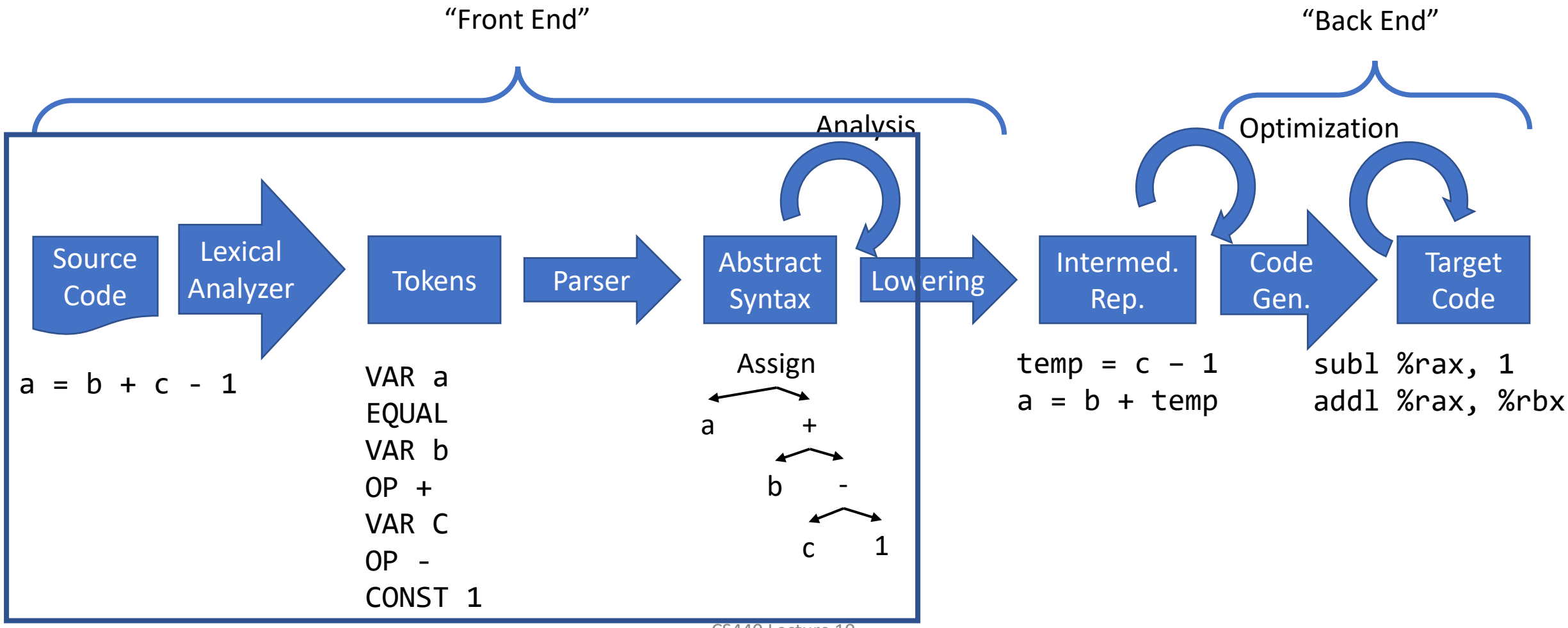
CS440: Programming Languages and Translators

Lecture 10

Compilers translate code in phases



Abstract Syntax

- BNF (Backus-Naur Form)

type ::= int | bool | string | type -> type | type * type | …

bop ::= + | - | * | / | … *uop* ::= ~ | not

exp ::= x | *num* | *"string"* | *exp bop exp* | *uop exp* | *exp exp*

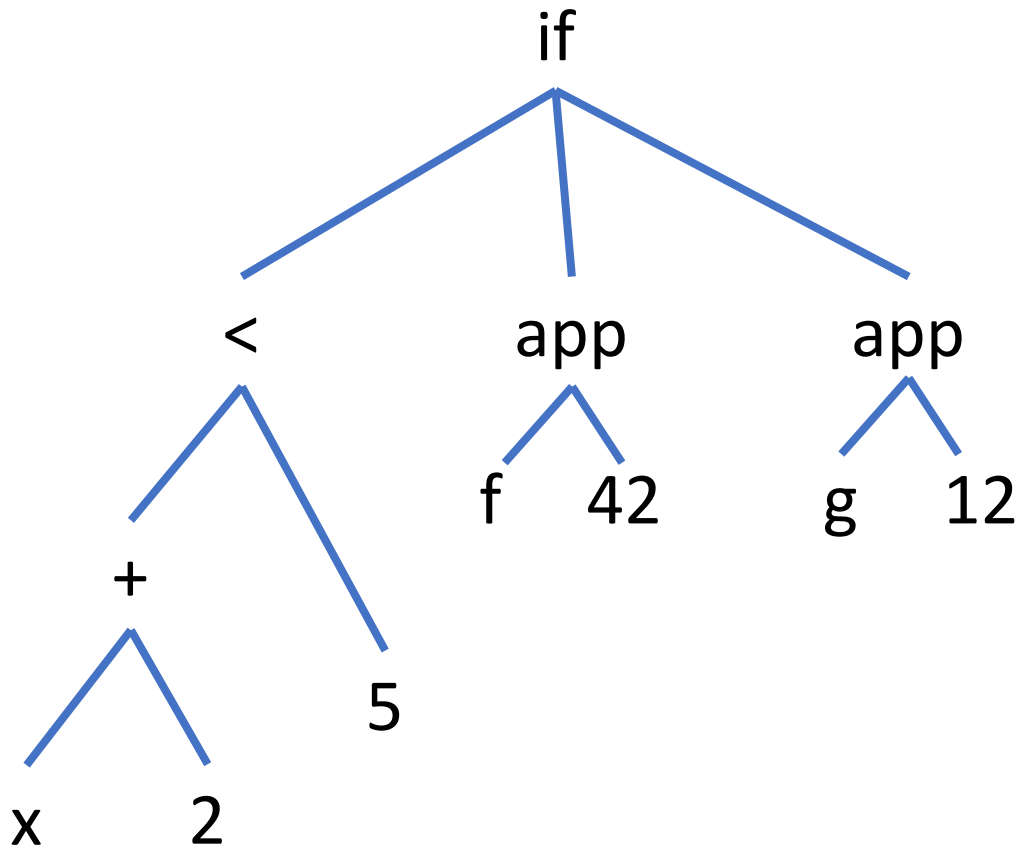
| if *exp* then *exp* else *exp* | let *x* = *exp* in *exp* | fun *x* -> *exp* | …

decl ::= let *x* = *exp* | let *f x* = *exp*

prog ::= ε | *decl;; prog*

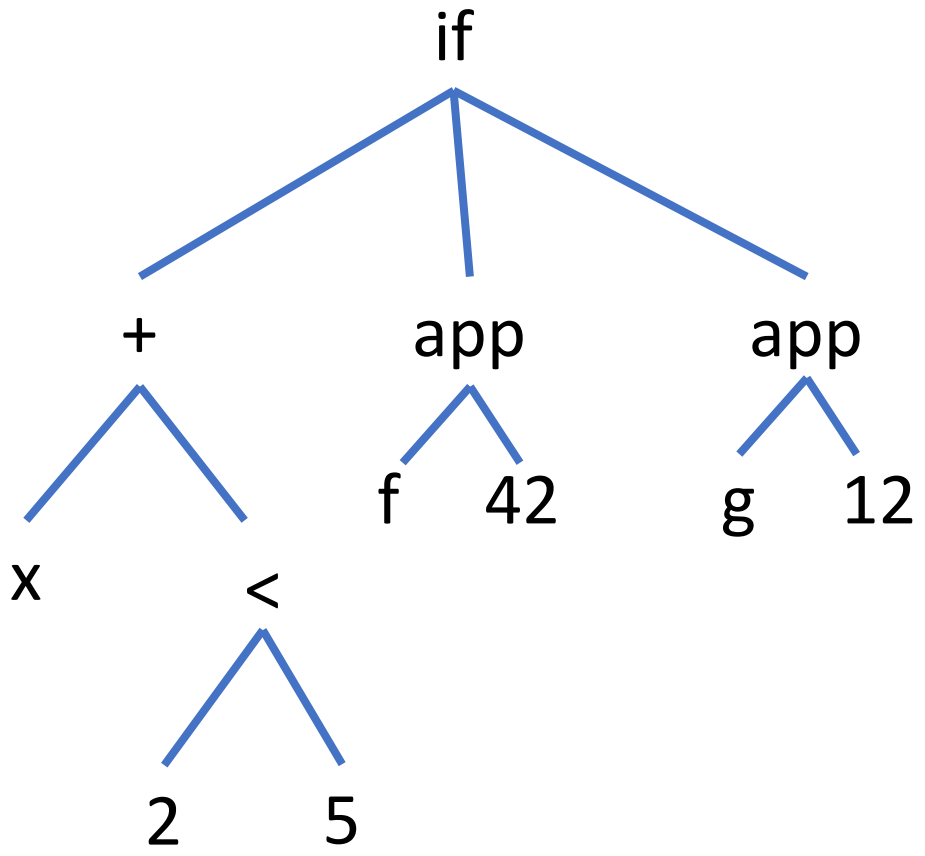
Abstract Syntax Trees (ASTs)

