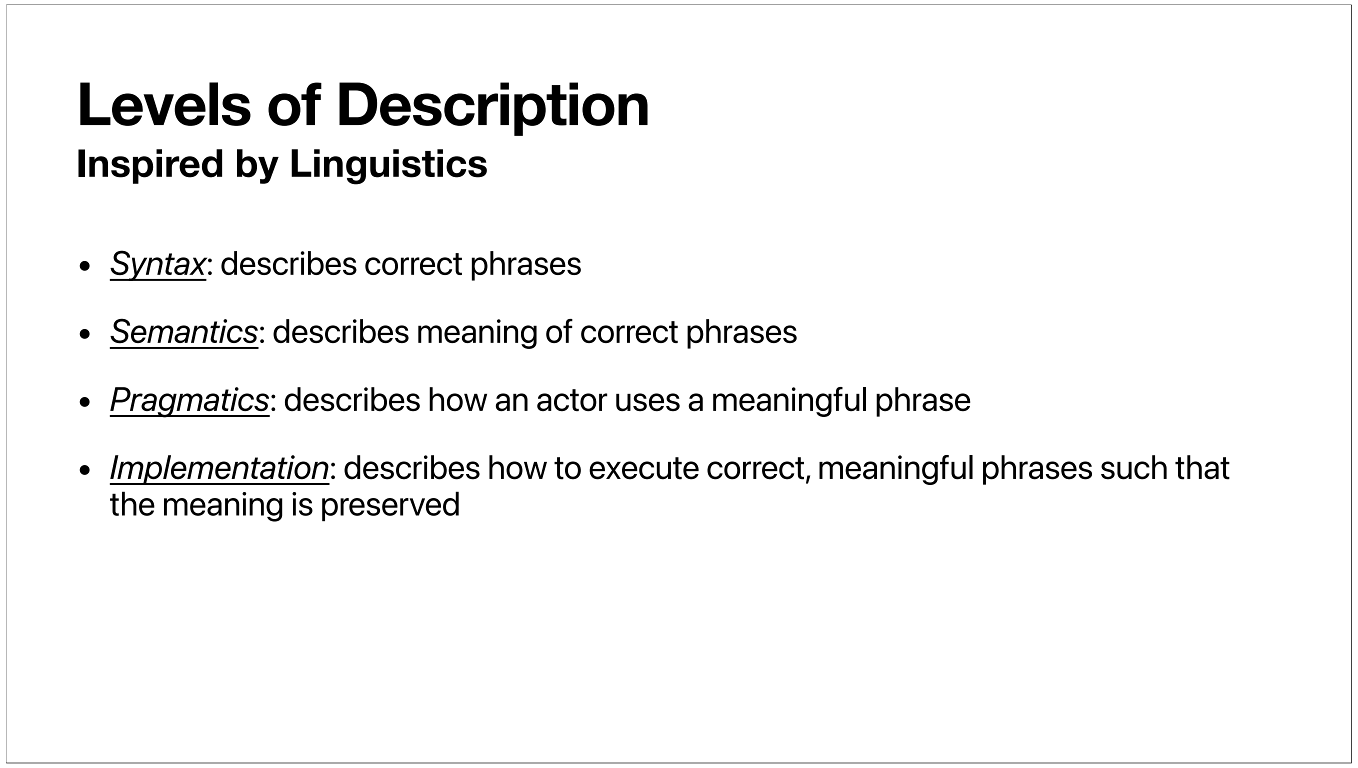
Syntax and Semantics

Defining Programming Languages

CSCI 305
Prof. Matt Revelle



Levels of Description

Inspired by Linguistics

- *Syntax*: describes correct phrases
- *Semantics*: describes meaning of correct phrases
- *Pragmatics*: describes how an actor uses a meaningful phrase
- *Implementation*: describes how to execute correct, meaningful phrases such that the meaning is preserved

Programming languages are languages.

In Linguistics, three areas were identified of how a language can be identified: syntax (or grammar), semantics, and pragmatics.

For programming languages, we might also consider the implementation of the language. Does the implementation preserve the semantics?

Grammar and Syntax

- A *grammar* provides the *alphabet* and rules and phrases that are correct for a language
- Given an alphabet of symbols and *tokens*, *syntax* describes which sequences are valid phrases

tacocat

Language syntax is defined by a grammar

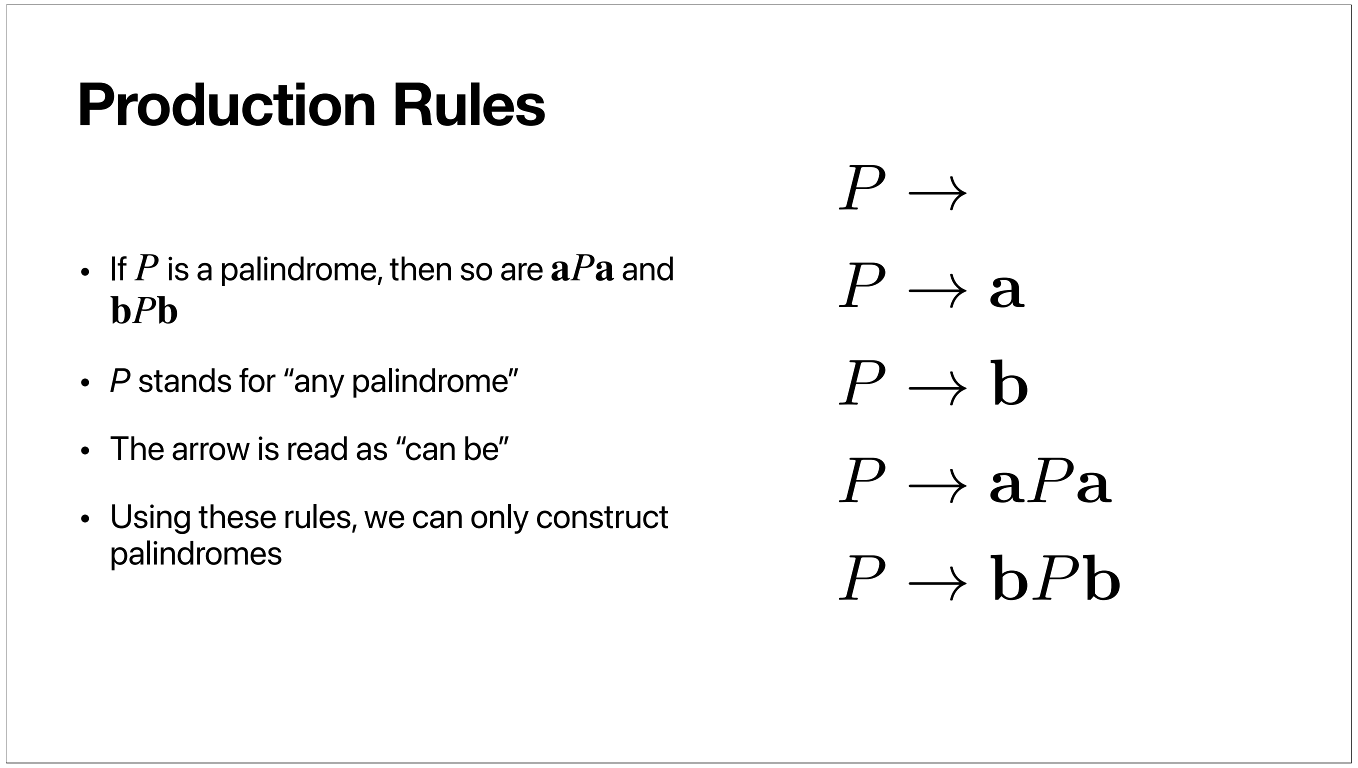
An alphabet consists of a set of symbols.

Describe Syntax

Palindromes

- Define the language of palindromic strings
- Start with an alphabet, $A = \{a, b\}$
- Select all palindromic strings over A
- If P is a palindrome, then so are $\mathbf{a}P\mathbf{a}$ and $\mathbf{b}P\mathbf{b}$
- How can we formalize this idea?

a
aba
bb
abbbaaabbba



Production Rules

- If P is a palindrome, then so are $\mathbf{a}P\mathbf{a}$ and $\mathbf{b}P\mathbf{b}$
- *P* stands for "any palindrome"
- The arrow is read as "can be"
- Using these rules, we can only construct palindromes

$$P \to$$
$$P \to \mathbf{a}$$
$$P \to \mathbf{b}$$
$$P \to \mathbf{a}P\mathbf{a}$$
$$P \to \mathbf{b}P\mathbf{b}$$

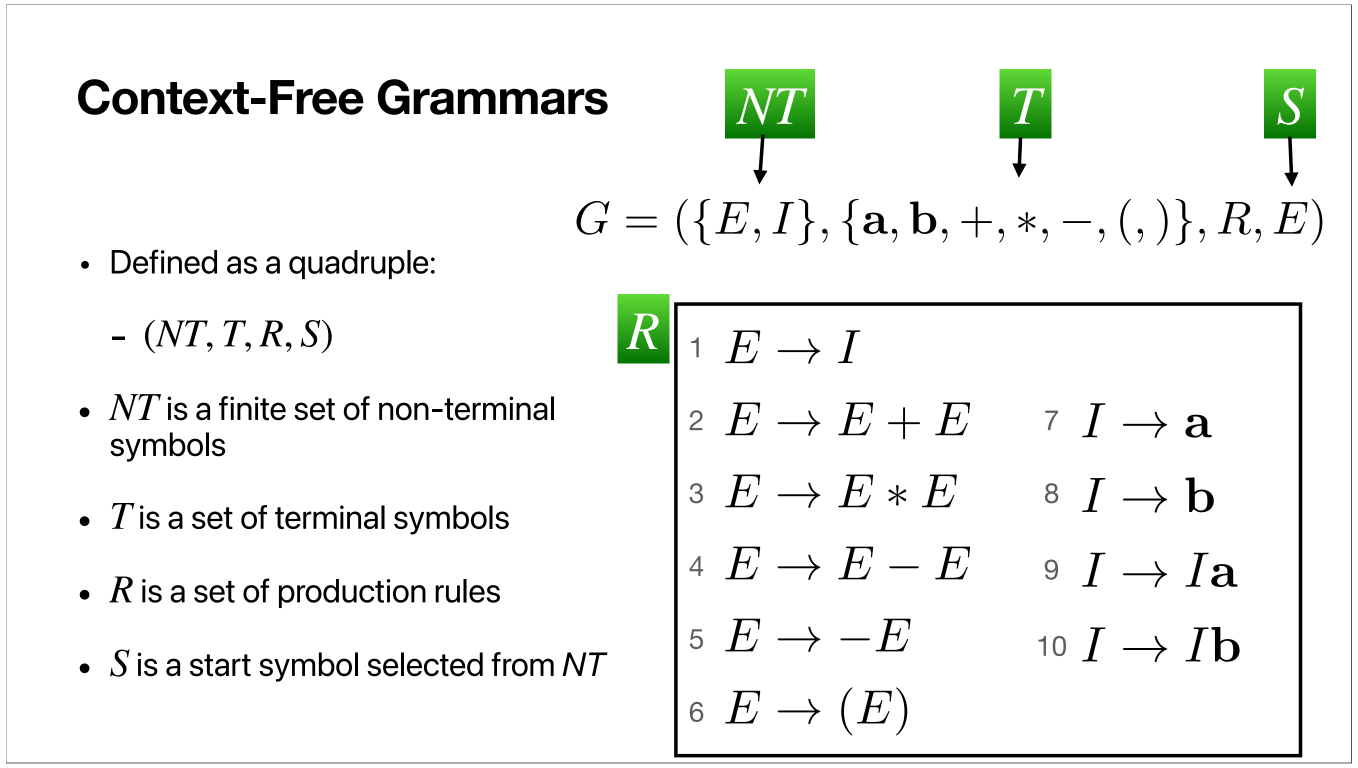
On the left-hand side we have a single non-terminal symbol

On the right-hand side we can have a combination of non-terminal and terminal symbols.

