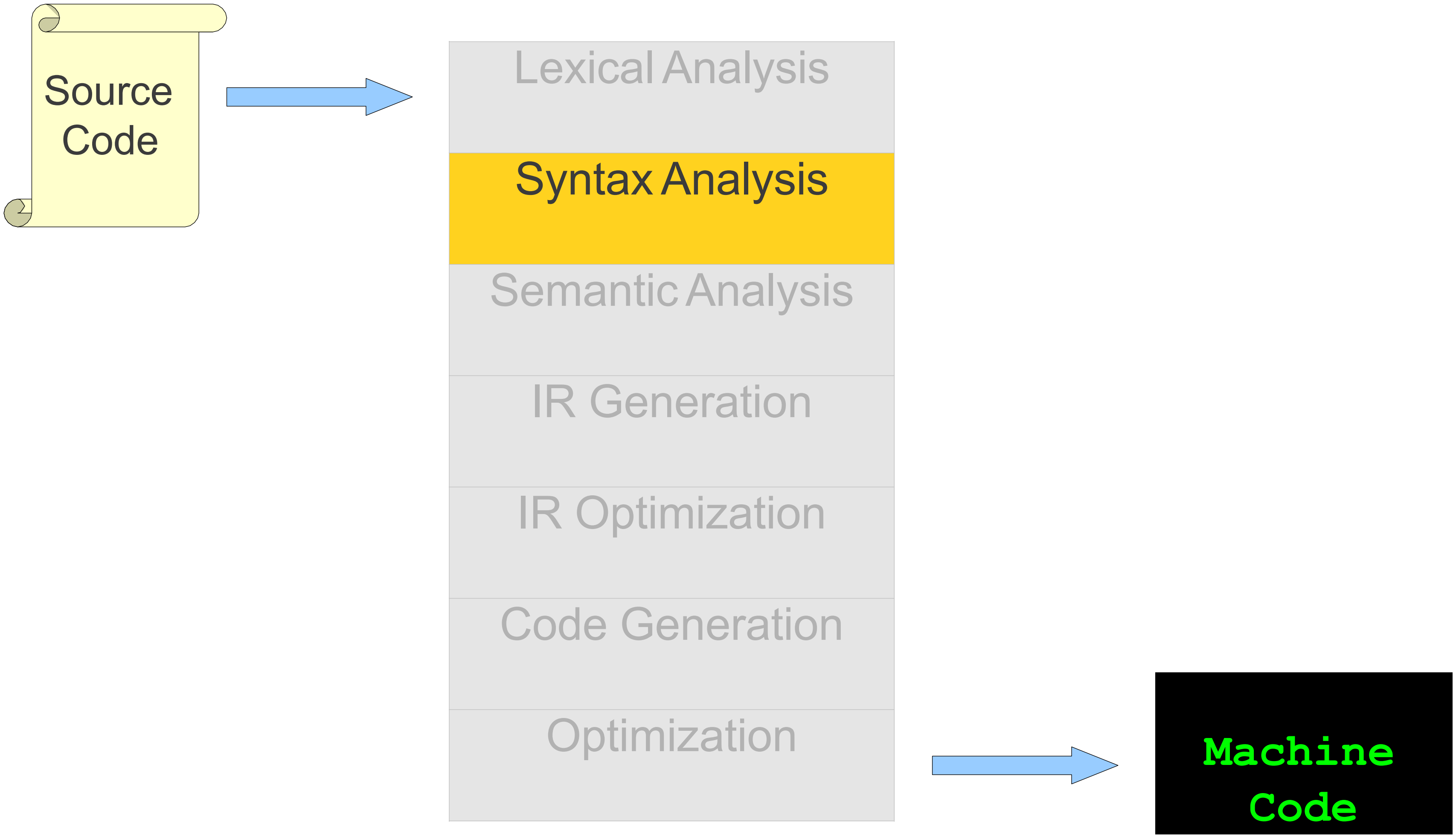
بسم الله الرحمن الرحيم

# Parsing:
## Top-Down Parsing,
## Recursive Descent & Predictive Parser & LL(1)

# Next Time


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

# Review from Last Time

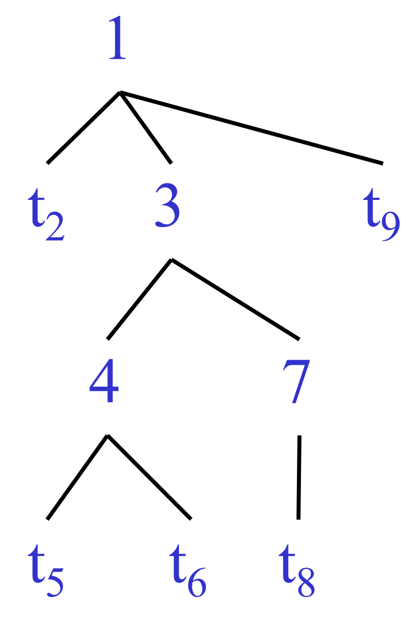
- Goal of syntax analysis: recover the intended structure of the program.
- Idea: Use a **context-free grammar** to describe the programming language.
- Given a sequence of tokens, look for a **parse tree** that generates those tokens.
- Recovering this syntax tree is called **parsing** and is the topic of this week (and part of next!)

# Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top
  - From left to right

- Terminals are seen in order of appearance in the token stream:

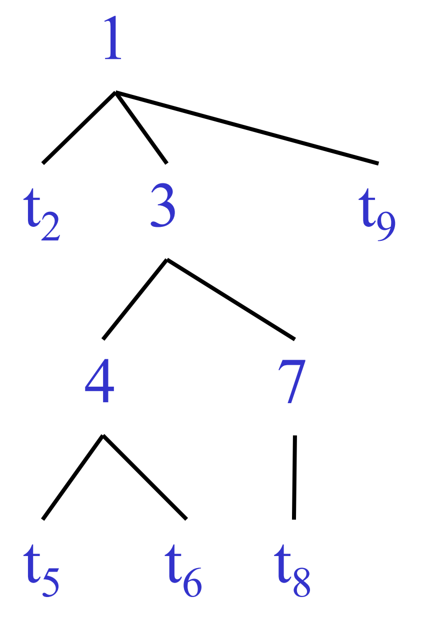
$$t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9$$



# Intro to Top-Down Parsing: The Idea

- We have seen:
  - BFS: Memory and time problems
  - DFS: Time problems
- Terminals are seen in order of appearance in the token stream:

$t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9$



