For all terminals, use function `match_tok` a
- If lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
- Fails with a parse error if lookahead is not a

For each nonterminal N, create a function `parse_N`
- Called when we’re trying to parse a part of the input which corresponds to (or can be derived from) N
- `parse_S` for the start symbol S begins the parse
Example Parser

- Given grammar $S \rightarrow xyz \mid abc$
- Parser
  
  ```ml
  let parse_S () =
  if lookahead () = "x" then (* $S \rightarrow xyz$ *)
      (match_tok "x";
       match_tok "y";
       match_tok "z")
  else if lookahead () = "a" then (* $S \rightarrow abc$ *)
      (match_tok "a";
       match_tok "b";
       match_tok "c")
  else raise (ParseError "parse_S")
  ```
Another Example Parser

- **Given grammar** \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)

let rec parse_S () =
  if lookahead () = "x" ||
  lookahead () = "y" then
  parse_A () (* S \rightarrow A *)
  else if lookahead () = "z" then
  parse_B () (* S \rightarrow B *)
  else raise (ParseError "parse_S")

and parse_A () =
  if lookahead () = "x" then
  match_tok "x" (* A \rightarrow x *)
  else if lookahead () = "y" then
  match_tok "y" (* A \rightarrow y *)
  else raise (ParseError "parse_A")

and parse_B () = ...
Execution Trace = Parse Tree

- If you draw the execution trace of the parser
  - You get the parse tree

- Examples
  - Grammar
    - S → xyz
    - S → abc
  - String “xyz”
    - parse_S ()
      - match_tok “x”
      - match_tok “y”
      - match_tok “z”

  - Grammar
    - S → A | B
    - A → x | y
    - B → z
  - String “x”
    - parse_S ()
      - parse_A ()
      - match_tok “x”
Consider grammar $S \rightarrow Sa \mid \varepsilon$

- Try writing parser

```ocaml
let rec parse_S () =
  if lookahead () = "a" then
    (parse_S ();
    match_tok "a") (* S \rightarrow Sa *)
  else ()
```

- Body of `parse_S ()` has an infinite loop!
  - Infinite loop occurs in grammar with left recursion
Consider grammar \( S \rightarrow aS \mid \epsilon \)  
Again, \( \text{First}(aS) = a \)

- Try writing parser

\[
\text{let rec parse}_S() = \\
\quad \text{if } \text{lookahead}() = "a" \text{ then} \\
\quad \quad (\text{match_token } "a"; \\
\quad \quad \quad \text{parse}_S()) (* S \rightarrow aS *) \\
\quad \text{else } ()
\]

- Will \( \text{parse}_S() \) infinite loop?
  - Invoking \text{match_token} will advance lookahead, eventually stop
- Top-down parsers handles grammar w/ right recursion
Algorithm To Eliminate Left Recursion

- **Given grammar**
  - \( A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta \)
    - \( \beta \) must exist or no derivation will yield a string

- **Rewrite grammar as (repeat as needed)**
  - \( A \rightarrow \beta L \)
  - \( L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \epsilon \)

- **Replaces left recursion with right recursion**

- **Examples**
  - \( S \rightarrow Sa \mid \epsilon \quad \Rightarrow \quad S \rightarrow L \quad L \rightarrow aL \mid \epsilon \)
  - \( S \rightarrow Sa \mid Sb \mid c \quad \Rightarrow \quad S \rightarrow cL \quad L \rightarrow aL \mid bL \mid \epsilon \)
Quiz 4

What does the following code parse?

```ocaml
let parse_S () =
    if lookahead () = "a" then
        (match_tok "a";
         match_tok "x";
         match_tok "y";
         match_tok "q")
    else
        raise (ParseError "parse_S")
```

A. $S \rightarrow axyq$
B. $S \rightarrow a \mid q$
C. $S \rightarrow aaxy \mid qq$
D. $S \rightarrow axy \mid q$
Quiz 4

What does the following code parse?

```ocaml
let parse_S () =
  if lookahead () = "a" then
    (match_tok "a";
     match_tok "x";
     match_tok "y";
     match_tok "q")
  else
    raise (ParseError "parse_S")
```

A. $S \rightarrow axyq$
B. $S \rightarrow a \mid q$
C. $S \rightarrow aaxy \mid qq$
D. $S \rightarrow axy \mid q$
Quiz 5

What Does the following code parse?

```
let rec parse_S () =
    if lookahead () = "a" then
        (match_tok "a";
         parse_S ())
    else if lookahead () = "q" then
        (match_tok "q";
         match_tok "p")
    else
        raise (ParseError "parse_S")
```

A. $S \rightarrow aS \mid qp$
B. $S \rightarrow a \mid S \mid qp$
C. $S \rightarrow aqSp$
D. $S \rightarrow a \mid q$
What Does the following code parse?

```ocaml
let rec parse_S () =
    if lookahead () = "a" then
        (match_tok "a";
         parse_S ())
    else if lookahead () = "q" then
        (match_tok "q";
         match_tok "p")
    else
        raise (ParseError "parse_S")
```

A. $S \rightarrow aS \mid qp$
B. $S \rightarrow a \mid S \mid qp$
C. $S \rightarrow aqSp$
D. $S \rightarrow a \mid q$
Can recursive descent parse this grammar?

S → aBa
B → bC
C → ε | Cc

A. Yes
B. No
Quiz 6

Can recursive descent parse this grammar?

\[
\begin{align*}
S & \rightarrow aBa \\
B & \rightarrow bC \\
C & \rightarrow \varepsilon \mid Cc
\end{align*}
\]

A. Yes
B. No
(due to left recursion)
Recall: The Compilation Process

Where does this come from?
Parse Trees to ASTs

- Parse trees are a representation of a parse, with all of the syntactic elements present
  - Parentheses
  - Extra nonterminals for precedence

- This extra stuff is needed for parsing

- Lots of that stuff is not needed to actually implement a compiler or interpreter
  - So in the abstract syntax tree we get rid of it
Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts.