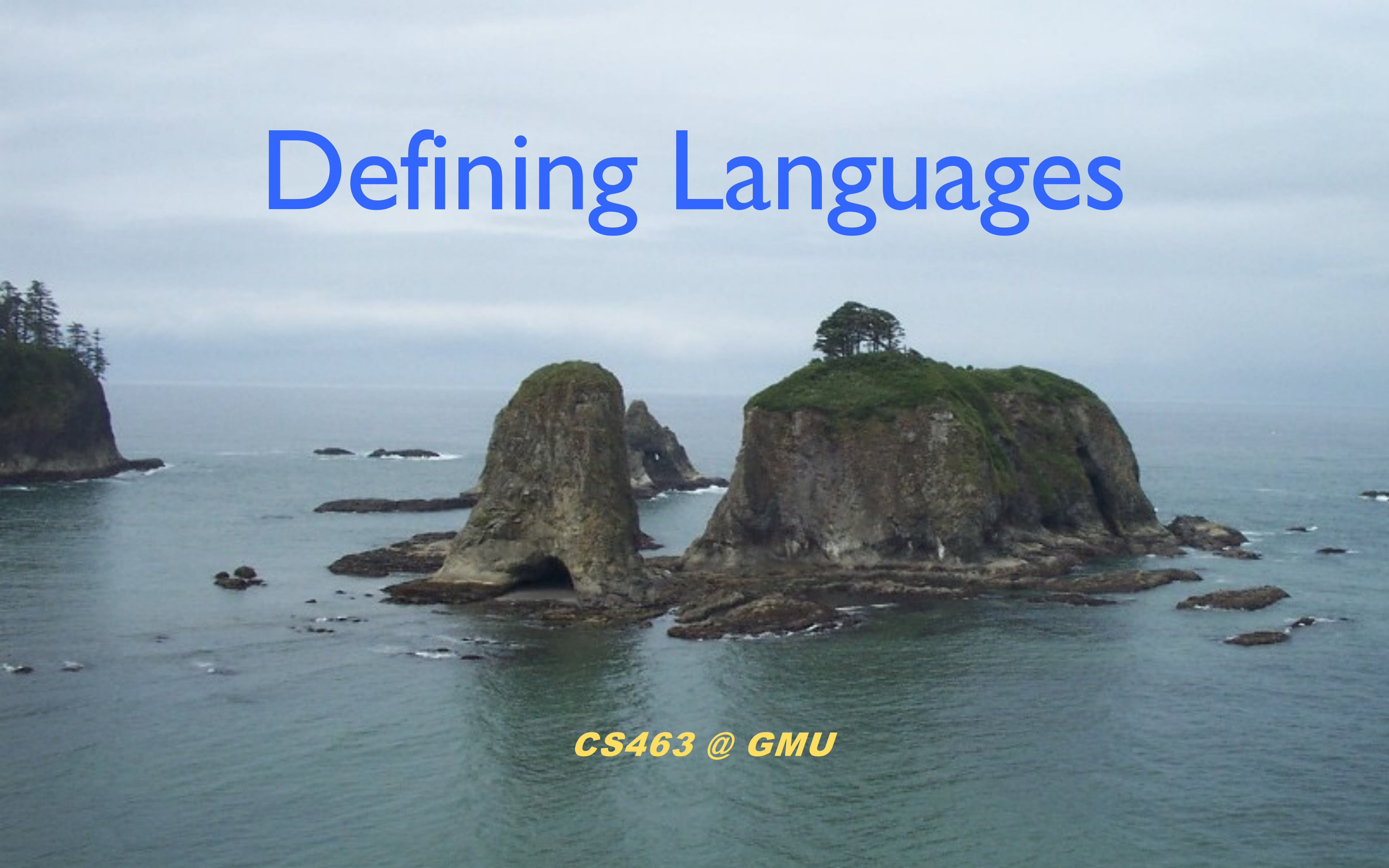
Defining Languages



CS463 @ GMU

How do we discuss languages?

We might focus on these qualities:

- **readability**: how well does a language explicitly and clearly describe its purpose?
- **writability**: how expressive is the language? how convenient is it for a programmer to write new code or edit existing code?
- **reliability**: how well does the language predictably enforce certain rules or patterns of usage?

Goals

Let's put that CS 330 prerequisite to use!