We can take a moment to convince ourselves that these production rules allow us to only construct palindromes.

Inductively, if P is a palindrome, then surrounding P with a pair of a's or b's will also produce a palindrome.

When we finally stop growing our palindrome, we choose P to be either the empty string, "a", or "b".



Context-Free Grammars

- Defined as a quadruple:
  - (NT, T, R, S)
- NT is a finite set of non-terminal symbols
- T is a set of terminal symbols
- R is a set of production rules
- S is a start symbol selected from NT

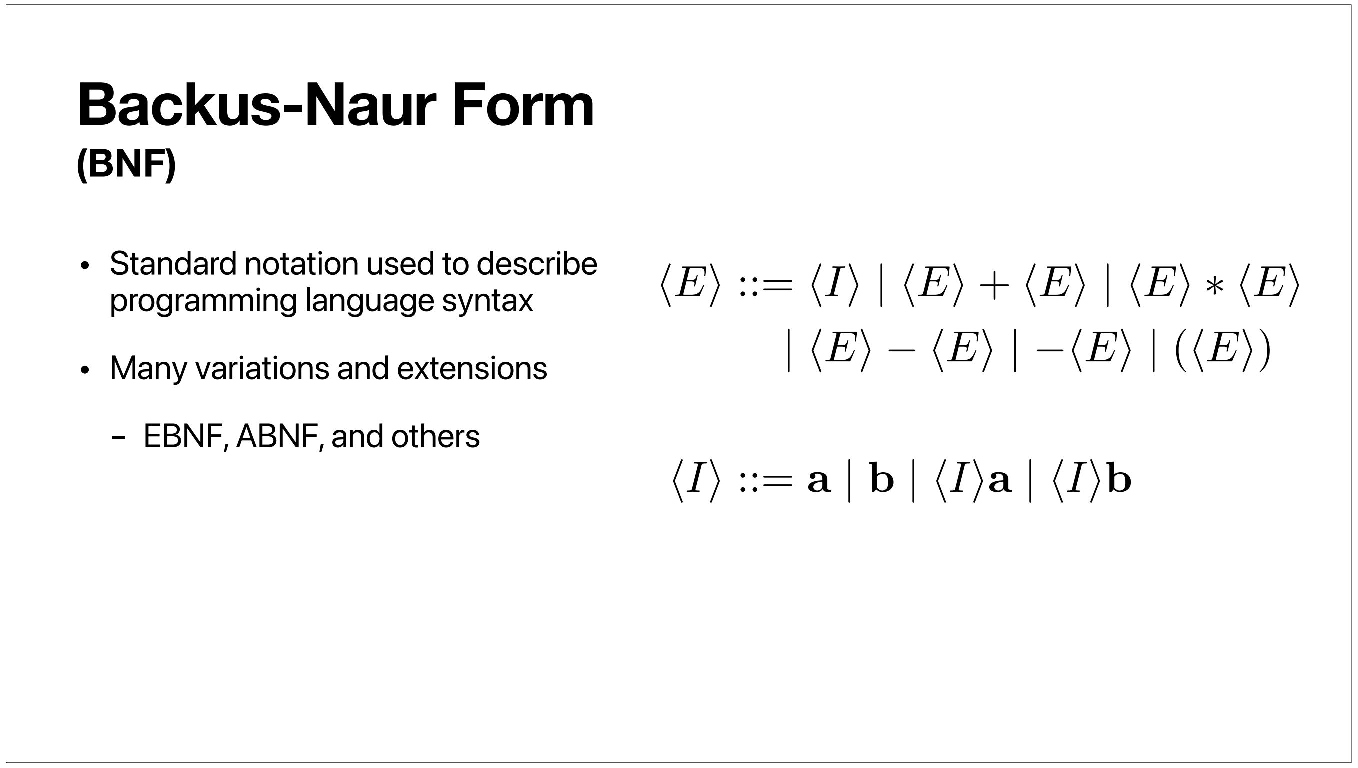
$$G = (\overset{NT}{\downarrow}\{E, I\}, \overset{T}{\downarrow}\{\mathbf{a}, \mathbf{b}, +, *, -, (,)\}, R, \overset{S}{\downarrow}E)$$

R

1	$E \rightarrow I$		
2	$E \rightarrow E + E$	7	$I \rightarrow \mathbf{a}$
3	$E \rightarrow E * E$	8	$I \rightarrow \mathbf{b}$
4	$E \rightarrow E - E$	9	$I \rightarrow I\mathbf{a}$
5	$E \rightarrow -E$	10	$I \rightarrow I\mathbf{b}$
6	$E \rightarrow (E)$		

Context-free grammars are commonly used to define the syntax of programming languages.

Note that the syntax of many programming languages cannot be fully represented by a context-free grammar. More on this later.



Backus-Naur Form
(BNF)

- Standard notation used to describe programming language syntax
- Many variations and extensions
  - EBNF, ABNF, and others

$$\begin{aligned}\langle E\rangle &::= \langle I\rangle \mid \langle E\rangle + \langle E\rangle \mid \langle E\rangle * \langle E\rangle \\ &\quad \mid \langle E\rangle - \langle E\rangle \mid -\langle E\rangle \mid (\langle E\rangle)\end{aligned}$$

$$\langle I\rangle ::= \mathbf{a} \mid \mathbf{b} \mid \langle I\rangle\mathbf{a} \mid \langle I\rangle\mathbf{b}$$

BNF is a more compact way to represent production rules.

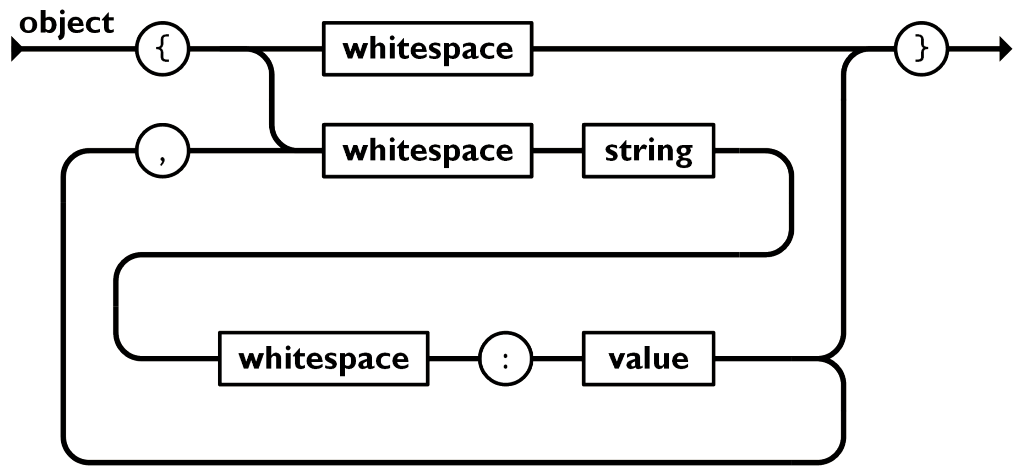
BNF was introduced to define the syntax of the Algol60 programming language. Backus and Naur were two members of the Algol committee.

All expansions from the same head (non-terminal) are grouped together and separate by a vertical bar.

Non-terminal symbols are surrounded by angle brackets and terminal symbols may be bolded or surrounded in quotation marks.

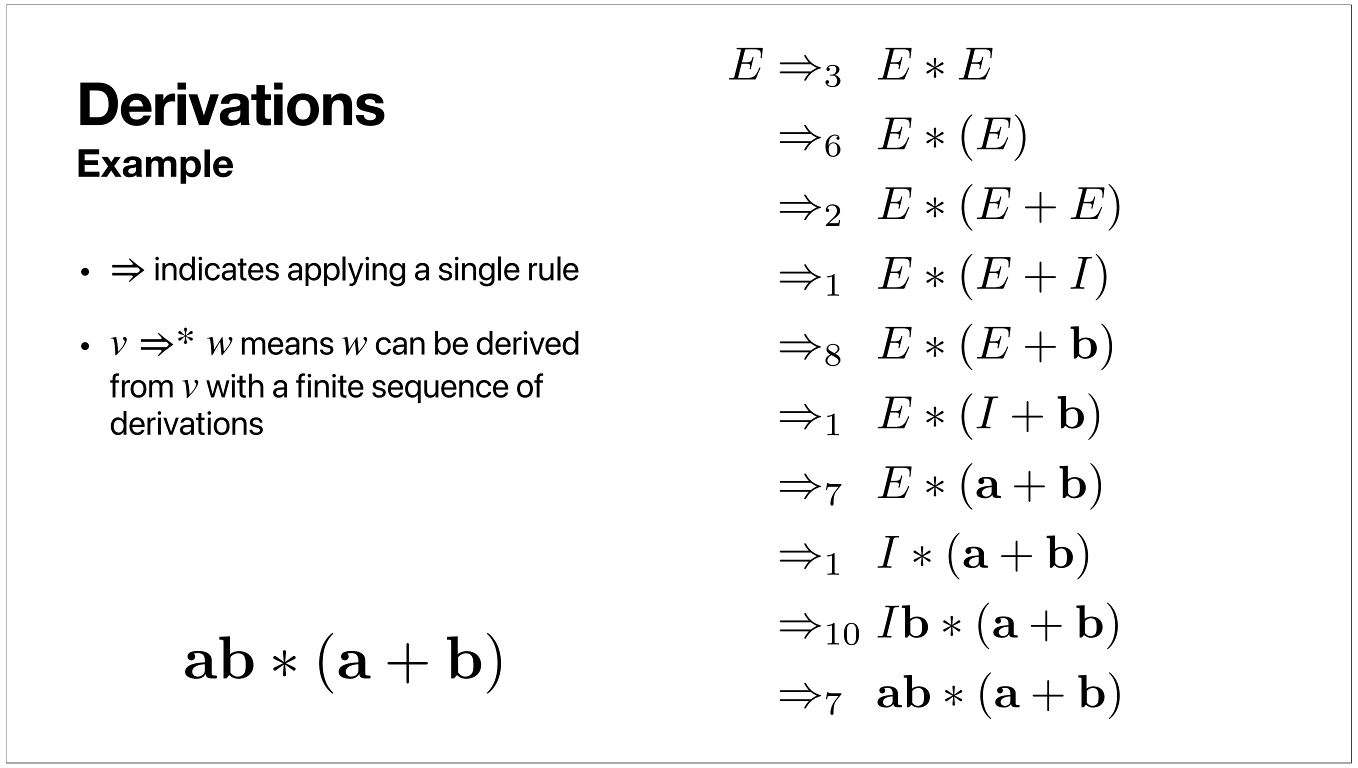
It’s not uncommon to see variations of this form.