if x + 2 < 5 then f 42 else g 12



Abstract Syntax is not Concrete Syntax

if x + 2 < 5 then f 42 else g 12



Terminology

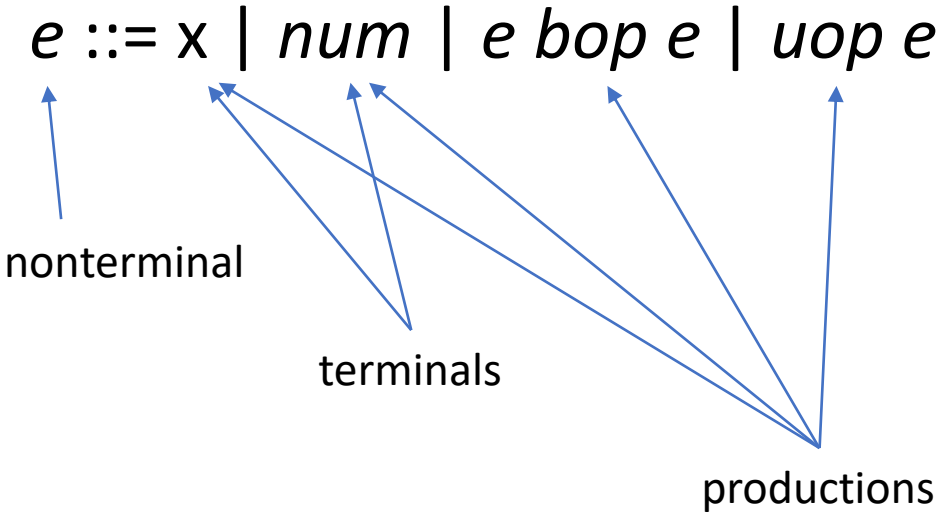
- *Lexical analysis* “lexing”
- Performed by *lexical analyzer* “lexer”
- Produces stream of *tokens*