# Recursive Descent Parsing

- Consider the grammar

$$E \rightarrow T \mid T + E$$

$$T \rightarrow int \mid int * T \mid ( E )$$

- Token stream is: ( int )

- Start with top-level non-terminal E
  - Try the rules for E in order

# Recursive Descent Parsing

$E \to T \mid T + E$

$T \to \text{int} \mid \text{int} * T \mid ( E )$

E

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid ( E )$

E

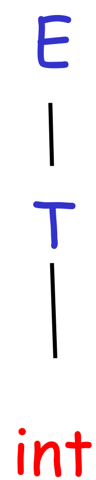
T

( int )

# Recursive Descent Parsing

E → T | T + E

T → int | int * T | ( E )




Mismatch: int is not ( !

Backtrack …

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$

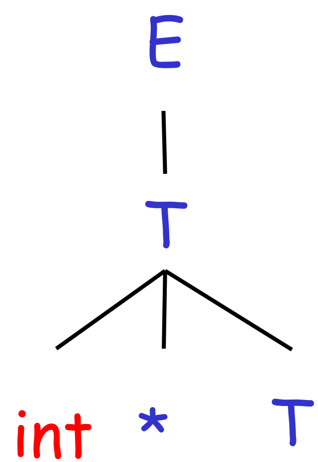
$$\begin{array}{c} E \\ | \\ T \end{array}$$

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$




Mismatch: int is not ( !

Backtrack …

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$

$$\begin{array}{c} E \\ | \\ T \end{array}$$

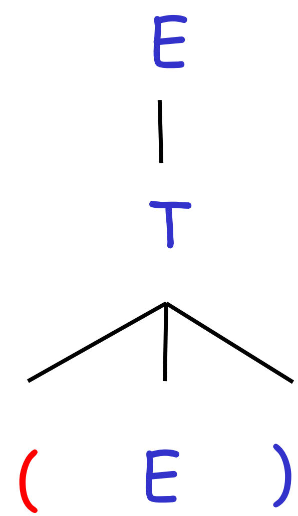
( int )

↑

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$



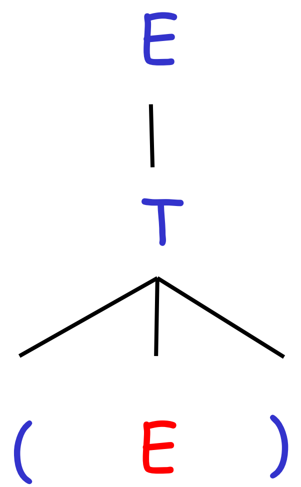

Match! Advance input.