Syntax Diagrams



Syntax diagrams are a more-visual alternative to BNF.

This is a syntax diagram for JSON objects. E.g., {“foo”: 42, “bar”: “a string value”}



Derivations

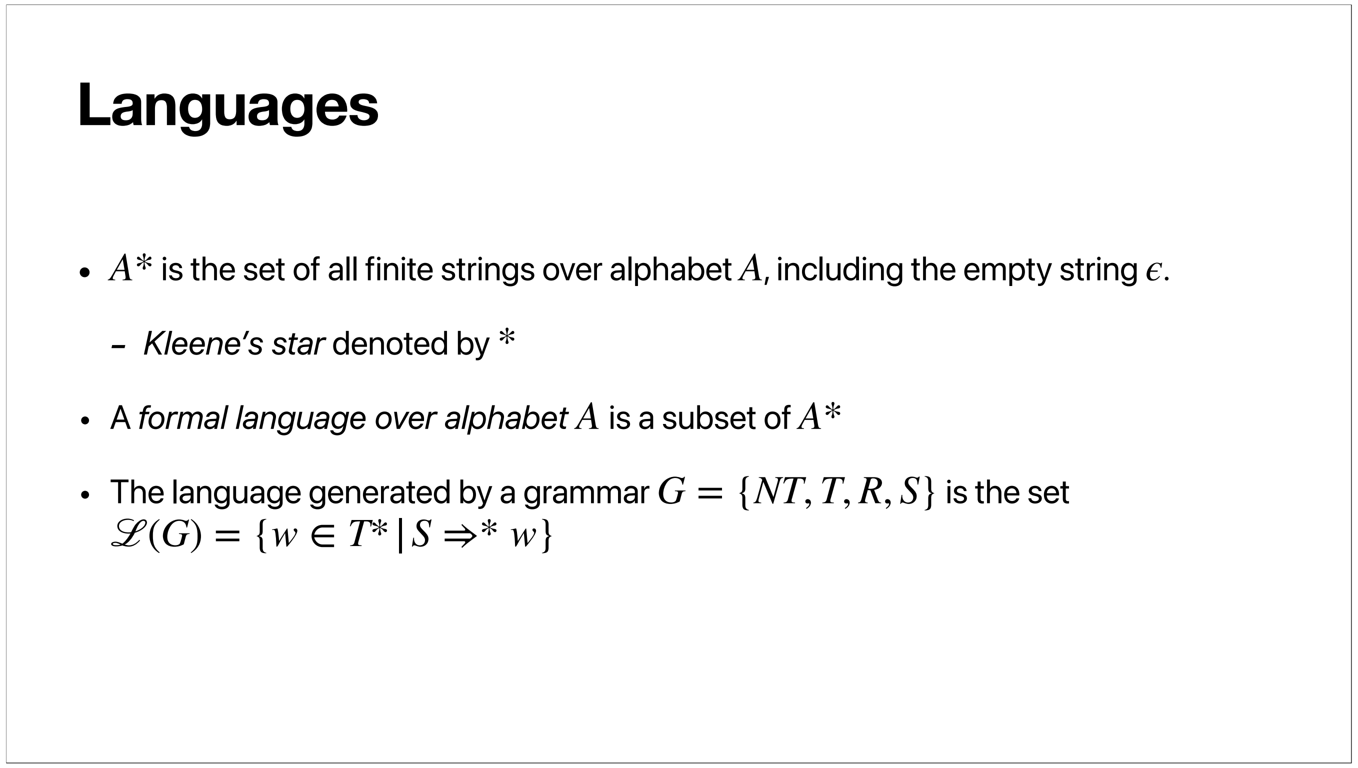
Example

- $\Rightarrow$ indicates applying a single rule
- $v \Rightarrow^* w$ means w can be derived from v with a finite sequence of derivations

$$\mathbf{ab} * (\mathbf{a} + \mathbf{b})$$

$$\begin{aligned}
E &\Rightarrow_3 E * E \\
&\Rightarrow_6 E * (E) \\
&\Rightarrow_2 E * (E + E) \\
&\Rightarrow_1 E * (E + I) \\
&\Rightarrow_8 E * (E + \mathbf{b}) \\
&\Rightarrow_1 E * (I + \mathbf{b}) \\
&\Rightarrow_7 E * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_1 I * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_{10} I\mathbf{b} * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_7 \mathbf{ab} * (\mathbf{a} + \mathbf{b})
\end{aligned}$$

Here is a sequence of rules that can be used to produce the string shown here by started at the start symbol E.



Languages

- A^* is the set of all finite strings over alphabet A, including the empty string ϵ.
  - *Kleene's star* denoted by $*$
- A *formal language over alphabet* A is a subset of A^*
- The language generated by a grammar $G = \{NT, T, R, S\}$ is the set $\mathscr{L}(G) = \{w \in T^* \mid S \Rightarrow^* w\}$

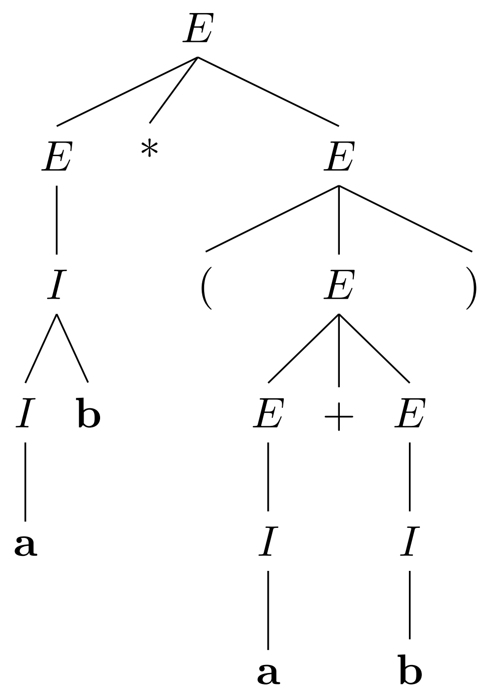
We can then define a language for a grammar as being a subset of all finite sequences .

Parse Trees

aka Derivation Trees

- Each branch is the application of a rule
- Leaves correspond to the terminal symbols
- For this string, applying the rules in different orders results in the same tree

$$\mathbf{ab} * (\mathbf{a} + \mathbf{b})$$

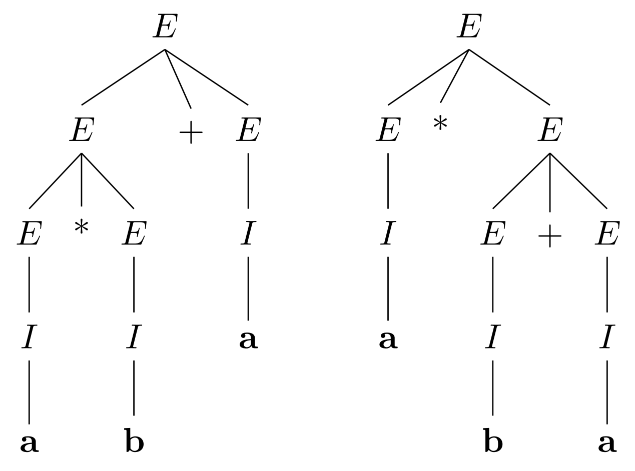


- A tree consists of *nodes* and *arcs*.
- The number of arcs is one less than the number of nodes.
- A tree T is *connected*, that is there is a path through the tree connecting every pair of nodes
- The *root* node is the node at the top of the tree.
- Tree has *levels* which are defined based on distance from root, starting at level 0.
- Nodes below another node are called *children* and the node above is their *parent*
- Nodes with the same parent are called *siblings*
- Nodes without children are called *leaves*

Parsing Ambiguity

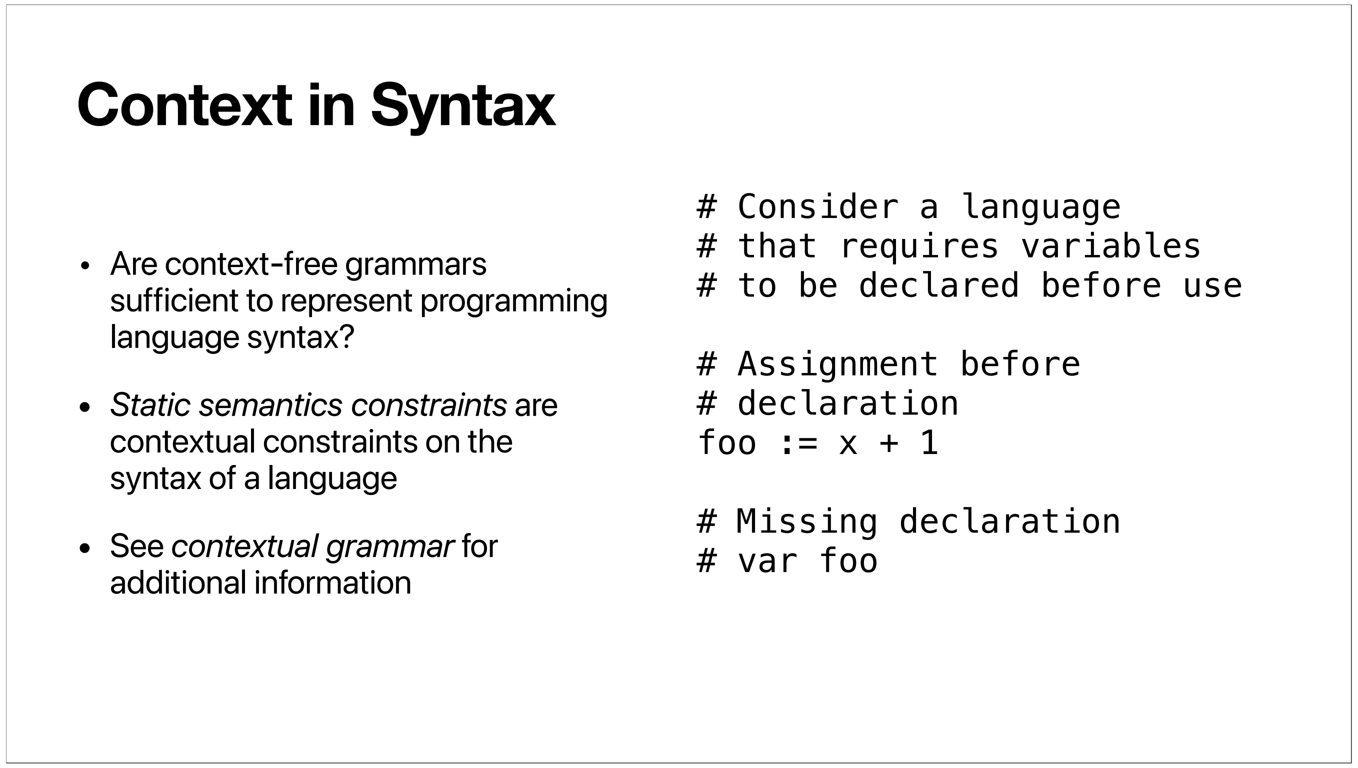
$$\mathbf{a} * \mathbf{b} + \mathbf{a}$$