Lexing examples

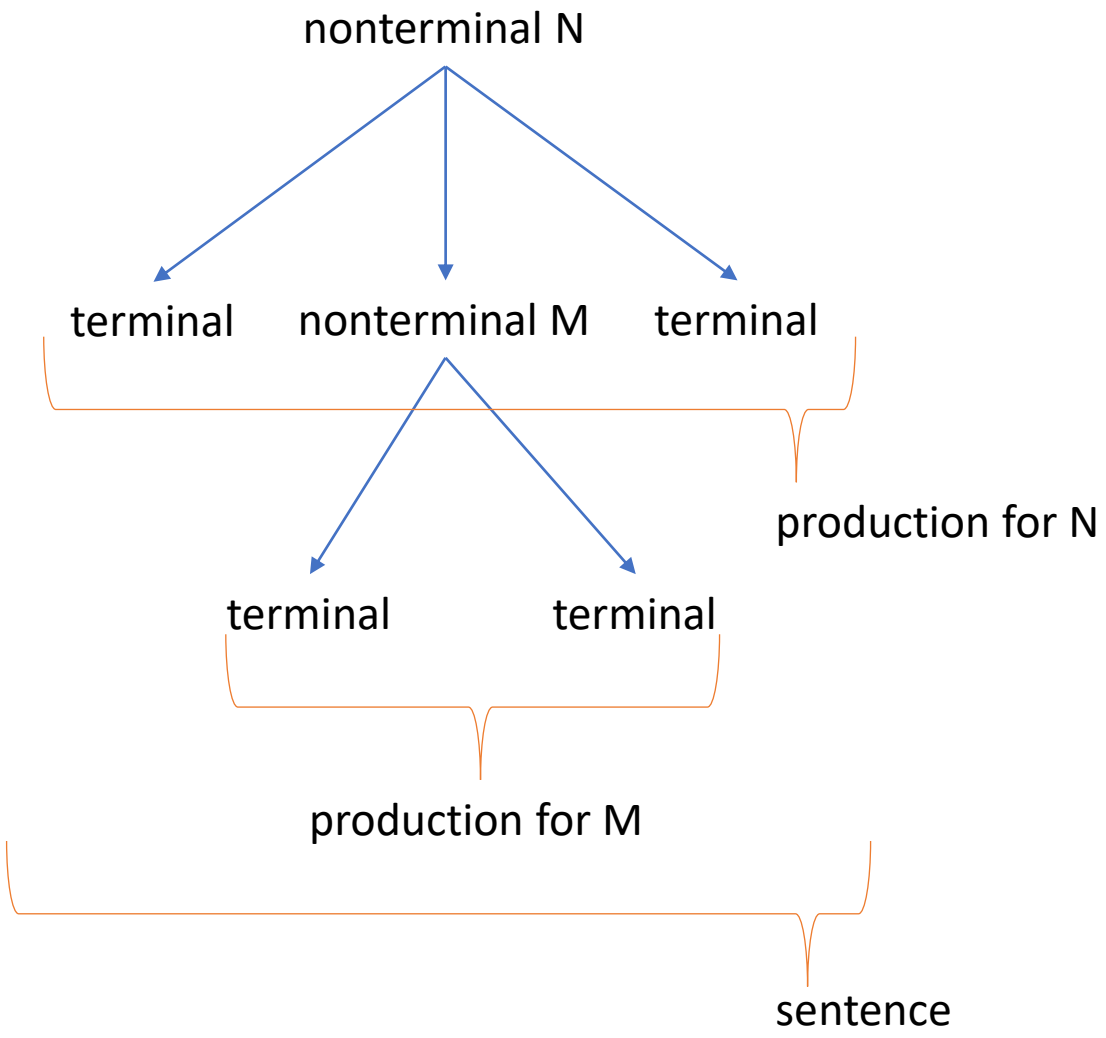
- while (i < 5) WHILE; LPAREN; IDENT “i”; LT; NUM 5; RPAREN
- while i < 5) WHILE; IDENT “i”; LT; NUM 5; RPAREN
- whole (i < 5) IDENT “whole”; LPAREN; IDENT i; LT; NUM 5; RPAREN

Might be syntax errors during *parsing*. *Not* errors during lexing.

BNF grammars are “context-free”