- Connect the grammar and language definitions to **programming** languages

- Syntax vs Semantics:
  - Separate the representation (syntax)
  - from the meaning (semantics)

- Realize how much we can control with BNFs, and what we cannot

Syntax

What is a language?

- **a language is just a set of sentences.**
  - a sentence is a (finite) sequence of symbols.
  - each (finite) sentence either is, or is not, in some language

- **formal rules determine the set of sentences in a language**
  - If you can generate a sentence with the given production rules, you "recognize" that it is in the language.
  - **production rules** map **non-terminals** to some mixture of **terminals** and non-terminals.
  - From a "start symbol" to all terminals, these rules generate *all* the sentences.
  - Example:
    - **S** → Num '**+**' S | Num
    - **Num** → Digit | Digit Num
    - **Digit** → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

  - non-terminals: **S**, **Num**, **Digit**
  - Start symbol: **S**
  - terminals: **+**, **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**

Chomsky Hierarchy of Languages

- **Type 3**: Regular Languages
- **Type 2**: Context-Free Languages
- **Type 1**: Context-Sensitive Languages
- **Type 0**: Recursively Enumerable Languages

- (we will only focus on types 3 and 2)
- Limits on what can be on the lefthand side (LHS) or righthand side (RHS) of production rules can differentiate them

Regular Languages (type 3)

These are the languages that regular expressions can describe.

- examples of regular expressions: **a* b+ c? (d | e | f)**
- **Notes on <u>regular expressions:</u>**
  - a terminal is also a (very simple!) regular expression (a "regex" for short).
  - the empty string is a regex, represented as ε.
  - **concatenation**: **AB** means a string from A followed by a string from B.
  - **repetition (Kleene Star)**: **A*** indicates zero or more strings from A.
    - A^+ indicates one or more strings from A.
  - **selection**: **A | B** indicates either a string from A, or a string from B.
  - **grouping**: **(A)** uses parentheses to facilitate the other

Regular Languages (type 3)

- Recognizable by a DFA.
- **Production rules:**
  - LHS: only a non-terminal.
  - RHS: terminals, and at most one non-terminal, at edge.

examples

$S \longrightarrow tN$

$S \longrightarrow Nt$

- **Production rules for this regex:** **$a^+ b? c^*$**
  - **$S \rightarrow aS \mid aT$**
  - **$T \rightarrow bU \mid U$**
  - **$U \rightarrow \varepsilon \mid cU$**

Practice: produce these strings:
- aabc
- ac
- a

- **style:** no "information" can transmit from different places, such as "how many a's were there? let's have those many b's over here."

Context-free Languages (type 2)

- **Most programming languages are context-free.**
- These are the languages recognizable by a pushdown automaton (PDA).
- Can be described by BNFs (Backus-Naur Form)
- Production rules:
  - LHS: a single non-terminal
  - RHS: any mixture of terminals and non-terminals

- Example representable languages:
  - $\{a^n b^n \mid n>0\}$ (some number of a's, followed by same number of b's)
  - $S \rightarrow SS \mid (S) \mid \varepsilon$ (balanced parentheses)
- A basic mathematical expressions example:

$$
\begin{array}{lll}
\text{Expr} & \rightarrow & \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term} \\
\text{Term} & \rightarrow & \text{Var} \mid \text{Num} \\
\text{Var} & \rightarrow & a \mid b \mid c \mid d \\
\text{Num} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\end{array}
$$

Context-sensitive Languages (type 1)

- **Recognized by a linear bounded automaton**
  ("a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input")
- **Rules of shape $\alpha A\beta \rightarrow \alpha\gamma\beta$, where A is a non-terminal, α,β are zero or more terminals and non-terminals, and γ is one or more terminals and non-terminals.** *(what does this mean?)*
  - LHS: at least one non-terminal.
  - RHS: at least one terminal or non-terminal.