- The same string can result in different parse trees
  - Our grammar is ambiguous
- It is possible to to disambiguate a grammar
- Check out *CSCI 468: Compilers* to learn more



The ambiguity in this grammar is caused by not capturing the precedence of the multiplication and addition operators.

In the left tree, “a * b” is evaluated first, and then added to “a”
In the right tree, “b + a” are added first and then multiplied by “a”



Context in Syntax

- Are context-free grammars sufficient to represent programming language syntax?
- *Static semantics constraints* are contextual constraints on the syntax of a language
- See *contextual grammar* for additional information

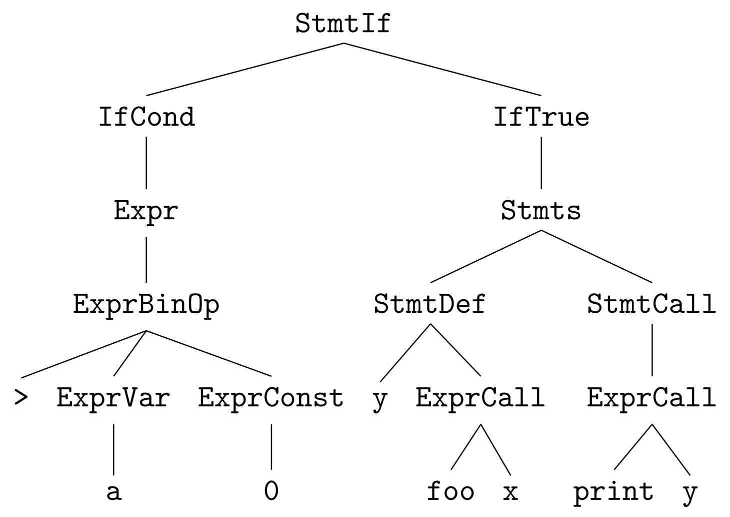
```
# Consider a language
# that requires variables
# to be declared before use

# Assignment before
# declaration
foo := x + 1

# Missing declaration
# var foo
```

- The syntax of this language requires that a variable is declared before being assigned.
- An assignment without declaration is then syntactically-invalid for this language
- The grammar alone is not enough to describe the language…but it gets close!
- Not unlike ChatGPT and other LLM-based approaches which learn a sort of probabilistic grammar of a language

Abstract Syntax Trees

(ASTs)

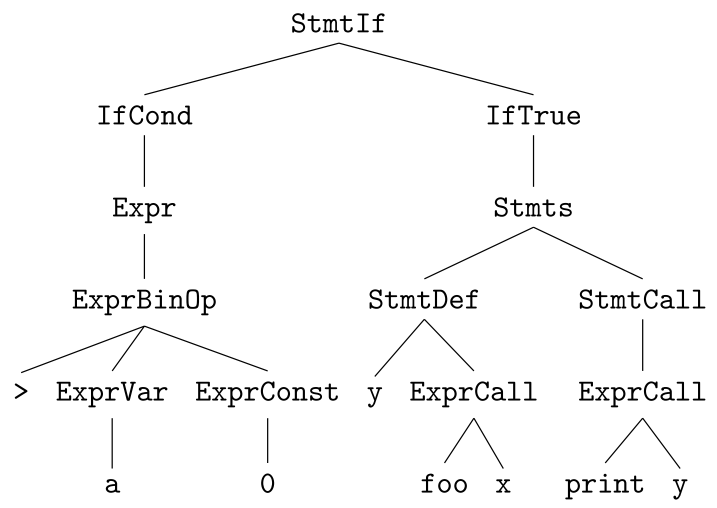
- Parse trees that satisfy the static semantic analysis are *abstract syntax trees*
- ASTs may be simplified/optimized and then later compiled, interpreted, or evaluated
- We can apply the language semantics to ASTs in order to interpret them



What might the concrete syntax for the code snippet shown here as an AST look like in a Python-like language be given this AST?

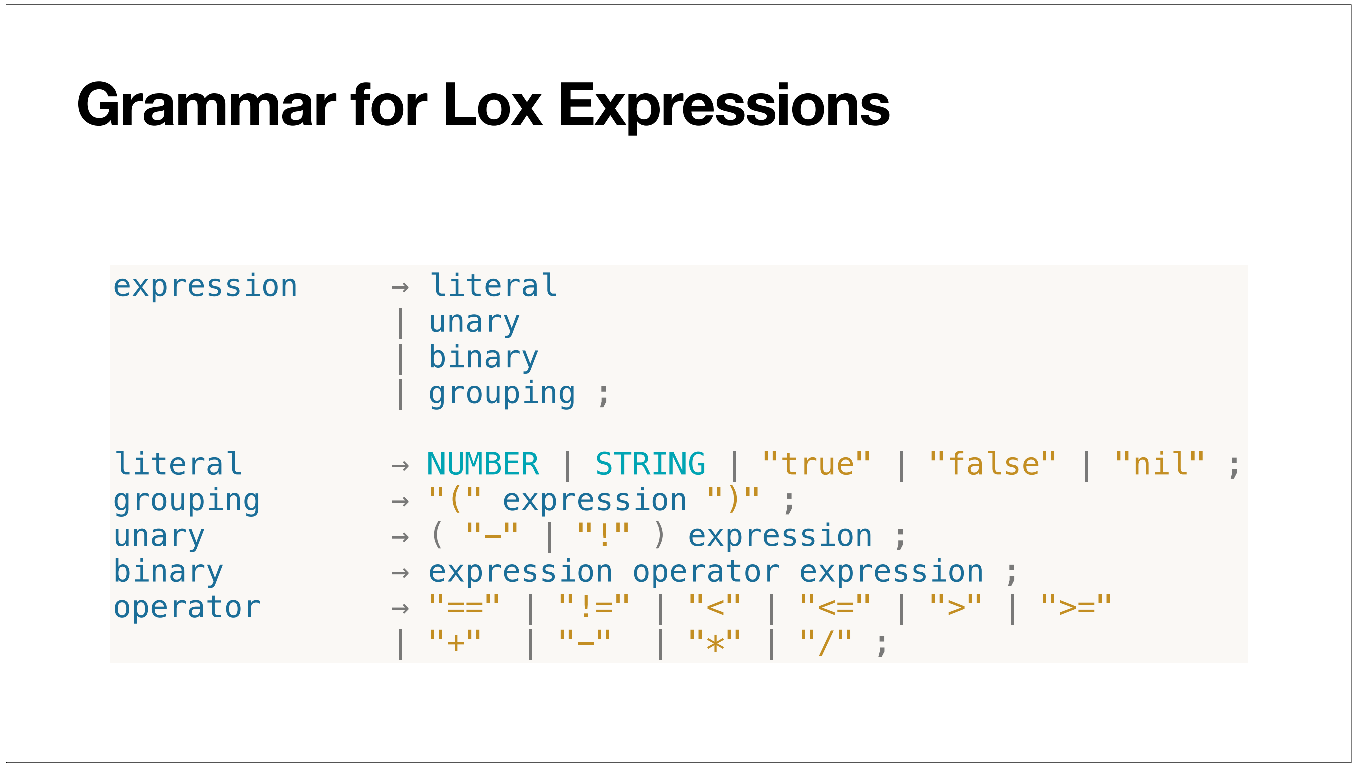
Abstract Syntax Trees

(ASTs)



```
if (a > 0):
 y = foo(x)
 print(y)
```

Note that there are many different concrete syntaxes that could result in the same abstract syntax tree.
E.g., code written in Python, C, Rust, and many others could be parsed into an AST that looks like this.



