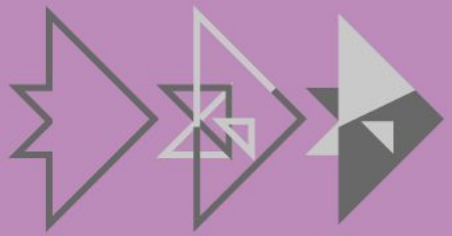
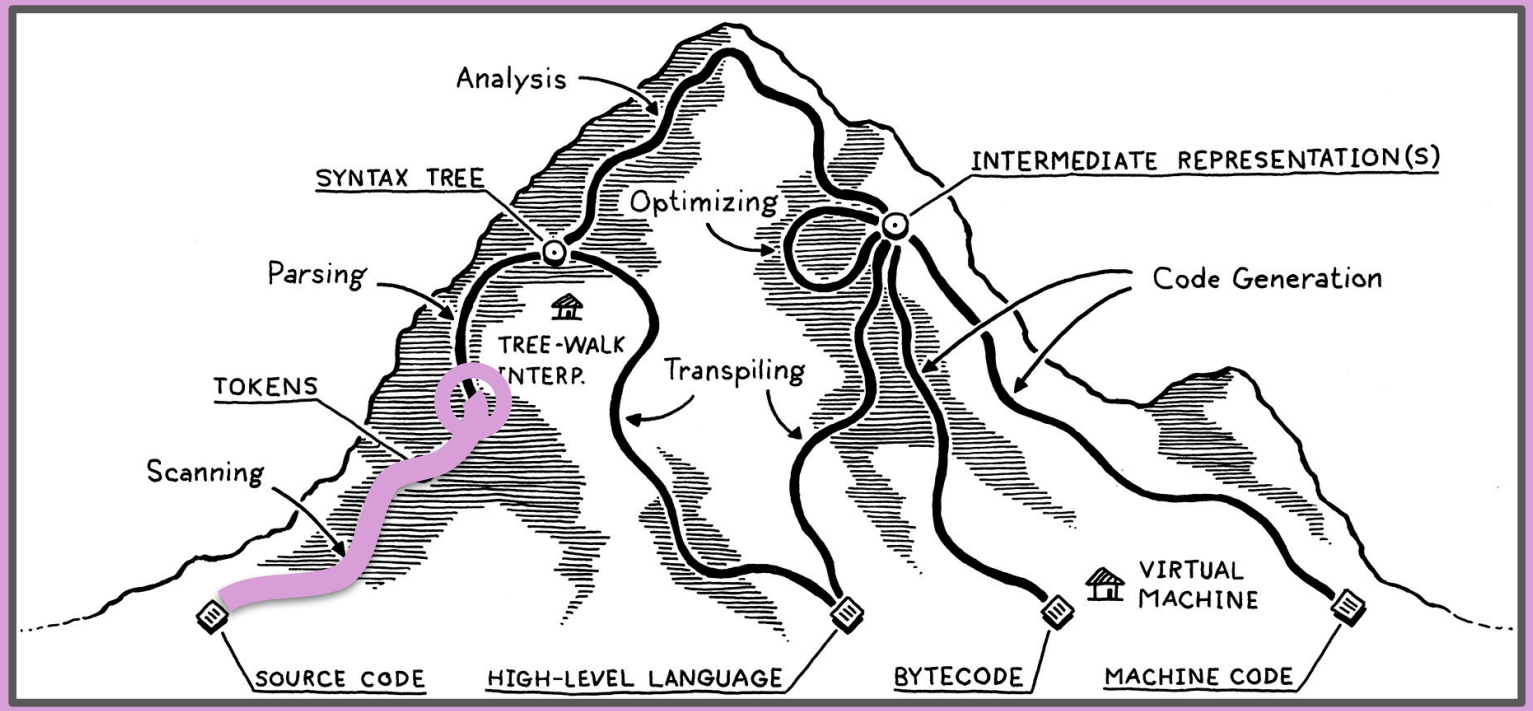


CMPSC 201: PROGRAMMING LANGUAGES


Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK
INTERP.
TOKENS
Transpiling
Scanning
VIRTUAL
MACHINE
SOURCE CODE
HIGH-LEVEL LANGUAGE
BYTECODE
MACHINE CODE

Lexical vs. Syntactic Grammar

	Lexical Grammar
Composed of…	Letters that form valid lexemes
Expressions contain…	Any number of valid lexemes
Are validated by…	Lexical analysis (“Lexing”)

Do I recognize the words?