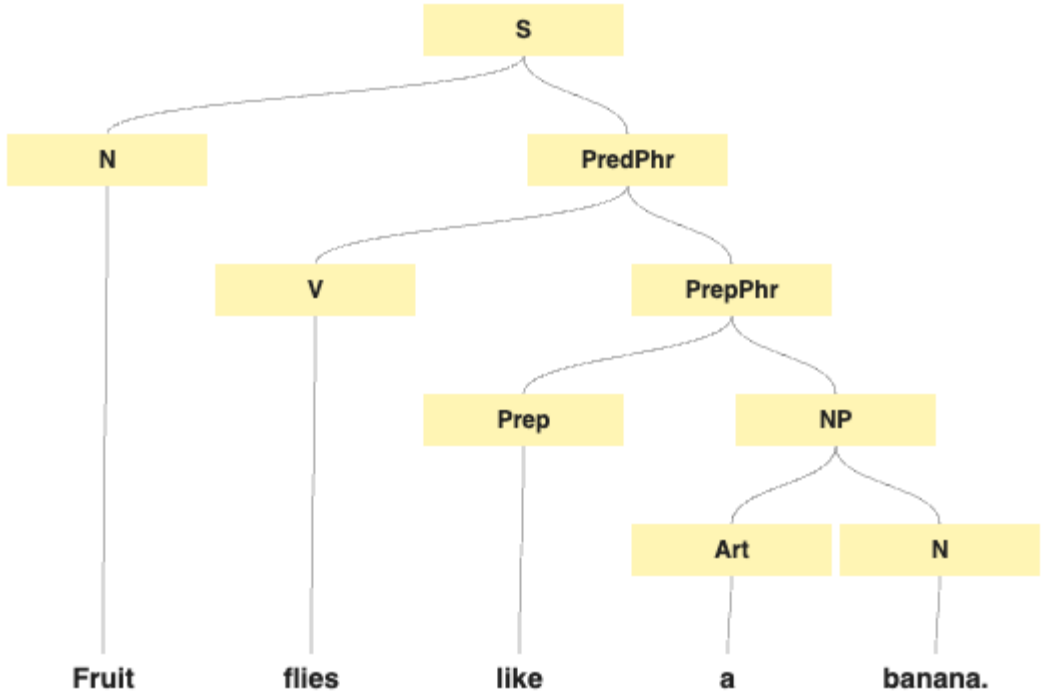


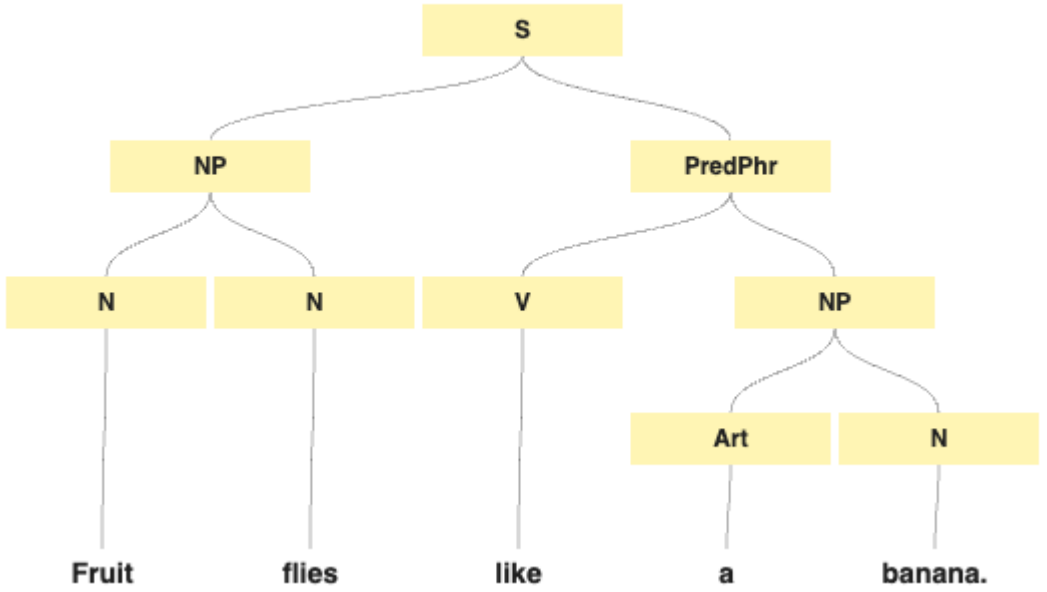
Parsing: Produce a *parse tree* from a stream of tokens



Ambiguous grammars allow multiple correct parse trees

- “Fruit flies like a banana”



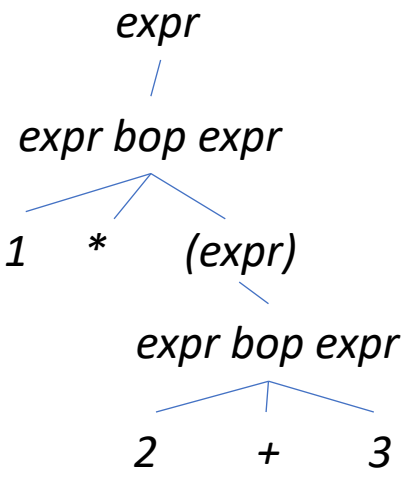


Slide Credit: Hannah Ringler

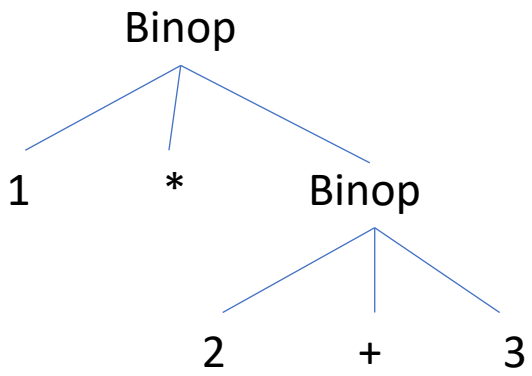
Parse tree vs. AST

1 * (2 + 3)