Recursively Enumerable Languages (type 0)

- **set of all languages that a Turing machine can recognize.**
- **contains all other types of languages (3, 2, 1)**
- **no restriction on production rules – any amount of terminals and non-terminals on both sides.**

Example Grammar (type 2)

Program → Stmts

Stmts → Stmt | Stmt ; Stmts

Stmt → Var = Expr | print Expr

Var → a | b | c | d

Expr → Expr + Term | Expr – Term | Term

Term → Var | Const

Const → Digit | Digit Const

Digit → 0 | 1 | 2 | … | 9

Examples (see solutions):

- c = 1
- a = 3 ; b = a+1; print b

Example Derivation

Program	***must begin with start symbol***
⇒ **Stmts**	***only apply one rule at a time!***
⇒ **Stmt**	
⇒ **Var = Expr**	***… each line is a sentential form …***
⇒ **a = Expr**	
⇒ **a = Expr + Term**	
⇒ **a = Term + Term**	
⇒ **a = Var + Term**	
⇒ **a = b + Term**	
⇒ **a = b + Const**	
⇒ **a = b + Digit**	
⇒ **a = b + 5**	**end with no more non-terminals: a sentence**

Derivations

Sentential Form: combination of terminals and non-terminals. Each step of a derivation must be a sentential form.

Leftmost derivation: always replacing the leftmost non-terminal next. Derivations can also be rightmost, or neither.

→ previous example was leftmost
→ rightmost derivations exist too

Practice Problems (Production Rules)

- Create production rules that recognize the language described by $\{a^n b^n \mid n \geq 1\}$

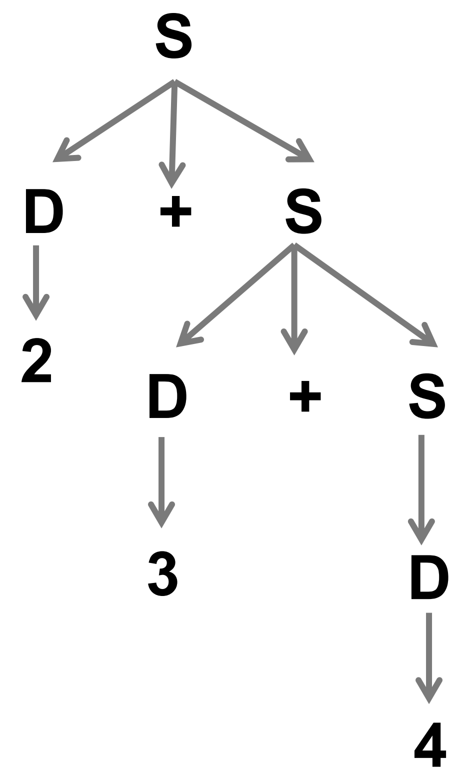
- Create production rules that recognize lists of single digits, e.g. [2,4,6,8]

Parsing Sentence Structures

Parse Trees

- **leaves**: terminals
- **nodes**: non-terminals

- Records which production rules were applied.
  - *(but not in what order!)*
- extracts the meaningful structure out of the original source text.



Example Parse Tree

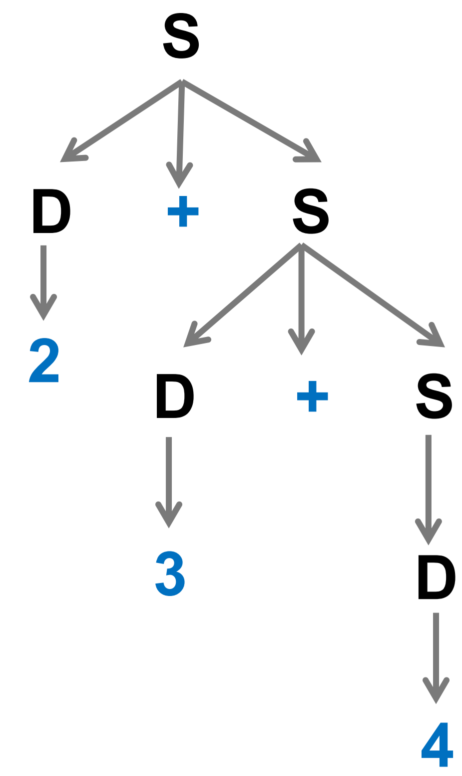
Production Rules:

S → D+S | D

D → 0 | 1 | 2 | … | 9

Derivation:

S
→ D + S
→ 2 + S
→ 2 + D + S
→ 2 + 3 + S
→ 2 + 3 + D
→ 2 + 3 + 4



Associativity

Idea: does **x-y-z** behave like **((x-y)-z)**, or like **(x-(y-z))** ?

Impact: does the left or right operator grab its arguments first?

Left-associative addition:
rule repeats LHS's non-terminal on the left.