If so, what do they mean?

Syntactical Analysis (“Parsing”)

Detects a formal language by:

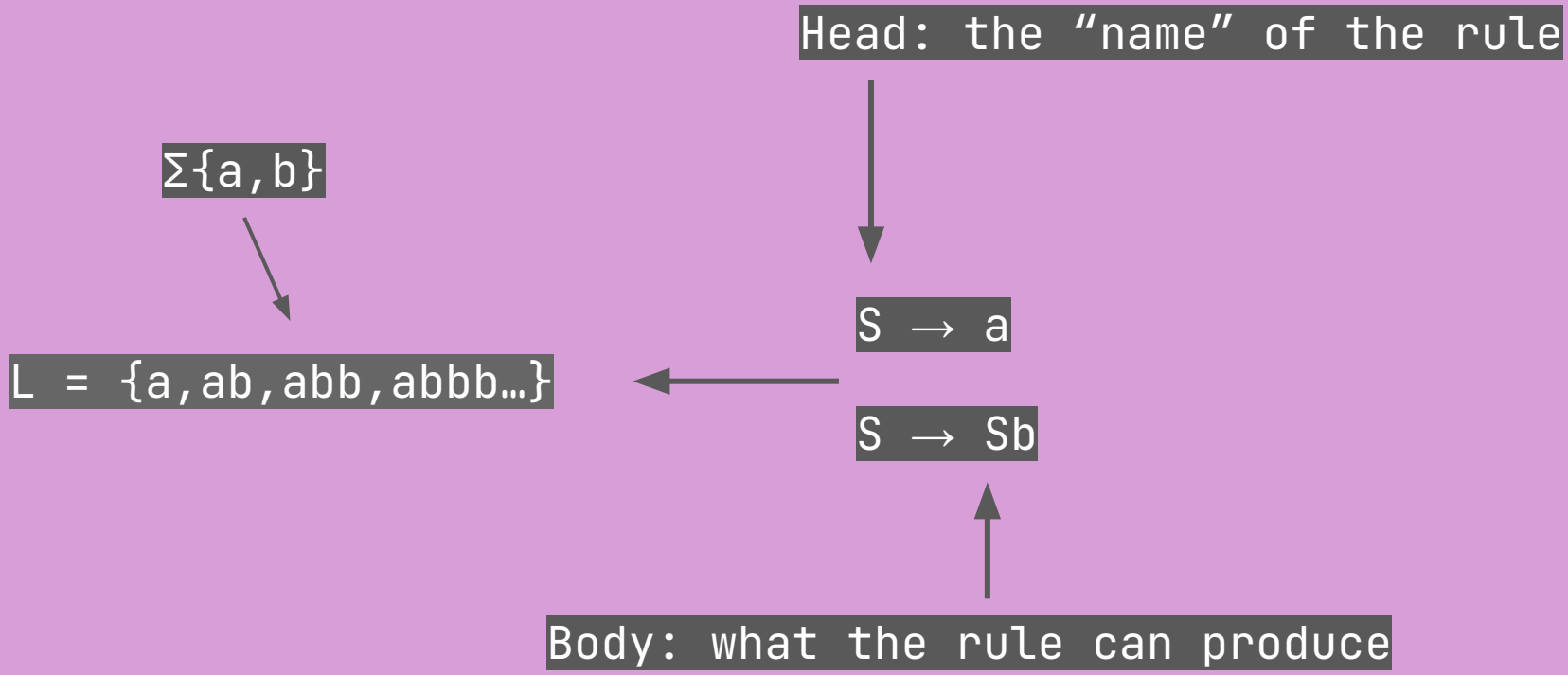
- Validating a set of strings…
- Over a valid alphabet…
- By using determinate production rules…
- Represented by their formal grammar

“Parsing”

Alphabet	Strings	Production Rules
A finite set (Σ), such as: {0,1,2,3,4,5,6,7,8,9} {aardvark,…,zebra} {var,print,…,class,+,-}	Ordered set of elements from a given Σ: Σ = {a,b}	Axioms that produce valid expressions from of given language (L). Each production is called a derivation.

This only works for regular languages!