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid ( E )$

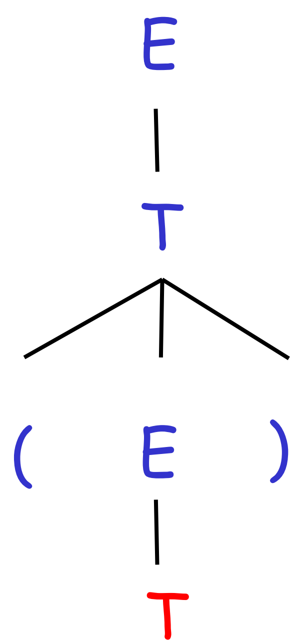



( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow$ int $\mid$ int $*$ $T \mid ( E )$

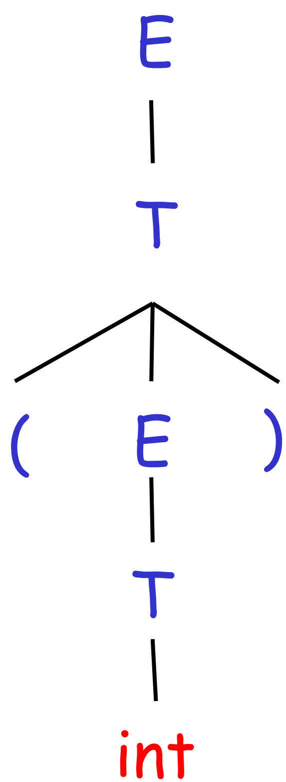



( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$



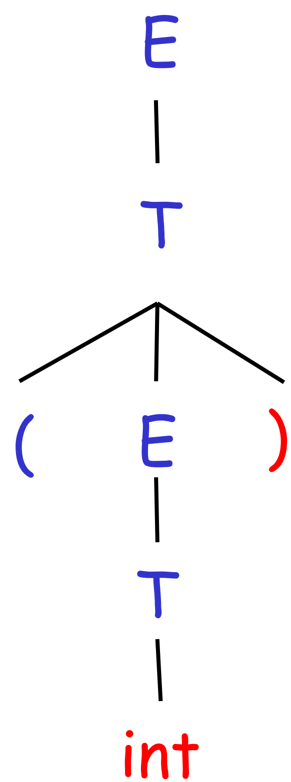

Match! Advance input.

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid ( E )$



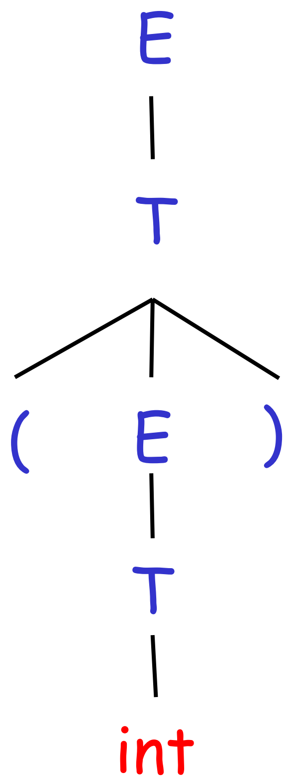

Match! Advance input.

( int )

# Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid ( E )$




End of input, accept.

( int )

# Problems with Top Down Parsing

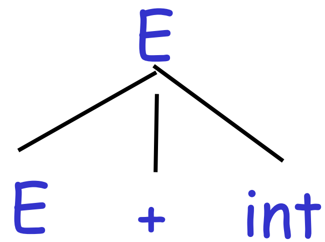
- Left Recursion in CFG may cause parser to loop forever!
- In there is a production of form $A \rightarrow A\alpha$, we say the grammar has left recursion

$$E \rightarrow E + \text{int} \mid \text{int}$$

- Solution: Remove Left Recursion...
  - without changing the Language defined by the Grammar.