Expr → **Expr** + Num | Num

Num → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Right-associative addition:
repeats LHS's non-terminal on the right.

Expr → Num + **Expr** | Num

Num → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Usually, all math operators are left-associative except exponentiation.

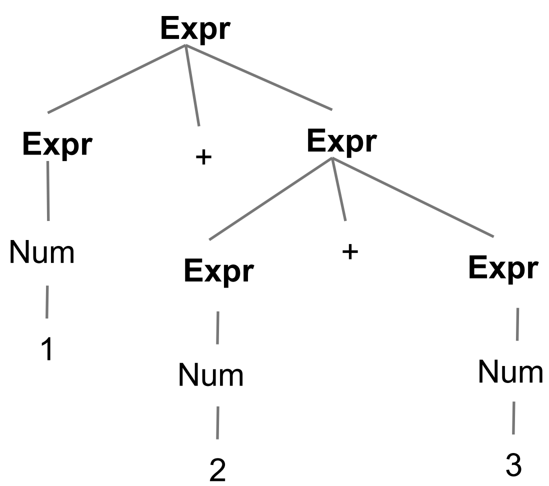
Try drawing parse trees for 1+2+3 in each of the above grammars.

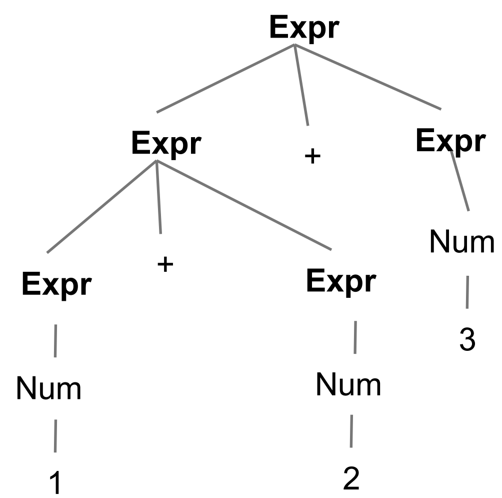
Associativity

Ambiguous: The following rules don't enforce left- or right-associativity.

Expr → **Expr** + **Expr** | Num

Num → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9





Operator Precedence

Idea: different operators are more aggressive or less aggressive in acquiring their arguments. (it's a generalized PEMDAS/BEDMAS).
Impact: operators further from the start symbol are higher precedence.

No Precedence (same level): Expr → Expr + Num | Expr * Num | Num
Num → 0 | 1 | 2 | 3 | … | 9

Introducing Precedence (split out to separate non-terminals/rules):

Expr → Expr + Term | Term
Term → Term * Num | Num
Num → 0 | 1 | 2 | 3 | ... | 9

- Here, multiplication binds more tightly than addition, by design.
- more levels of precedence possible via more nestings of non-terminals.
- *Because we have Expr+Term (and not Expr+Expr), addition happens to be left-associative. But it could have been right-associative or ambiguous.*

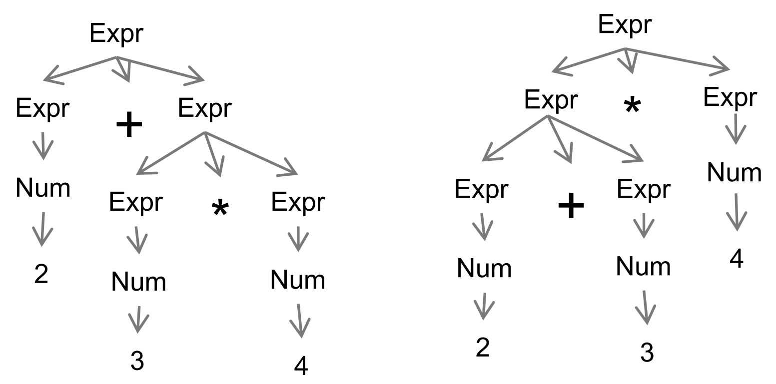
Ambiguity

There may be multiple valid parse trees for a single sentence in a language. This is **ambiguity**, and we can't have it in our programming languages.