Production Rules



Production Rules

Symbols:

Terminal: an element from the alphabet with can’t expand

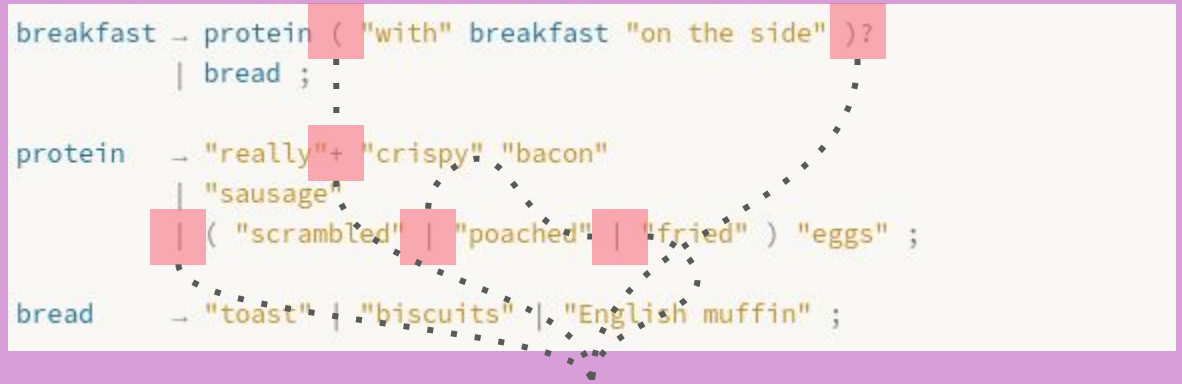
Non-terminal: References another rule of the grammar.

Production Rules

```
breakfast  → protein "with" breakfast "on the side" ;
breakfast  → protein ;
breakfast  → bread ;

protein    → crispiness "crispy" "bacon" ;
protein    → "sausage" ;
protein    → cooked "eggs" ;

crispiness → "really" ;
crispiness → "really" crispiness ;

cooked     → "scrambled" ;
cooked     → "poached" ;
cooked     → "fried" ;

bread      → "toast" ;
bread      → "biscuits" ;
bread      → "English muffin" ;
```

Make a breakfast!

Production Rules



Regular Expression Notation:

(...): Group
?: May or may not be used
+: One instance or more (unbounded)
|: Logical “or”

Lox Grammar and Production

```
expression     → literal      End-of-the-line values
               | unary        Single-operand expressions
               | binary       Expressions composed of 2 operands
               | grouping ;   Any expression bounded by (...)

literal        → NUMBER | STRING | "true" | "false" | "nil" ;
grouping       → "(" expression ")" ;
unary          → ( "-" | "!" ) expression ;
binary         → expression operator expression ;
operator       → "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+"  | "-"  | "*" | "/" ;
```

The Visitor Pattern

Hello, I am wealthy socialite Childs Montgomery Hunton-Blather

The Visitor Pattern

It is the social season, so I am currently taking callers at my residence.

The Visitor Pattern

But! I only accept those callers who have been *invited*.

The Visitor Pattern

Ah! A knock at my door! I must send Jeeves to see if this visitor is strictly *invited*.

VERY LOUD KNOCKING

The Visitor Pattern

First, let’s look at our guest list for whom to expect, tut tut.

```
abstract class Socialite{
     interface Visitor<T> {
          T visitChildsHome(person);
     }
}
```

The Visitor Pattern

It looks like anyone who has permission to visitChildsHome can come see me!

```
abstract class Socialite{
    interface Visitor<T> {
        T visitChildsHome(person);
    }
}
```

The Visitor Pattern

First, let us examine their invitation.

I accept(Duckworth)!

Duckworth Charmington V

```
public String visitChildsHome{
     return “Yes, hrmph, indeed;”
}
```

The Visitor Pattern

Oh, good! Duckworth! Jolly good. Always glad to see a friend.

Duckworth Charmington V

```
public String visitChildsHome{
    return “Yes, hrmph, indeed!”;
}
```

The Visitor Pattern

Another knock! I am a popular socialite today!

VERY LOUD KNOCKING

The Visitor Pattern

First, let us examine their invitation.

Burt
Chimney Sweep

The Visitor Pattern

That’s not an invitation at all! I do not recognize you as a visitor, sir!

Burt
Chimney Sweep

The Visitor Pattern

Oh, I didn't see that you were a chimney sweep! You are in luck, friend, you can visit me for jobs!

Burt
Chimney Sweep

```
public String visitForJobs{
     return "Alright, guv'nor!";
}
```

accept(Burt)

Alright, guv'nor!

The Visitor Pattern

Let’s add that to my list of social calls.