Grammar for Lox Expressions

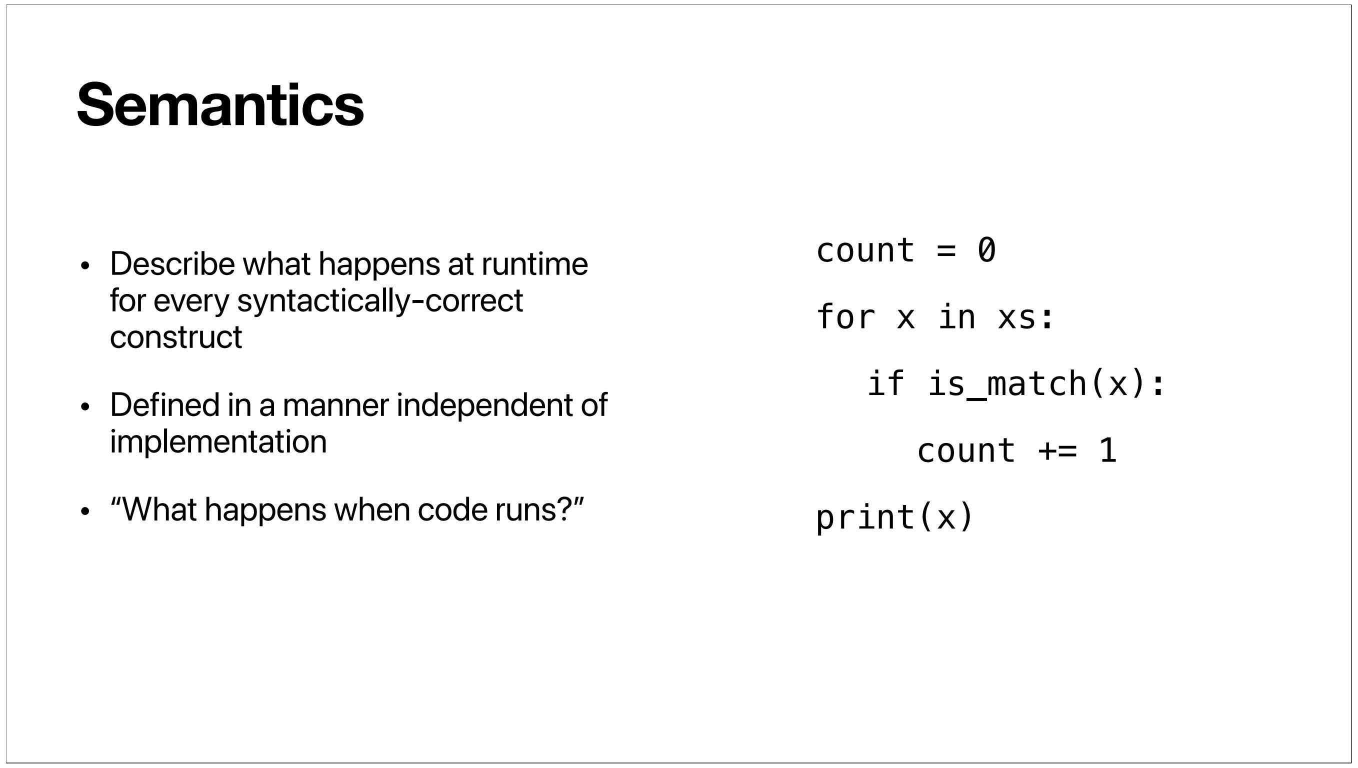
```
expression     → literal
               | unary
               | binary
               | grouping ;

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

The full grammar is a bit more involved.

But this is the grammar which corresponds to the data types you all are implementing.

You are defining the data types which are used to build ASTs for Lox expressions

We will later be adding support for statements, functions, and classes.



Semantics

- Describe what happens at runtime for every syntactically-correct construct
- Defined in a manner independent of implementation
- “What happens when code runs?”