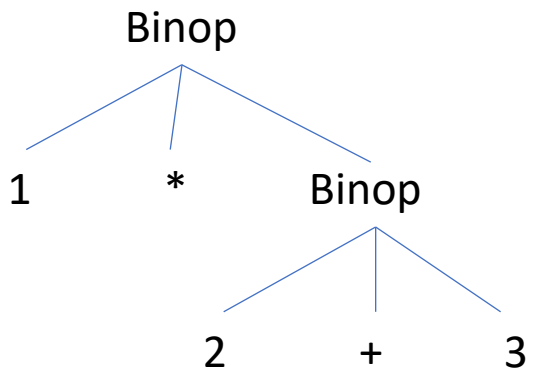
Parse Tree



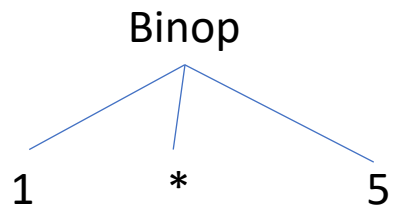
AST



Most basic interpreter: traverse an AST from leaves to root evaluating



Most basic interpreter: traverse an AST from leaves to root evaluating



Most basic interpreter: traverse an AST from leaves to root evaluating

5

Most basic interpreter: traverse an AST from leaves to root evaluating

if 10 < 5 then 1 * 2 else 3 + 4

if false then 1 * 2 else 3 + 4

3 + 4

7

Order matters!

if 0 < 5 then 4 / 2 else 2 / 0

if 0 < 5 then 2 else (raise Divide_by_zero)

raise Divide_by_zero

Order matters!

if 0 < 5 then 4 / 2 else 2 / 0

if true then 4 / 2 else 2 / 0

4 / 2

2

Handling variables: first idea: substitute

```
let x = 5 in let y = x + 1 in x * y
             let y = 5 + 1 in 5 * y
             let y = 6     in 5 * y
                              5 * 6
                              30
```

Substitution isn’t efficient

```
let f = fun x -> x * 2 in [f 1; f 2; f 3; f 4; f 5]
                          [(fun x -> x * 2) 1; (fun x -> x * 2) 2; (fun x -> …
```

Handling variables: second idea: *environment*

let x = 5 in let y = x + 1 in x * y

Handling variables: second idea: *environment*

```
let x = 5 in let y = x + 1 in x * y
            let y = x + 1 in x * y
```

Var	Value
x	5

Handling variables: second idea: *environment*

```
let x = 5 in let y = x + 1 in x * y
             let y = x + 1 in x * y
             let y = 5 + 1 in x * y
             let y = 6     in x * y
```

Var	Value
x	5

Handling variables: second idea: *environment*

```
let x = 5 in let y = x + 1 in x * y
             let y = x + 1 in x * y
             let y = 5 + 1 in x * y
             let y = 6     in x * y
                              x * y
                              5 * 6
                              30
```

Var	Value
x	5
y	6

Environments need to handle shadowing

let x = 1 + 2 in (let x = x + 3 in x + 1) + x

let x = 3 in (let x = x + 3 in x + 1) + x

Var	Value

Environments need to handle shadowing

```
let x = 1 + 2 in (let x = x + 3 in x + 1) + x
let x = 3       in (let x = x + 3 in x + 1) + x
                   (let x = x + 3 in x + 1) + x
                   (let x = 3 + 3 in x + 1) + x
                   (let x = 6       in x + 1) + x
```

Var	Value
x	3

Environments need to handle shadowing

```
let x = 1 + 2 in (let x = x + 3 in x + 1) + x
let x = 3       in (let x = x + 3 in x + 1) + x
                   (let x = x + 3 in x + 1) + x
                   (let x = 3 + 3 in x + 1) + x
                   (let x = 6       in x + 1) + x
                                     (6 + 1) + x
                                           7 + x
```

Var	Value
x	3
x	6

Environments need to handle shadowing

let x = 1 + 2 in (let x = x + 3 in x + 1) + x

let x = 3 in (let x = x + 3 in x + 1) + x

(let x = x + 3 in x + 1) + x

(let x = 3 + 3 in x + 1) + x

(let x = 6 in x + 1) + x

(6 + 1) + x

7 + x

7 + 3 ➔ 10

Var	Value
x	3