Rules:
Expr→ Expr + Expr | Expr * Expr | Num
Op → * | +
Num→ 0 | 1 | 2 | … | 9

Two Valid Derivations

Expr	Expr
→ Expr+Expr	→ Expr*Expr
→ Num+Expr	→ Expr+Expr*Expr
→ 2+Expr	→ Num+Expr*Expr
→ 2+Expr*Expr	→ 2+Expr*Expr
→ 2+Num*Expr	→ 2+Num*Expr
→ 2+3*Expr	→ 2+3*Expr
→ 2+3*Num	→ 2+3*Num
→ 2+3*4	→ 2+3*4



Ambiguity

Removing Ambiguity: add more non-terminals to introduce precedence or associativity, or somehow remove all but one possible parse tree for any sentence that had more than one.

Ambiguous Example: ***(associativity and precedence issues!)***

Expr→ Expr + Expr | Expr * Expr | Num
Num→ 0 | 1 | 2 | … | 9

Disambiguated Version: ***(introduced operator precedence <u>and</u> left-assoc.)***

Expr → Expr + Term | Term
Term → Term * Num | Num
Num → 0 | 1 | 2 | ... | 9

Practice Problems - Ambiguity

- Consider a language with if-statements and if-else statements. Create a string showing the ambiguity.

  S ⟶ if S then S | if S then S else S | true | false | print

- Fix the grammar above so that is not ambiguous.

This is known as the "dangling else" problem.

Semantics

Semantics (overview)

Static Semantics: restrictions on strings in a language beyond basic syntax rules.

- declaring variables before usage
- numeric literals being assigned to wide enough types
- many type constraints are representable as static semantics
- Attribute Grammars help decorate a parse tree with information

Dynamic Semantics: represent the meaning of expressions, statements, and the execution of programs. Three main approaches are:

- operational semantics (results of running on specific machine)
- denotational semantics (recursive function theory)
- axiomatic semantics (pre- and post-conditions)

Attribute Grammars

attribute values: we **decorate a parse tree** with more information

- Example: what is the type of each expression in a program?

Attribute values: info about a node in a parse tree (about a non/terminal)

Semantic functions: how we generate attribute values, *per production rule*

Attribute predicates: constraints on attributes as they relate to each other.

- These encode the static semantics of the language.
  (whole-program constraints, using attribute values)
- must "pass" these predicate tests, or we reject the sentence.

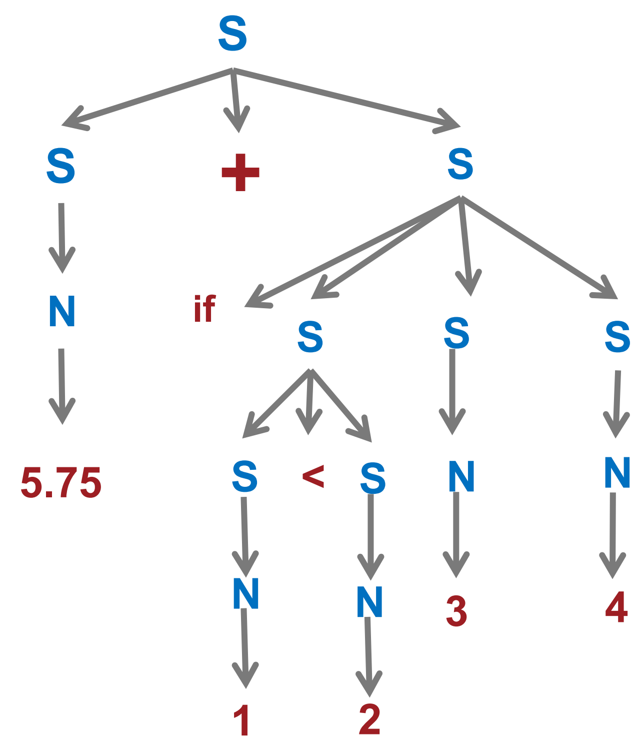
Example

S ⟶ if S S S | S+S | S<S | (S) | N
N ⟶ 1 | 2 | 3 | 4 | 5.75

Target sentence:

5.75 + if 1<2 3 4

Question: What is the actual type at each node? ($\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$)



More Examples

S ⟶ S + S | S?S:S | true | false | C | T | N
C ⟶ 'h' | 'j'
T ⟶ "ello"
N ⟶ 1 | 2 | 3

Think about having these types in a Java-like language: int, char, string, bool

Target Sentences:

- **2+3**
- **2+"ello"**
- **(true? 'h' : 2) + "ello"**

how would Java vs Python handle this? (assuming we wrote corresponding syntax)

Side note: fun with type inference

Getting more context from code is necessary for it to be compiled.

```
class Main {
  public static void main(String[] args) {
    System.out.println((2 < 3 ? 80 : 'c'));
    System.out.println((2 < 3 ? 80 : 'c') + 100);
    System.out.println((2 < 3 ? 80 : 'c') + "" + 100);
  }
}
```

- This code prints "P", "180", and "P100".
- Java had to choose a type for the if-expression (char); it wasn't stored in an explicitly typed variable.
- println doesn't get to affect the type of its argument

Attribute Values

What are they?

- pieces of information that can decorate nodes in a parse tree.
- example: the **actual_type** of `true` is `Boolean`. (see example in later slides)

Where do they come from?

- **Semantic functions**, defined per production rule, generate attribute values of that rule's output.
- example: <u>S⟶S+S</u> , where adding two ints creates an int.
  - the add node's actual_type is based on the types of its two operands.
- example: <u>S⟶S?S:S</u>
  - an if-expr's actual_type is shared by both branches

Semantic Functions

Production Rules each have semantic functions associated with them to generate attribute values.

- Some are **synthetic**: building info up from leaves to the root node of the parse tree (looking at sub-nodes).
  - addition of two nodes with integer attributes is thus an integer
- Some are **inherited:** deciding attributes from root to leaves (looking at parent-nodes).
  - an expression that happens to be the guard statement of an if-statement is expected to be a Boolean expression
- Some are **intrinsic**: the attribute value is determined by the node itself without looking at any parent/child nodes
  - the actual_type of **true** is Boolean, without further needed context.

Attribute Predicates

- Constraints that compare attributes will further restrict the language. These semantic functions are called **attribute predicates**.

- These use the attribute values (calculated through semantic functions) to record the static semantics of the language.

Attribute Grammar Example

Given this grammar:

```
Assign → Var = Expr
Expr   → Var + Var | Num | Var
Num    → 0 | 1 | ... | 9
Var    → a | b | c
```

Attribute Values:

- **actual_type**: (will be synthesized)
- **expected_type**: (will be inherited)

Semantic Functions:

- actual()
- expected()

See implementations on next slides

Attribute Grammar Example

Targeted Production Rule: $Expr \rightarrow Var_1 + Var_2$

Semantic function implementation bits:

- expected(Expr) = *<inherited from parent>*
- actual(Expr) = actual(Var_1) *# either of Var_1 or Var_2 is OK here.*

Attribute Predicates:

- constraint: actual(Var_1)==actual(Var_2)
- constraint: actual(Expr)==expected(Expr)

Attribute Grammar Example

Targeted Production Rule: Var → *id*

Attribute Predicates:

- actual(*id*) = lookup_type(*id*)

lookup_type is something like a dictionary of all in-scope variables and their types.

Attribute Value Computation

How are attribute values computed?

- If all attributes were inherited:
  $\longrightarrow$ the tree could be decorated in top-down order.
- If all attributes were synthesized:
  $\longrightarrow$ the tree could be decorated in bottom-up order.

- In many cases, both kinds are used, and some combination of top-down and bottom-up must be used.
- **Example: typechecking tends to use both.**

Similarities to Typing Rules

$\longrightarrow$ evaluation

$::$ typecheck

We will discuss typing rules later on, and we will see rules such as these:

$$\frac{e_1::\text{int} \qquad e_2::\text{int}}{e_1+e_2 \;::\; T} \qquad\qquad \frac{}{\text{true} :: \text{boolean}}$$

$$\frac{e_1::\text{boolean} \qquad e_2::T \qquad e_3::T}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \;::\; T}$$

What similarities do you see to evaluation rules? Are there any synthetic, inherited, or intrinsic attributes?

Dynamic Semantics

How do we represent meaning?

- what does a language ***mean***?
- little agreement on "best" way to represent semantics
- Some needs for a formal semantics:
  - programmers need to know what statements mean
  - compiler writers must know exactly what language constructs do to implement them
  - correctness proofs are possible (but difficult)
  - compiler generators are possible
  - designers could detect ambiguities and inconsistencies

Three Approaches to Dynamic Semantics

Operational Semantics

define by running on a simpler machine

Denotational Semantics

map each non-terminal to a value

Axiomatic Semantics

axioms / inference rules per production rule

Operational Semantics (summary)

Monkey See, Monkey Do – We define meaning by showing the results of running each code structure on a specific machine.