# Problems with Top Down Parsing (Example)

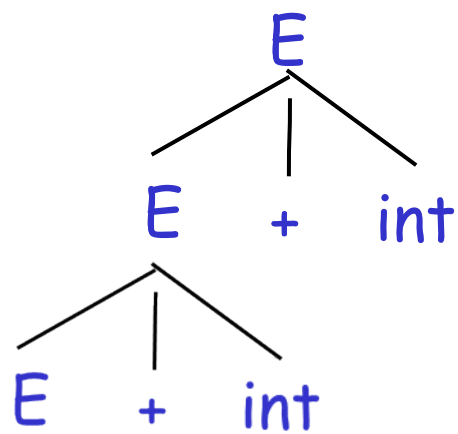
$E \rightarrow E + int \mid int$




int + int

# Problems with Top Down Parsing (Example)

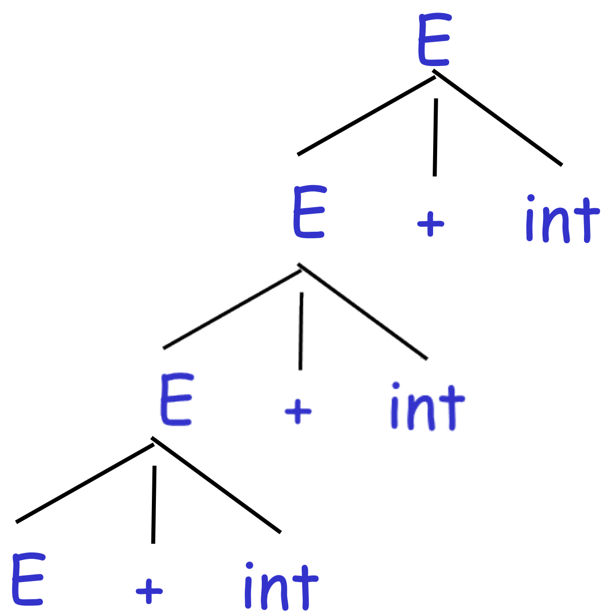
$E \rightarrow E + \text{int} \mid \text{int}$




int + int

# Problems with Top Down Parsing (Example)

$E \rightarrow E + int \mid int$




int + int

# Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\,\alpha \mid \beta$$

- $S$ generates all strings starting with a $\beta$ and followed by a number of $\alpha$

- Can rewrite using right-recursion

$$S \rightarrow \beta\, S'$$
$$S' \rightarrow \alpha\, S' \mid \varepsilon$$

# More Elimination of Left-Recursion

- In general

$$S \to S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from $S$ start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \to \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$

$$S' \to \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

# General Left Recursion

- The grammar

  $S \to A\ \alpha \mid \delta$

  $A \to S\ \beta$

  is also left-recursive because

  $S \to^{+} S\ \beta\ \alpha$

- This left-recursion can also be eliminated

- See Dragon Book for general algorithm
  - Section 4.3.3

# Left Recursion Elimination

- 1) Remove e-productions (A->e)
- 2) For i = 1 .. n:
  - 2-a) Remove Ai -> Ai \gamma
  - 2-b) Remove Ai -> Aj \gamma (j < i)
  - 2-c) Ai -> Aj -> Ai needs special attention

# Predictive Top-Down Parsing

## Parser that Never Backtracks

For Example, Consider:

Start symbol

Type → Simple
        | ↑ **id**
        | **array** [ Simple ] **of** Type

Simple → **integer**
        | **char**
        | **num dotdot num**

Suppose input is :

**array** [ **num dotdot num** ] **of integer**

Parsing would begin with

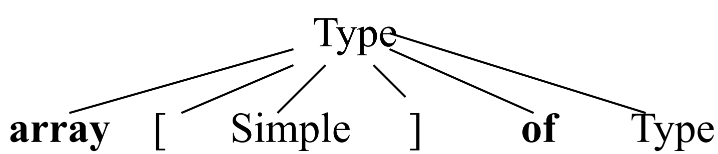
*Type* → ???

# Predictive Parsing Example

Lookahead symbol

Input :  **array** [ **num dotdot num** ] **of integer**

Type
?

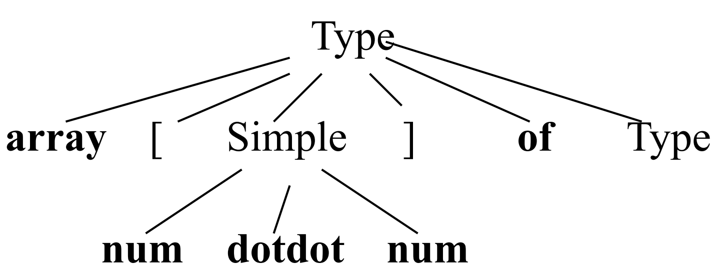



Start symbol

Type → simple
| ↑ **id**
| **array** [ Simple ] **of** Type
Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

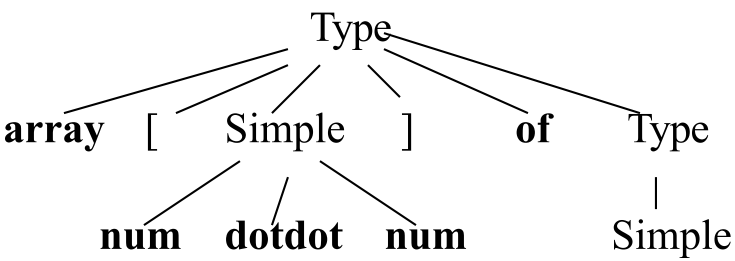
Input :  **array** [ **num dotdot num** ] **of integer**



# Predictive Parsing Example

Lookahead symbol

Input : **array [ num dotdot num ] of integer**

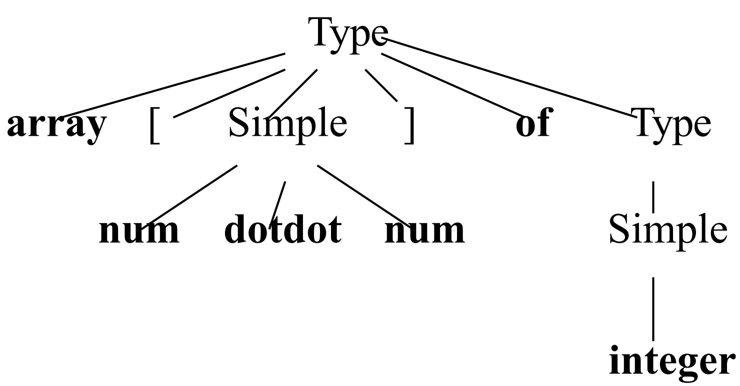



Start symbol

Type → Simple
    | ↑ **id**
    | **array** [ Simple ] **of** Type
Simple → **integer**
    | **char**
    | **num dotdot num**

Lookahead symbol

Input : **array [ num dotdot num ] of integer**



## Predictive Recursive Descent

- Parser is implemented by N + 1 subroutines, where N is the number grammar non-terminals
- There is one subroutine for attempting to Match tokens in the input stream
- There is also one subroutine for each non-terminal with two tasks:
  1. Deciding on the next production to use
  2. Applying the selected production

## Predictive Recursive Descent (Cont.)

Procedure "Match" checks if the token matches the expected input

```
procedure Match ( expected_token ) ;
  {
      if lookahead = expected_token then
            lookahead  :=  get_next_token
      else error
  }
```

# Predictive Recursive Descent (Cont.)

- The subroutine for each non-terminals has two tasks:
  1. Selecting the appropriate production
  2. Applying the chosen production

- Selection is done based on the result of a number of if-then-else statements

- Applying a production is implemented by calling the match procedure or other subroutines, based on the rhs of the production

# Predictive Recursive Descent (Cont.)

Type → Simple
| ↑ **id**
| **array** [ Simple ] **of** Type
Simple → **integer**
| **char**
| **num dotdot num**

Subroutine "Simple" for the given example:

```
procedure Simple {
  if lookahead = integer then call Match ( integer );
  else if lookahead = char then call Match ( char );
  else if lookahead = num then {
    call Match ( num );
    call Match ( dotdot );
    call Match ( num )
  } else error
}
```

# Predictive Recursive Descent (Cont.)

```
procedure Type ;{
  if lookahead is in { integer, char, num } then call Simple;
  else if lookahead = '↑' then {
    call Match ( '↑' ) ;
    call Match( id );
  } else if lookahead = array then {
    call Match( array );
    call Match( '[' );
    call Simple;
    call Match( ']' );
    call Match( of );
    call Type
  } else error
}
```

Type → Simple
  | ↑ **id**
  | **array** [ Simple ] **of** Type
Simple → **integer**
  | **char**
  | **num dotdot num**