhs = parse_value()
  take_space()
  rhs = parse_value()
  take_rparen()
  ret Value::Pair(lhs, rhs)
```

Let's Implement the Core of a CharPair Parser

- Today's example has the skeleton of a CharPair to DOT compiler:
  - Tokenization is *naively* handled in src/tokenization.rs
  - Parsing (ready for our implementation) is in src/parser.ts
  - CodeGen is in src/dot_gen.rs (what we implemented last lecture)

- The main function follows our high-level process:

```
let source = read_line();
// 1. Tokenization / Lexing
let tokens = Tokenizer::new(&source);
// 2. Parsing
let parse_tree = Parser::new(tokens).parse_value();
// 3. Code Generation
let target = DotGen::new(&parse_tree).to_string();
println!("{}", target);
```

The **Parser**'s Helper Method: **take**

- You'll need to *take* specific tokens during parsing
  - For example, when parsing a Pair, after the first value there must be a Space token.

- The helper method below will *take* the next token and ensure it's what we expected.

- Our CharPair parser will not handled malformed source strings gracefully. It'll *panic!*

```rust
fn take(&mut self, expected: Token) {
    if let Some(next) = self.tokens.next() {
        if next != expected {
            panic!("Expected: {:?} - Found {:?}", expected, next);
        }
    } else {
        panic!("Expected: {:?} - Found None", expected);
    }
}
```

Follow-along: Implementing the Parser

1. Write a function for each non-terminal production rule
   - parse_value is already established in the skeleton code
   - We'll need to add parse_char and parse_pair

2. Each non-terminal function's body translates its production rule
   - The production rules for Char and Pair are in the comments

3. Parse an input string by calling the initial non-terminal production rule
   - This is already handled in the main function

```rust
// Grammar: Value -> Char | Pair
pub fn parse_value(&mut self) -> Value {
    if let Some(token) = self.tokens.peek() {
        match *token {
            Token::Char(_) => return self.parse_char(),
            Token::LParen => return self.parse_pair(),
            _ => { /* Fall through to panic */ }
        }
    }
    panic!("parse_value: Expected Char | Pair");
}

// TODO #1) Define methods for non-terminals of grammar.

// Grammar: Char -> Any Token::Char
fn parse_char(&mut self) -> Value {
    if let Token::Char(c) = self.tokens.next().unwrap() {
        Value::Char(c)
    } else {
        panic!("This should never happen...");
    }
}

// Grammar: Pair -> LParen Value Space Value RParen
fn parse_pair(&mut self) -> Value {
    self.take(Token::LParen);
    let lhs = self.parse_value();
    self.take(Token::Space);
    let rhs = self.parse_value();
    self.take(Token::RParen);
    Value::Pair(Box::new(lhs), Box::new(rhs))
}
```