- The machine could be a VM or some idealized (simplified) machine.
- change in state of the machine defines each statement's meaning.

Operational Semantics

We define meaning by showing how some structure runs on a particular machine (simulated or actual).

- change in state of the entire machine (registers, memory, etc.) defines the structure's meaning.
- there's too much detail on a real machine:
  we choose idealized virtual machines.
- great for informal descriptions: "it works like this simpler thing."

Reading further? There are two main variations:

- Natural Operational Semantics ("Big Step"): focuses on the final result
- Structural Operational Semantics ("Small Step"):
  focuses on the sequence of state transitions

Operational Semantics

The definition becomes very machine-dependent ☹

- translate source code to an idealized computer's machine code
  - what if this translation is formally written down? When the operational semantics gets too complicated, it becomes useless.
- best usage: informal definitions serve programmers and implementers.

Operational Semantics

- Horror Story: the semantics of **PL/1** *(est. 1976)* were formally defined via operational semantics with the Vienna Definition Language(VDL), but it was too complex to be useful.

- Operational Semantics defines meaning in terms of a lower-level programming language. Leads to circular definitions; **keeps pushing the true meaning to another level**. We're not going to focus too much on operational semantics.

Denotational Semantics (overview)

Recursive Function Theory –
we map each structure to a mathematical object (a value).

- Examples: eval_bool, eval_expr, eval_stmt, etc.

- these mapping functions of one structure might in turn use other mapping functions, all the way down.
- usefulness: a carefully specified definition is executable! ☺
  - *See our haskell implementations of the lambda calculus.*

Denotational Semantics

- Based on **recursive function theory**, different semantics are encoded per production rule. This works very well with the recursive nature of a BNF's definition.
- abstract (not tied to specific machine characteristics)
- Originally from Scott and Strachey (1970)
- Tied more directly to mathematics, it can express correctness of programs; it can be used in compiler generation systems.
  - It's not as useful for language users though.

Denotational Semantics: Approach

Quick definition:

- The state of a program is all its current variables and their values.
- To track the current state: we could record a list or table of names-to-values.

  [a:5, b:12, c:[1,2,3], other:True, …]

- **Helper functionality**: let `lookup` be a function that accepts a Name to look up, a current State, and returns the named variable's current Value.

  lookup :: Name → State → Value

Expressions (example: evalE)

Define: expression evaluation. **Value space**: IntegersU{error}

evalE :: Expr ⟶ State ⟶ Int

```
evalE(Num n,     state) = n

evalE(Var name, state) = lookup(name, state)

evalE(BinOp expr1 '+' expr2,state) =
    let    v1 = evalE(expr1,state)
           v2 = evalE(expr2,state)
    in if (v1==error or v2==error):
       then error
       else v1+v2

evalE(BinOp expr1 '*' expr2,state) =  …
```

Denotational Semantics: Approach

- value space : choose a mathematical object(type) for each language entity.
  - Perhaps integers, bools, strings, state-of-memory, or combinations of such things.
  - example: valuespace = ints ∪ reals ∪ bools ∪ ⊥
- Define functions mapping from each entity to the chosen objects
  - E.g. mapping each term to an integer
- Include the error value, so failing computations have a value representation. This value is called Bottom; it's considered an element of every single type. Often shown as ⊥.
- Loops are converted to recursion.
  - As statements, the value space is probably the new program state.

Denotational Semantics: Basic Idea

To define the semantics of some calculation, such as type checking/evaluation:

- group up the language's structures: related expressions, statements, or even more fine-grained, such as digits, identifiers, boolean expressions, etc.
- choose a value space: usually the type of results your calculation yields. Perhaps integers, other types, whatever the calculation should yield. This can be a combination of things, such as **Ints ∪ Doubles ∪ Booleans ∪ …**
- define functions that can all call upon each other, each defining the meaning (e.g., typeof) for each of those groups.
    - e.g., evalBool, evalExpr, evalStmt
- taken together, these functions define the semantics of that calculation.