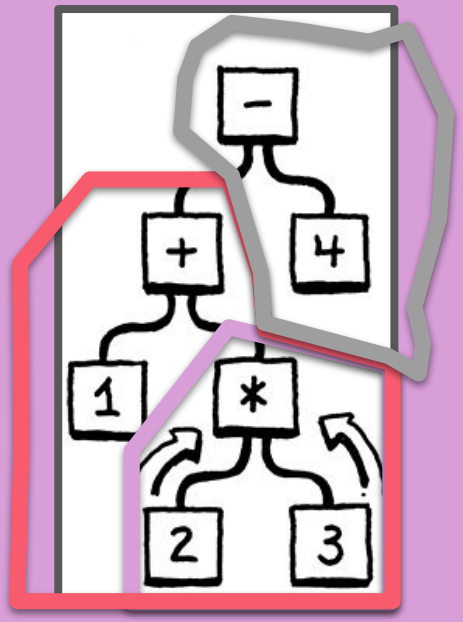
```
abstract class Socialite{
    interface Visitor<T> {
        T visitChildsHome(person);
        T visitForJobs(person);
    }
}
```

The Visitor Pattern

I am growing in influence!

```
abstract class Socialite{
    interface Visitor<T> {
        T visitChildsHome(person);
        T visitForJobs(person);
    }
}
```

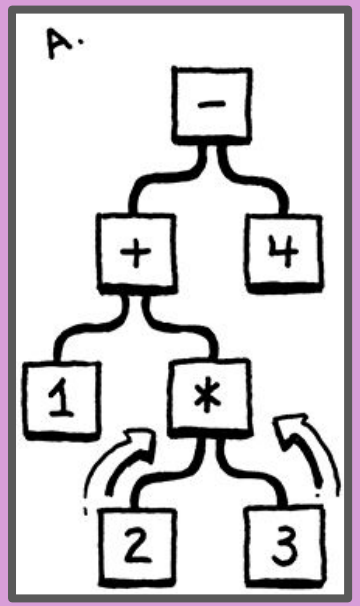
Post-Order Traversal

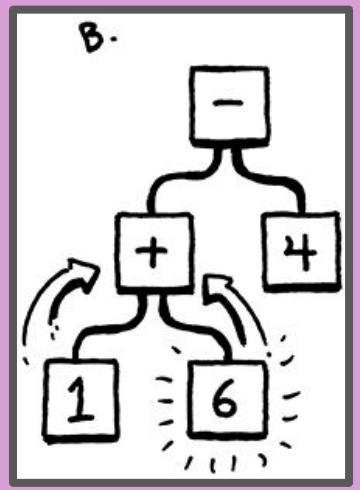


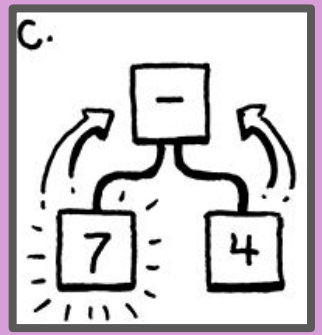

```
1 + 2 * 3 - 4
```

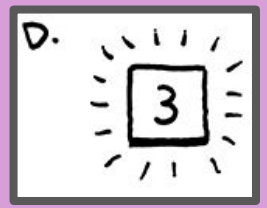
Post-Order Traversal

`1 + (2 * 3) - 4`









Post-Order Traversal

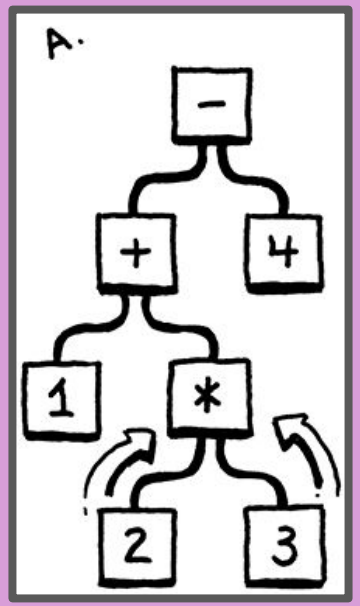
```
1 + (2 * 3) - 4
```

Normal Polish notation (NPN)

```
(- (+ 1.0 ((* 2.0 3.0))) 4.0)
```

Reverse Polish notation (RPN)

```
1.0 2.0 3.0 * + 4.0 -
```



Post-Order Traversal (RPN)

Hint: think in binaries!

1 + (2 * 3) - 4

2

1.0 2.0 3.0 * + 4.0 -

1

3