```
count = 0
for x in xs:
  if is_match(x):
    count += 1
print(x)
```

In the example code we have an integer variable (count) that is conditionally incremented.

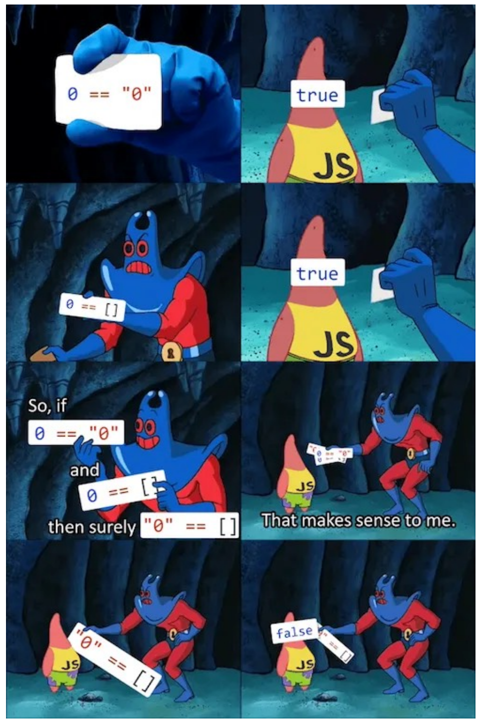
The for loop iterates over a collection of items (xs)

There are two function calls, what happens when a function is called?

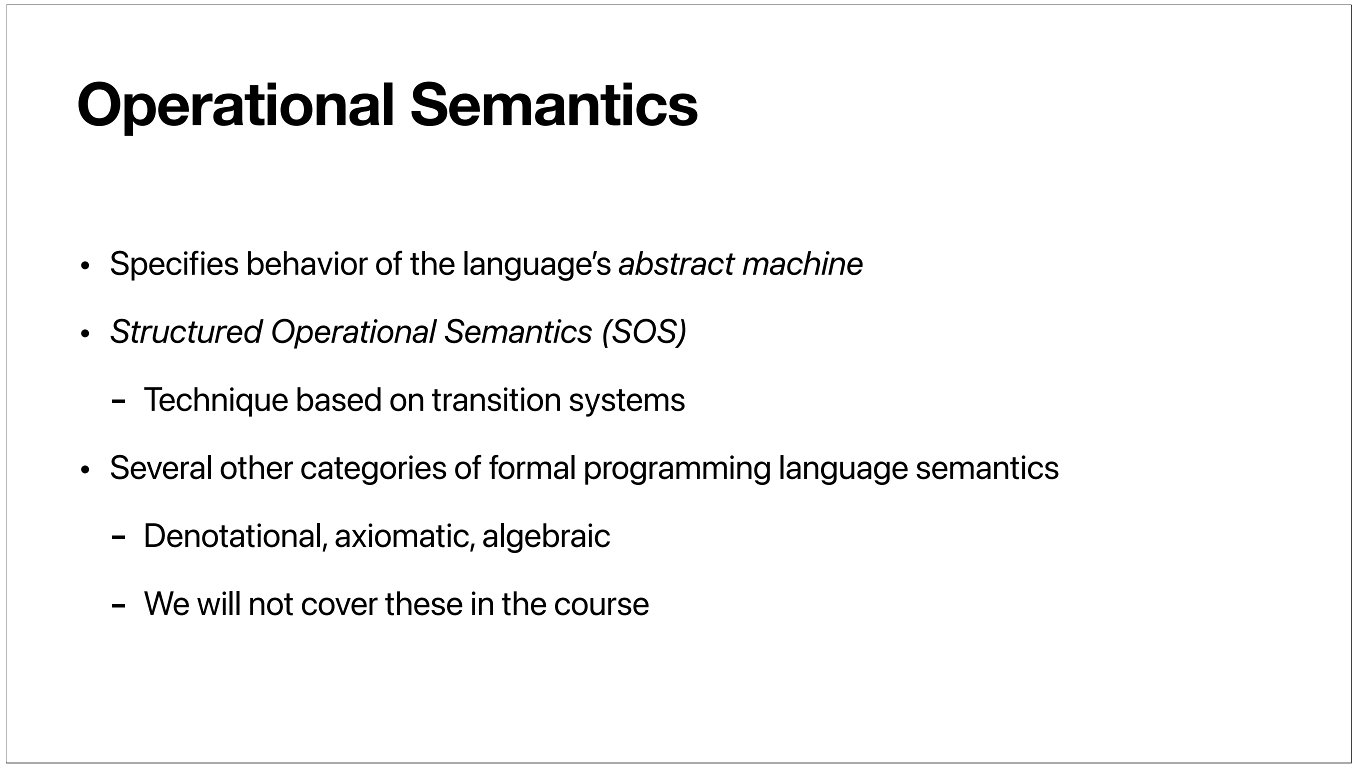
The print call is outside of the loop, does x have a valid binding?

Semantics in the Wild
JavaScript

- Language semantics can be surprising
- Putting thought into the semantics when designing can produce a friendlier language
- Understanding language semantics can help prevent errors when writing code



For 0 == “0”, JS coerces the string “0” to number 0
For 0 == [ ], JS converts an empty array to 0 when comparing to a number
For “0” == [ ], JS treats the “0” as true and the empty array as false when using the ‘==‘ operator

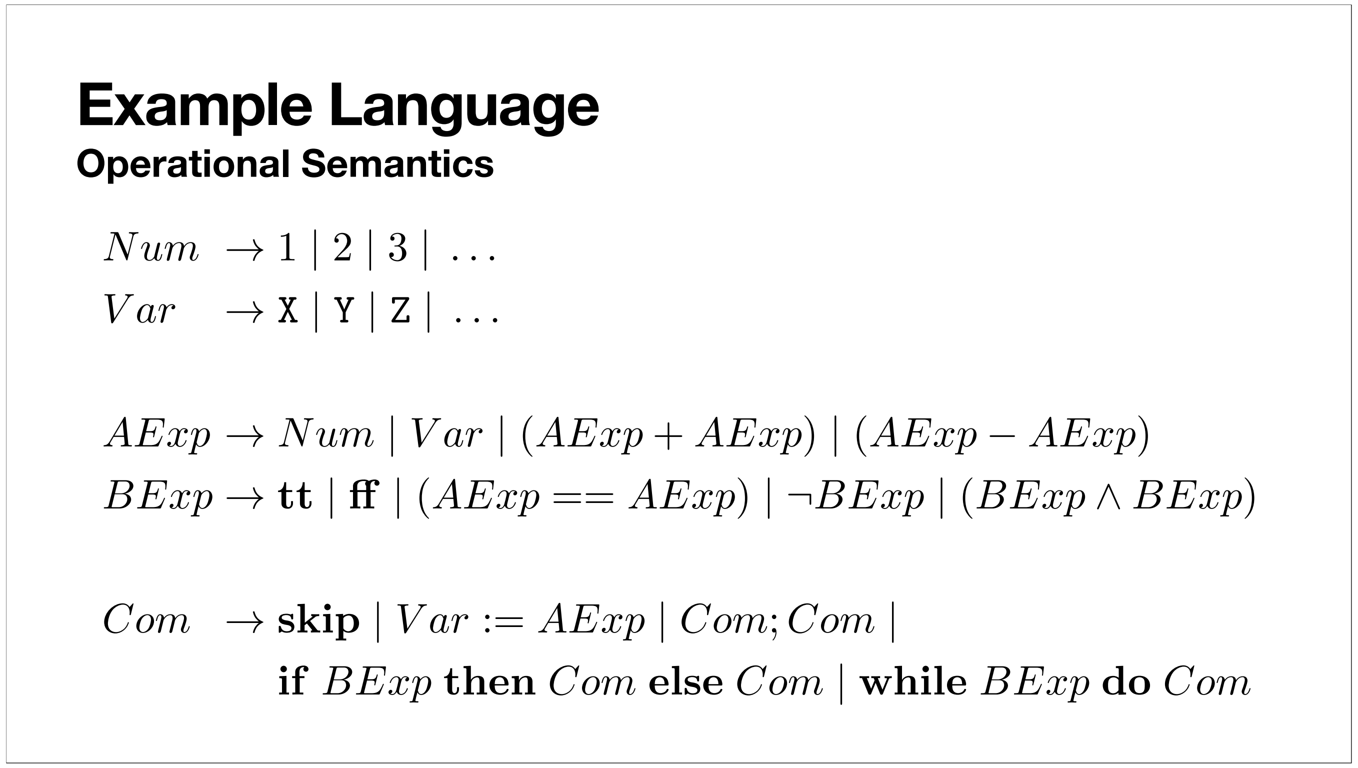


Operational Semantics

- Specifies behavior of the language's *abstract machine*
- *Structured Operational Semantics (SOS)*
  - Technique based on transition systems
- Several other categories of formal programming language semantics
  - Denotational, axiomatic, algebraic
  - We will not cover these in the course

Operational semantics is one way to formalize programming language semantics

Denotational semantics is the other major approach to semantics, but we will not cover it in this course.



Example Language

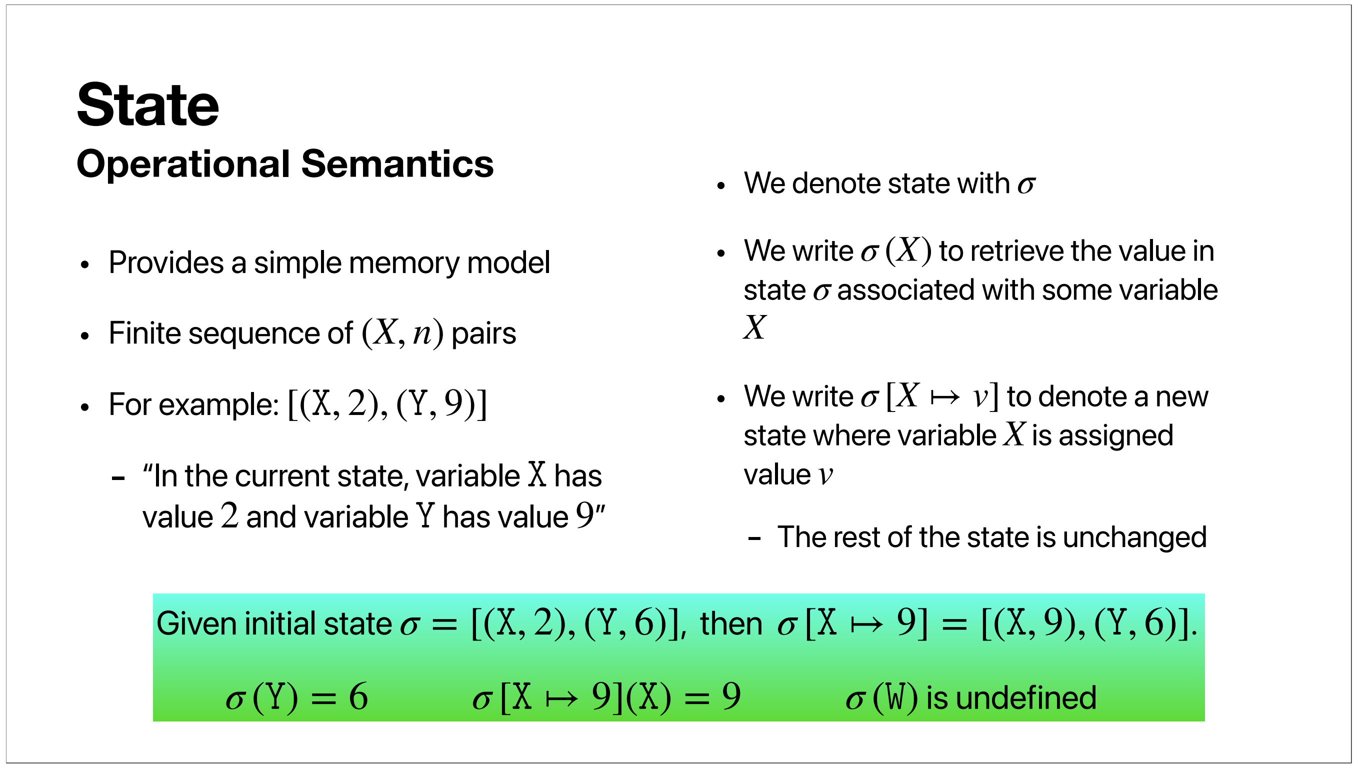
Operational Semantics

$$
\begin{aligned}
Num &\rightarrow 1 \mid 2 \mid 3 \mid \ldots \\
Var &\rightarrow \mathtt{X} \mid \mathtt{Y} \mid \mathtt{Z} \mid \ldots \\
\\
AExp &\rightarrow Num \mid Var \mid (AExp + AExp) \mid (AExp - AExp) \\
BExp &\rightarrow \mathbf{tt} \mid \mathbf{ff} \mid (AExp == AExp) \mid \neg BExp \mid (BExp \wedge BExp) \\
\\
Com &\rightarrow \mathbf{skip} \mid Var := AExp \mid Com; Com \mid \\
&\quad\ \mathbf{if}\ BExp\ \mathbf{then}\ Com\ \mathbf{else}\ Com \mid \mathbf{while}\ BExp\ \mathbf{do}\ Com
\end{aligned}
$$

The language supports arithmetic and Boolean expressions (AExp, BExp).

The terminal symbols tt and ff denote Boolean true and false.

Commands can be sequenced (Com; Com) and include support for variable assignment, conditional execution (if-statement) and looping (while-statement).



State

Operational Semantics

- Provides a simple memory model
- Finite sequence of (X, n) pairs
- For example: $[(\mathtt{X}, 2), (\mathtt{Y}, 9)]$
  - "In the current state, variable $\mathtt{X}$ has value 2 and variable $\mathtt{Y}$ has value 9"

- We denote state with σ
- We write $\sigma(X)$ to retrieve the value in state σ associated with some variable X
- We write $\sigma[X \mapsto v]$ to denote a new state where variable X is assigned value v
  - The rest of the state is unchanged

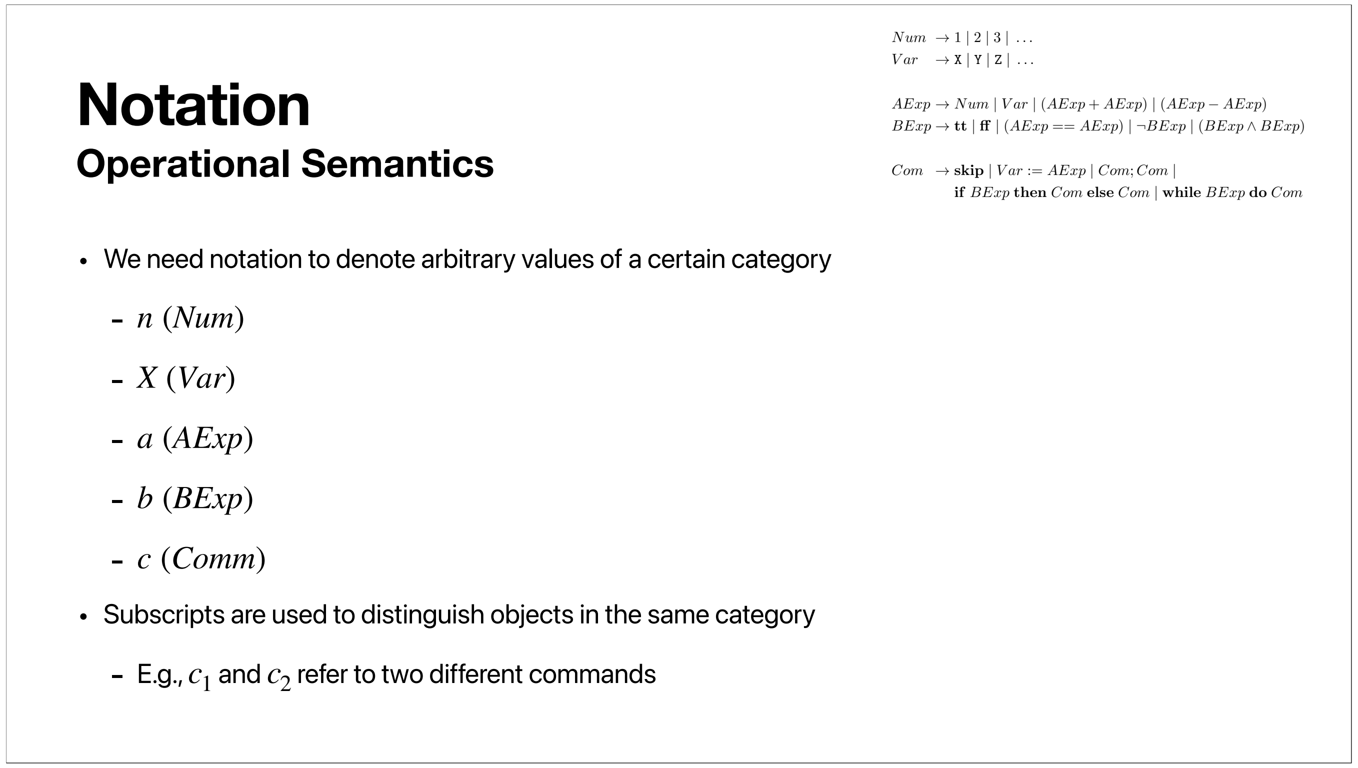
Given initial state $\sigma = [(\mathtt{X}, 2), (\mathtt{Y}, 6)]$, then $\sigma[\mathtt{X} \mapsto 9] = [(\mathtt{X}, 9), (\mathtt{Y}, 6)]$.

$\sigma(\mathtt{Y}) = 6$ $\quad$ $\sigma[\mathtt{X} \mapsto 9](\mathtt{X}) = 9$ $\quad$ $\sigma(\mathtt{W})$ is undefined

We can model memory using a finite sequence of variable and number pairs

State is denoted by sigma

The arrow which indicates the value associated with a variable can be read as "maps to" or "has value".



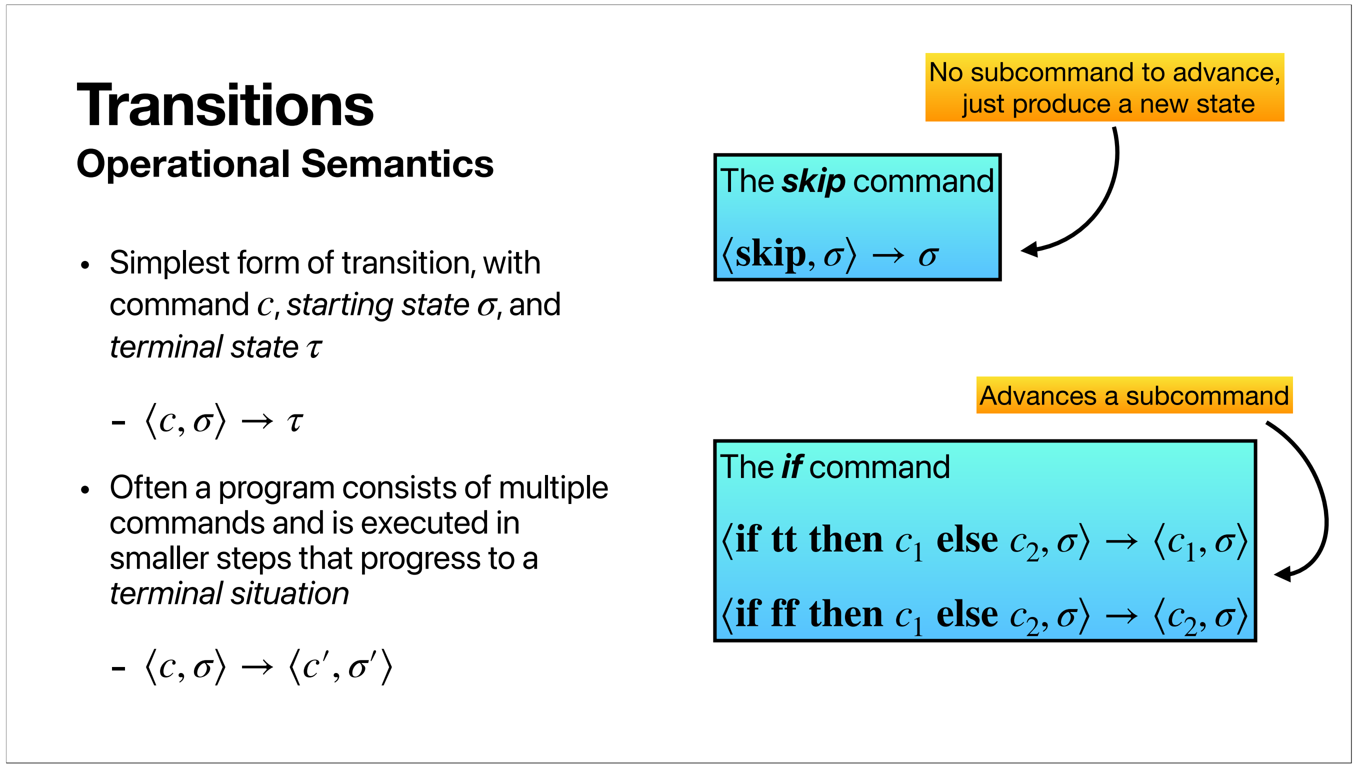
$$
\begin{aligned}
Num &\rightarrow 1 \mid 2 \mid 3 \mid \ldots \\
Var &\rightarrow \mathtt{X} \mid \mathtt{Y} \mid \mathtt{Z} \mid \ldots \\
\\
AExp &\rightarrow Num \mid Var \mid (AExp + AExp) \mid (AExp - AExp) \\
BExp &\rightarrow \mathbf{tt} \mid \mathbf{ff} \mid (AExp == AExp) \mid \neg BExp \mid (BExp \wedge BExp) \\
\\
Com &\rightarrow \mathbf{skip} \mid Var := AExp \mid Com; Com \mid \\
&\quad\; \mathbf{if}\ BExp\ \mathbf{then}\ Com\ \mathbf{else}\ Com \mid \mathbf{while}\ BExp\ \mathbf{do}\ Com
\end{aligned}
$$

Notation

Operational Semantics

- We need notation to denote arbitrary values of a certain category
  - n (Num)
  - X (Var)
  - a $(AExp)$
  - b $(BExp)$
  - c $(Comm)$
- Subscripts are used to distinguish objects in the same category
  - E.g., c_1 and c_2 refer to two different commands

In the case of variables,



Transitions

Operational Semantics

- Simplest form of transition, with command c, *starting state* σ, and *terminal state* τ
  - $\langle c, \sigma \rangle \to \tau$
- Often a program consists of multiple commands and is executed in smaller steps that progress to a *terminal situation*
  - $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$

No subcommand to advance, just produce a new state

The ***skip*** command

$\langle \mathbf{skip}, \sigma \rangle \to \sigma$

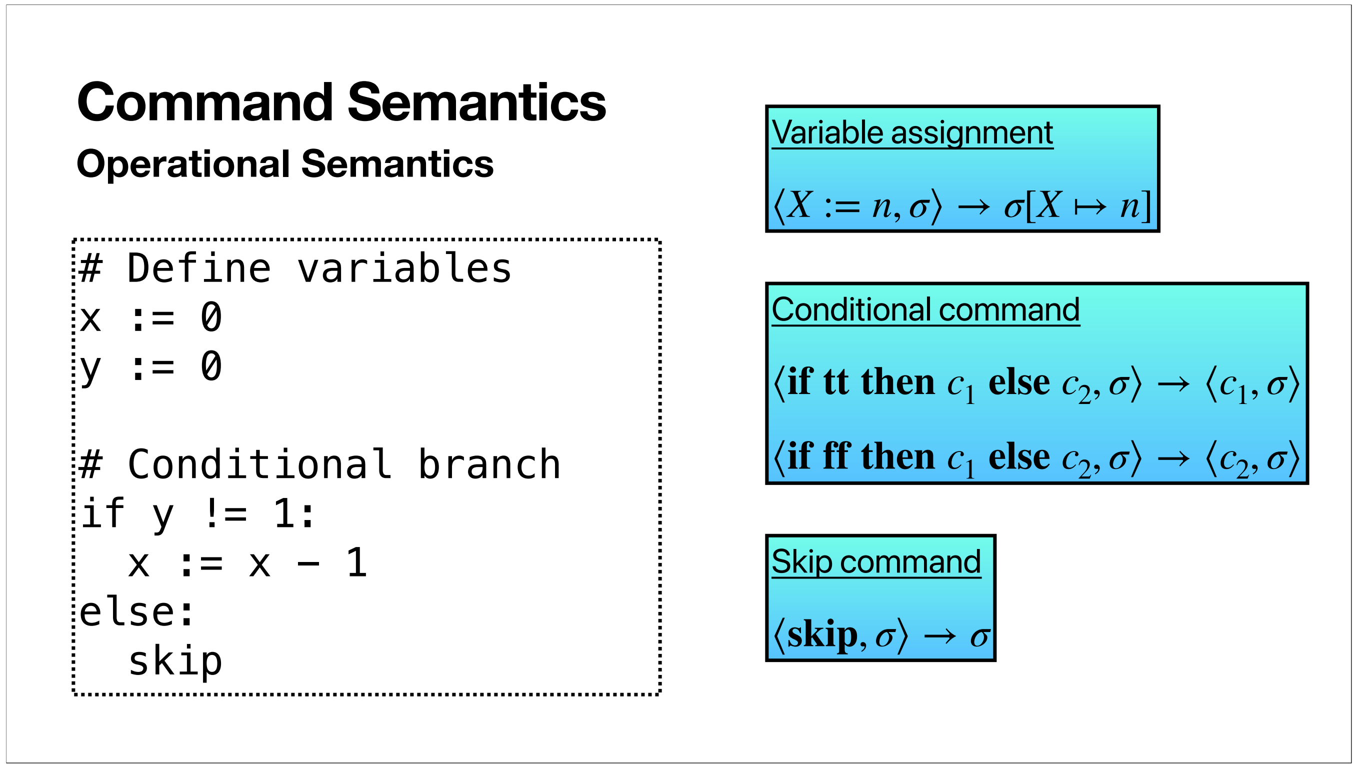
Advances a subcommand

The ***if*** command

$\langle \mathbf{if}\ \mathbf{tt}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to \langle c_1, \sigma \rangle$

$\langle \mathbf{if}\ \mathbf{ff}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to \langle c_2, \sigma \rangle$

We use c’ and sigma` to denote possibly different commands and state after executing the command c.



Command Semantics

Operational Semantics

