Recursive Descent Parser Implementation

- 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 → xyz | abc
- Parser

```
let parse_S () =
  if lookahead () = "x" then (* S → xyz *)
    (match_tok "x";
     match_tok "y";
     match_tok "z")
   else if lookahead () = "a" then (* S → abc *)
     (match_tok "a";
     match_tok "b";
     match_tok "c")
   else raise (ParseError "parse_S")
```

Another Example Parser

- Given grammar S → A | B   A → x | y   B → z

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

and parse_A () =
   if lookahead () = "x" then
     match_tok "x" (* A → x *)
   else if lookahead () = "y" then
     match_tok "y" (* A → 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”

```
  S
 /|\
x y z
```

  - Grammar

    S → A | B

    A → x | y

    B → z
  - String “x”

    parse_S ()

      parse_A ()

        match_tok “x”

```
S
|
A
|
x
```

Left Recursion

- Consider grammar $S \rightarrow Sa \mid \varepsilon$
  - Try writing parser

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

  - Body of parse_S () has an infinite loop!
    - Infinite loop occurs in grammar with left recursion

Right Recursion

- Consider grammar S → aS | ε　　　Again, First(aS) = a
  - Try writing parser

```
let rec parse_S () =
    if lookahead () = "a" then
      (match_tok "a";
      parse_S ())  (* S → aS *)
    else ()
```

  - Will parse_S( ) infinite loop?
    - Invoking match_tok will advance lookahead, eventually stop
  - Top-down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
    - β must exist or no derivation will yield a string
- Rewrite grammar as (repeat as needed)
  - $A \rightarrow \beta L$
  - $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon$
- Replaces left recursion with right recursion
- Examples
  - $S \rightarrow Sa \mid \varepsilon$ ⇨ $S \rightarrow L$ $\quad L \rightarrow aL \mid \varepsilon$
  - $S \rightarrow Sa \mid Sb \mid c$ ⇨ $S \rightarrow cL$ $\quad L \rightarrow aL \mid bL \mid \varepsilon$

Quiz 4

- What does the following code parse?

```
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 → axyq
B. S → a | q
C. S → aaxy | qq
D. S → axy | q

Quiz 4

- What does the following code parse?

```
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. $S \rightarrow a \mid q$

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 → aS | qp

B. S → a | S | qp

C. S → aqSp

D. S → a | q

Quiz 6

Can recursive descent parse this grammar?

$S \rightarrow aBa$
$B \rightarrow bC$
$C \rightarrow \varepsilon \mid Cc$

A. Yes
B. No

Quiz 6

Can recursive descent parse this grammar?

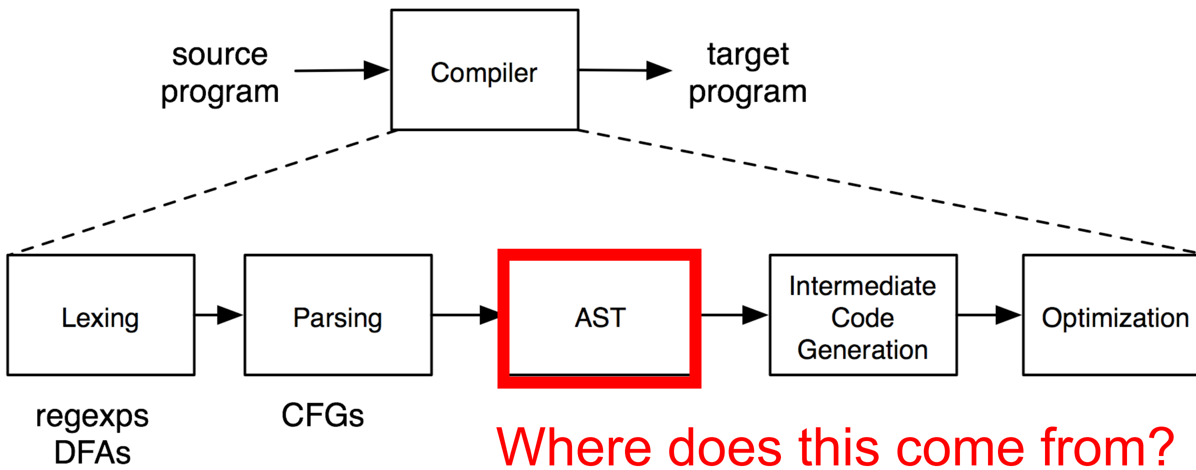
$$S \rightarrow aBa$$
$$B \rightarrow bC$$
$$C \rightarrow \varepsilon \mid Cc$$

A. Yes

B. No

(due to left recursion)

Recall: The Compilation Process

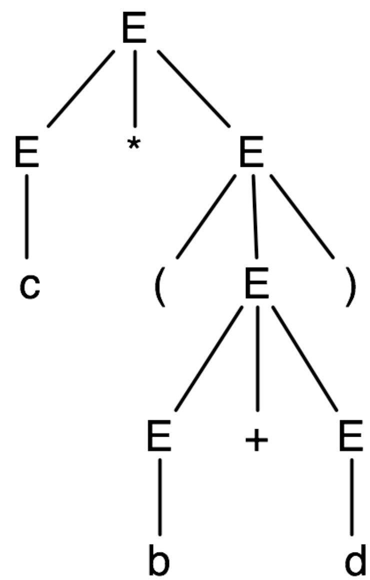


Parse Trees to ASTs

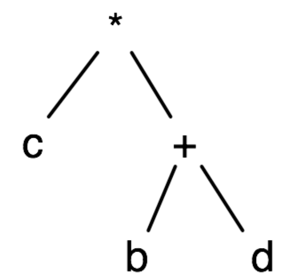
- Parse trees are a representation of a parse, with all of the syntactic elements present
  - Parentheses
  - Extra nonterminals for precedence
- This extra stuff is needed for parsing
- Lots of that stuff is not needed to actually implement a compiler or interpreter
  - So in the abstract syntax tree 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



parse
tree



AST