Expressions (same example, evalE)

Define: expression evaluation. **Value space**: Integers∪{error}

evalE :: Expr ⟶ State ⟶ Int

```
evalE(Num n,     state) = n

evalE(Var name, state) = lookup(name, state)

evalE(BinOp expr1 '+' expr2,state) =
    let    v1 = evalE(expr1,state)
           v2 = evalE(expr2,state)
    in if (v1==error or v2==error):
       then error
       else v1+v2

evalE(BinOp expr1 '*' expr2,state) =  …
```

Assignment Statements (evalS)

Define: assignment. **Value space**: state sets U{error}

```
evalS :: Stmt ⟶ State ⟶ State
evalS(x := expr, s) = let  v = evalE(expr,s)
                      in if v==error:
                         then error
                         else update(s, (x,v))

# however you want to reset name's value in s
update :: State ⟶ (Name, Val) ⟶ State
update(s, (name, val)) =   ( s[name] == val )
```

Loops (evalS)

```
evalS :: Stmt ⟶ State ⟶ State
evalS(while b do L, s) =
  let bv = evalB(b, s)
  in if (bv==false)
     then s
     else if (bv==true)
          then let s2 = evalS(L, s)
               in evalS(while b do L, s2)
          else error
```

Loop Meaning

- the loop's meaning is the resulting state: variables and their values.

- A loop is converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state-mapping functions.

- recursion is easier to describe with mathematical rigor than iteration

Denotational Semantics: Example

See DenExpr.hs for an example.

- defines digits, expressions, statements
- provides functions **evalE** and **evalS**.
- sneak peek at some Haskell code – just focus on general ideas of denotational semantics, and the specifics of Haskell code will be our focus later.
  - I think it reads better than the book's notation...

Denotational Semantics: Summary

- can be used to prove correctness of programs
- provides a rigorous way to think about programs
- can aid language design
- has been used in compiler generation systems
- not very useful to language users (programmers)

Axiomatic Semantics (overview)

Proof by Conditions – allows us to **prove some claim** by carefully showing the implications of each individual statement in the program.

- define axioms and inference rules for each production rule
- pre- and post-conditions help expose meaning of statement
- start with the post-condition of the entire program, and work backwards, seeking the weakest pre-condition necessary. Loops are difficult!

Axiomatic Semantics

- Pre-condition: an assertion (about variables' relations) that is true just before a statement
- Post-condition: an assertion (about variables' relations) that is true following a statement
- Weakest Pre-condition: the least restrictive pre-condition that will guarantee the post-condition.

Form

- **pre/post form:** **{P} statement {Q}**

let's find a precondition.

- example: **__?__ a=b+1 {a>1}**

  - possible precondition: **{b>10}**
  - weakest precondition: **{b> 0}**

Axiomatic Semantics: Assignment

- **axiom for assignment statements:**

  (x=E): *"same Q, except that now variable X maps to E."*

  $$\{Q\}\quad x = E \quad \{Q_{X\rightarrow E}\}$$

- **The Rule of Consequence:** *"update pre/post conditions."*

$$\frac{\{P\}\ S\ \{Q\},\quad P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Axiomatic Semantics: Sequences

Inference rule for sequences of the form: $s_1;\ s_2$

$\{P_A\}\ S_1\ \{P_B\}$

$\{P_B\}\ S_2\ \{P_C\}$

$$\frac{\{P_A\}\ S_1\ \{P_B\},\quad \{P_B\}\ S_2\ \{P_C\}}{\{P_A\}\ S_1;\ S_2\ \{P_C\}}$$

Axiomatic Semantics: Selection

- inference rule for selection: if B then S1 else S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\quad \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ Then } S1 \text{ else } S2\ \{Q\}}$$

Axiomatic Semantics: Summary

- difficult to develop axioms and inference rules for all statements in a language
- good for correctness proofs, great for formally reasoning about programs
- not useful for language implementers, nor for language users.

Comparing Semantics

Operational: the state changes are defined by coded algorithms

Denotational: state changes are defined by mathematical functions

Axiomatic: state changes are directly/manually referenced in pre- and post-conditions as needed (a bit ad hoc)

Summary

Syntax: regexs/BNFs/etc. describe the structural syntax of a language.

Static Semantics: Attribute Grammars and semantic functions can further encode the constraints on the structure of well-formed programs.

Dynamic Semantics: No best approach to record/reason about a program's meaning, but we compared operational, denotational, and axiomatic semantics.