```
# Define variables
x := 0
y := 0

# Conditional branch
if y != 1:
  x := x - 1
else:
  skip
```

Variable assignment

$\langle X := n, \sigma \rangle \to \sigma[X \mapsto n]$

Conditional command

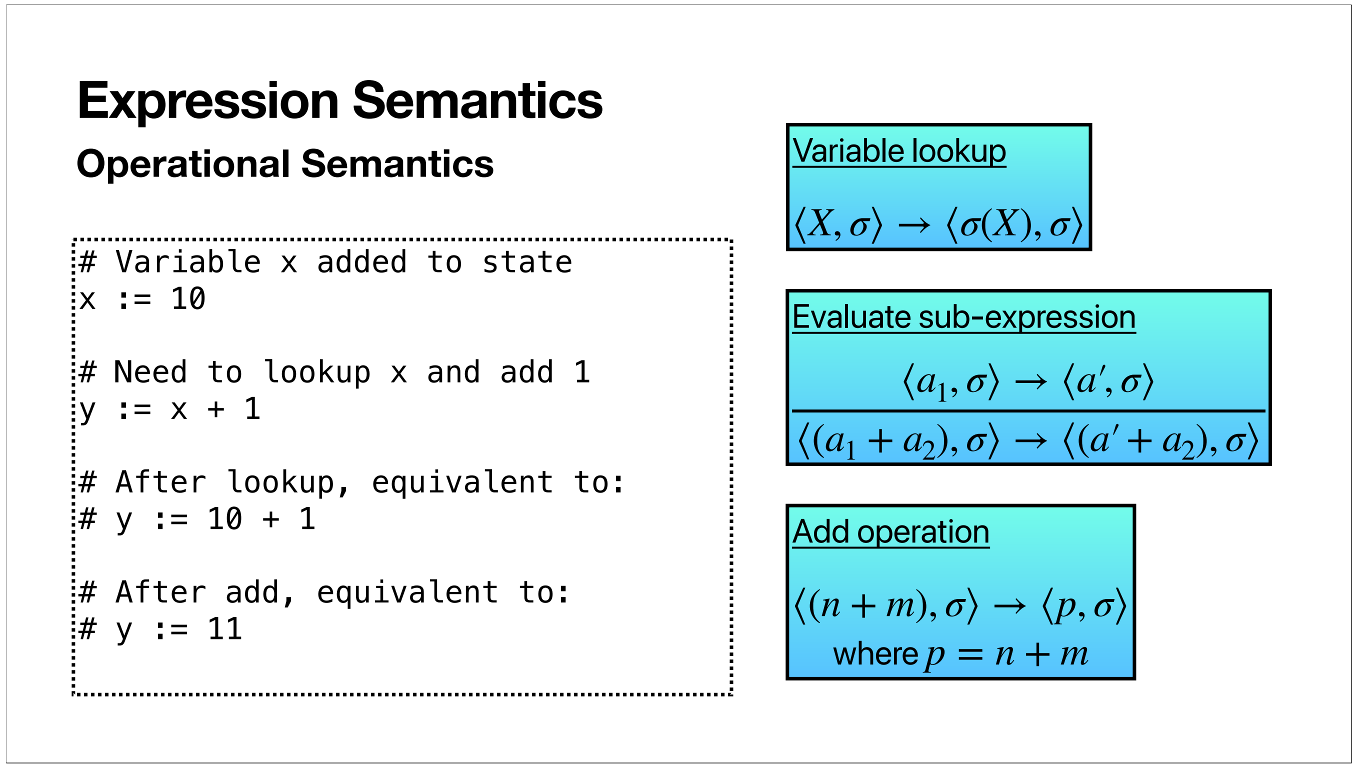
$\langle \textbf{if tt then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_1, \sigma \rangle$

$\langle \textbf{if ff then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_2, \sigma \rangle$

Skip command

$\langle \textbf{skip}, \sigma \rangle \to \sigma$

Commands allow us to modify machine state by either advancing to the next command or altering the machine state (sigma)



Expression Semantics

Operational Semantics

```
# Variable x added to state
x := 10

# Need to lookup x and add 1
y := x + 1

# After lookup, equivalent to:
# y := 10 + 1

# After add, equivalent to:
# y := 11
```

Variable lookup

$$\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle$$

Evaluate sub-expression

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a', \sigma \rangle}{\langle (a_1 + a_2), \sigma \rangle \rightarrow \langle (a' + a_2), \sigma \rangle}$$

Add operation

$$\langle (n + m), \sigma \rangle \rightarrow \langle p, \sigma \rangle$$

where $p = n + m$

For our example language, we have semantics for both arithmetic expressions and for boolean expressions.

Variable assignment is a command and those semantics will be reviewed shortly.

The three inference rules shown here are just a few of the rules defined for the language. There are more inference rules for arithmetic expressions as well as Boolean expressions.

Expressions do not modify state.

Computations

Operational Semantics

Example program, c

$\mathtt{X} := 0;\ \mathbf{if}\,\neg(\mathtt{X} = 0)\ \mathbf{then}\ \mathtt{Y} := 0\ \mathbf{else}\ \mathtt{Y} := 1$

- A *computation* is a sequence of transitions that cannot be extended
- A computation can be *terminating* (aka *finite*) or *divergent* (aka *infinite*)
- The computation of this program terminates

Computation for c

$\langle c, \sigma \rangle$

$\rightarrow_{Asg} \langle \mathbf{if}\,\neg(\mathtt{X} = 0)\ \mathbf{then}\ \mathtt{Y} := 0\ \mathbf{else}\ \mathtt{Y} := 1,\ \sigma[\mathtt{X} \mapsto 0] \rangle$

$\rightarrow_{Var} \langle \mathbf{if}\,\neg(0 = 0)\ \mathbf{then}\ \mathtt{Y} := 0\ \mathbf{else}\ \mathtt{Y} := 1,\ \sigma[\mathtt{X} \mapsto 0] \rangle$

$\rightarrow_{BEq} \langle \mathbf{if}\,\neg\mathbf{tt}\ \mathbf{then}\ \mathtt{Y} := 0\ \mathbf{else}\ \mathtt{Y} := 1,\ \sigma[\mathtt{X} \mapsto 0] \rangle$

$\rightarrow_{BNeg} \langle \mathbf{if}\ \mathbf{ff}\ \mathbf{then}\ \mathtt{Y} := 0\ \mathbf{else}\ \mathtt{Y} := 1,\ \sigma[\mathtt{X} \mapsto 0] \rangle$

$\rightarrow_{Else} \langle \mathtt{Y} := 1,\ \sigma[\mathtt{X} \mapsto 0] \rangle$

$\rightarrow_{Asg} \sigma[\mathtt{X} \mapsto 0,\ \mathtt{Y} \mapsto 1]$

Semantics

Rules

$$\langle \mathbf{skip}, \sigma\rangle \to \sigma \quad (c1)$$

$$\langle X := n, \sigma\rangle \to \sigma[X \leftarrow n] \quad (c2) \qquad \frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle X := a, \sigma\rangle \to \langle X := a', \sigma\rangle} \ (c3)$$

$$\frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_1; c_2, \sigma\rangle \to \langle c_2, \sigma'\rangle} \ (c4) \qquad \frac{\langle c_1, \sigma\rangle \to \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \to \langle c_1'; c_2, \sigma'\rangle} \ (c5)$$

$$\langle \mathbf{if\ tt\ then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to \langle c_1, \sigma\rangle \quad (c6)$$

$$\langle \mathbf{if\ ff\ then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to \langle c_2, \sigma\rangle \quad (c7)$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to \langle \mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle} \ (c8)$$

$$\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \langle \mathbf{if}\ b\ \mathbf{then}\ c;\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{else\ skip}, \sigma\rangle \quad (c9)$$

$$\langle X, \sigma\rangle \to \langle \sigma(X), \sigma\rangle \quad \text{(a1)}$$

$$\text{(a2)} \ \langle (n+m), \sigma\rangle \to \langle p, \sigma\rangle \ \text{where } p = n+m \qquad \langle (n-m), \sigma\rangle \to \langle p, \sigma\rangle \ \text{where } p = n-m \text{ e } n \geq m \ \text{(a3)}$$

$$\text{(a4)} \ \frac{\langle a_1, \sigma\rangle \to \langle a', \sigma\rangle}{\langle (a_1 + a_2), \sigma\rangle \to \langle (a' + a_2), \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \to \langle a'', \sigma\rangle}{\langle (a_1 + a_2), \sigma\rangle \to \langle (a_1 + a''), \sigma\rangle} \ \text{(a5)}$$

$$\text{(a6)} \ \frac{\langle a_1, \sigma\rangle \to \langle a', \sigma\rangle}{\langle (a_1 - a_2), \sigma\rangle \to \langle (a' - a_2), \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \to \langle a'', \sigma\rangle}{\langle (a_1 - a_2), \sigma\rangle \to \langle (a_1 - a''), \sigma\rangle} \ \text{(a7)}$$

$$\text{(b1)} \ \langle (n == m), \sigma\rangle \to \langle \mathbf{tt}, \sigma\rangle \ \text{if } n = m \qquad \langle (n == m), \sigma\rangle \to \langle \mathbf{ff}, \sigma\rangle \ \text{if } n \neq m \ \text{(b2)}$$

$$\langle (bv_1 \wedge bv_2), \sigma\rangle \to \langle bv, \sigma\rangle \ \text{where } bv \text{ is the } \textit{and} \text{ of } bv_1 \text{ and } bv_2 \quad \text{(b3)}$$

$$\text{(b4)} \ \langle \neg\mathbf{tt}, \sigma\rangle \to \langle \mathbf{ff}, \sigma\rangle \qquad \langle \neg\mathbf{ff}, \sigma\rangle \to \langle \mathbf{tt}, \sigma\rangle \ \text{(b5)}$$

$$\text{(b6)} \ \frac{\langle a_1, \sigma\rangle \to \langle a', \sigma\rangle}{\langle (a_1 == a_2), \sigma\rangle \to \langle (a' == a_2), \sigma\rangle} \qquad \frac{\langle a_2, \sigma\rangle \to \langle a'', \sigma\rangle}{\langle (a_1 == a_2), \sigma\rangle \to \langle (a_1 == a''), \sigma\rangle} \ \text{(b7)}$$

$$\text{(b8)} \ \frac{\langle b_1, \sigma\rangle \to \langle b', \sigma\rangle}{\langle (b_1 \wedge b_2), \sigma\rangle \to \langle (b' \wedge b_2), \sigma\rangle} \qquad \frac{\langle b_2, \sigma\rangle \to \langle b'', \sigma\rangle}{\langle (b_1 \wedge b_2), \sigma\rangle \to \langle (b_1 \wedge b''), \sigma\rangle} \ \text{(b9)}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \neg b, \sigma\rangle \to \langle \neg b', \sigma\rangle} \ \text{(b10)}$$

Computations

Example

Example program, c

$\mathtt{Y} := 1;\ \mathtt{X} := 0;\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)$

Computation for c

$\langle c, \sigma \rangle$

$\rightarrow_{c4,c2} \langle \mathtt{X} := 0;\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{Y} \mapsto 1] \rangle$

$\rightarrow_{c4,c2} \langle \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1] \rangle$

$\rightarrow_{c9} \langle \textbf{if}\ \neg(\mathtt{X} = \mathtt{Y})\ \textbf{then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1] \rangle$

$\rightarrow_{c5,a1} \langle \textbf{if}\ \neg(0 = \mathtt{Y})\ \textbf{then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1] \rangle$

$\rightarrow_{c5,a1} \langle \textbf{if}\ \neg(0 = 1)\ \textbf{then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1] \rangle$

$\rightarrow_{c8,b1} \langle \textbf{if}\ \neg\textbf{ff}\ \textbf{then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1] \rangle$

Computations
Example

Example program, c

$\mathtt{Y} := 1;\ \mathtt{X} := 0;\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)$

Computation for c

$\rightarrow_{c8,b5} \langle \textbf{if tt then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c6} \langle \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c5,a1} \langle \mathtt{X} := (0 + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c5,a2} \langle \mathtt{X} := 1;\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{X} \mapsto 0, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c4,c2} \langle \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1), \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c9} \langle \textbf{if}\ \neg(\mathtt{X} = \mathtt{Y})\ \textbf{then}\ \mathtt{X} := (\mathtt{X} + 1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)\ \textbf{else skip}, \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

Computations

Example

Example program, c

$\mathtt{Y} := 1;\ \mathtt{X} := 0;\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X} + 1)$

Computation for c

$\rightarrow_{c5,a1} \langle \textbf{if}\ \neg(1 = \mathtt{Y})\ \textbf{then}\ \mathtt{X} := (\mathtt{X}+1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X}+1)\ \textbf{else skip},\ \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c5,a1} \langle \textbf{if}\ \neg(1 = 1)\ \textbf{then}\ \mathtt{X} := (\mathtt{X}+1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X}+1)\ \textbf{else skip},\ \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c8,b1} \langle \textbf{if}\ \neg\textbf{tt}\ \textbf{then}\ \mathtt{X} := (\mathtt{X}+1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X}+1)\ \textbf{else skip},\ \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c8,b4} \langle \textbf{if}\ \textbf{ff}\ \textbf{then}\ \mathtt{X} := (\mathtt{X}+1);\ \textbf{while}\,\neg(\mathtt{X} = \mathtt{Y})\ \textbf{do}\ \mathtt{X} := (\mathtt{X}+1)\ \textbf{else skip},\ \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c7} \langle \textbf{skip},\ \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]\rangle$

$\rightarrow_{c1} \sigma[\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 1]$