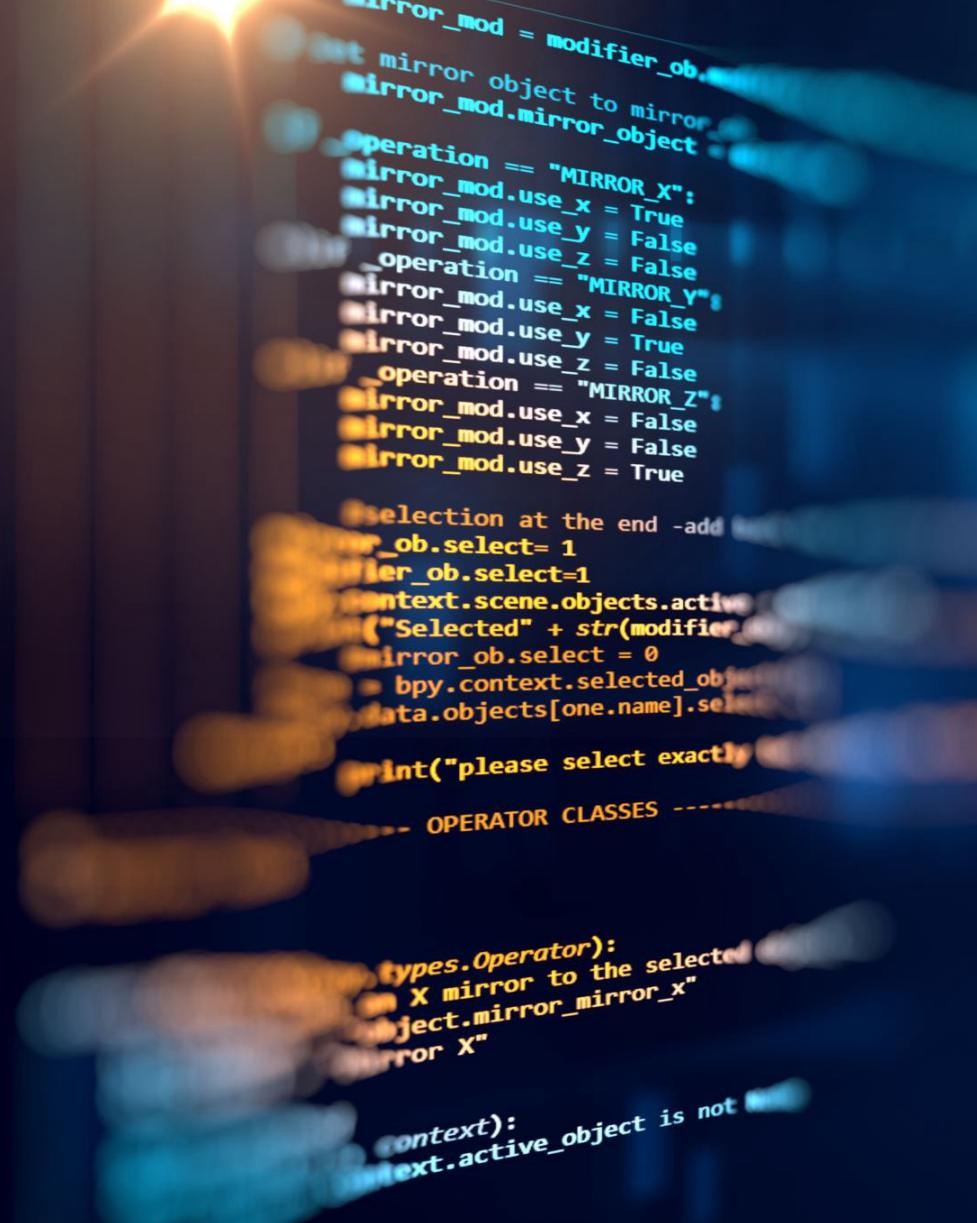


Concepts in Programming Languages: compilation



```
    mirror_mod = modifier_obj
    # mirror object to mirror
    mirror_mod.mirror_object = ob
    if operation == "MIRROR_X":
        mirror_mod.use_x = True
        mirror_mod.use_y = False
        mirror_mod.use_z = False
    elif operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    elif operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    # selection at the end - add
    ob.select= 1
    ob.select=1
    context.scene.objects.active = eval("Selected" + str(modifier))
    mirror_ob.select = 0
    bpy.context.selected_objects.append(mirror)
    data.objects[one.name].select = 1
    print("please select exactly one object")
    print("----- OPERATOR CLASSES -----")

    types.Operator:
        X mirror to the selected
        object.mirror_mirror_x"
        mirror X"
        context):
            context.active_object is not None
```



So many
programming
languages

- Peter J. Landin: “The Next 700 Programming Languages”, CACM (published in 1966!).
- Some programming-language ‘family trees’ (too big for slide):
 - <https://www.levenez.com/lang/>
 - Wikipedia’s historical summary page:
 - https://en.wikipedia.org/wiki/History_of_programming_languages
- This course: pick out interesting programming-language concepts and major evolutionary trends for security.



GLOBAL PROGRAMMING LANGUAGE MARKET



The Business
Research Company



North America
is the largest region in
the programming
language market



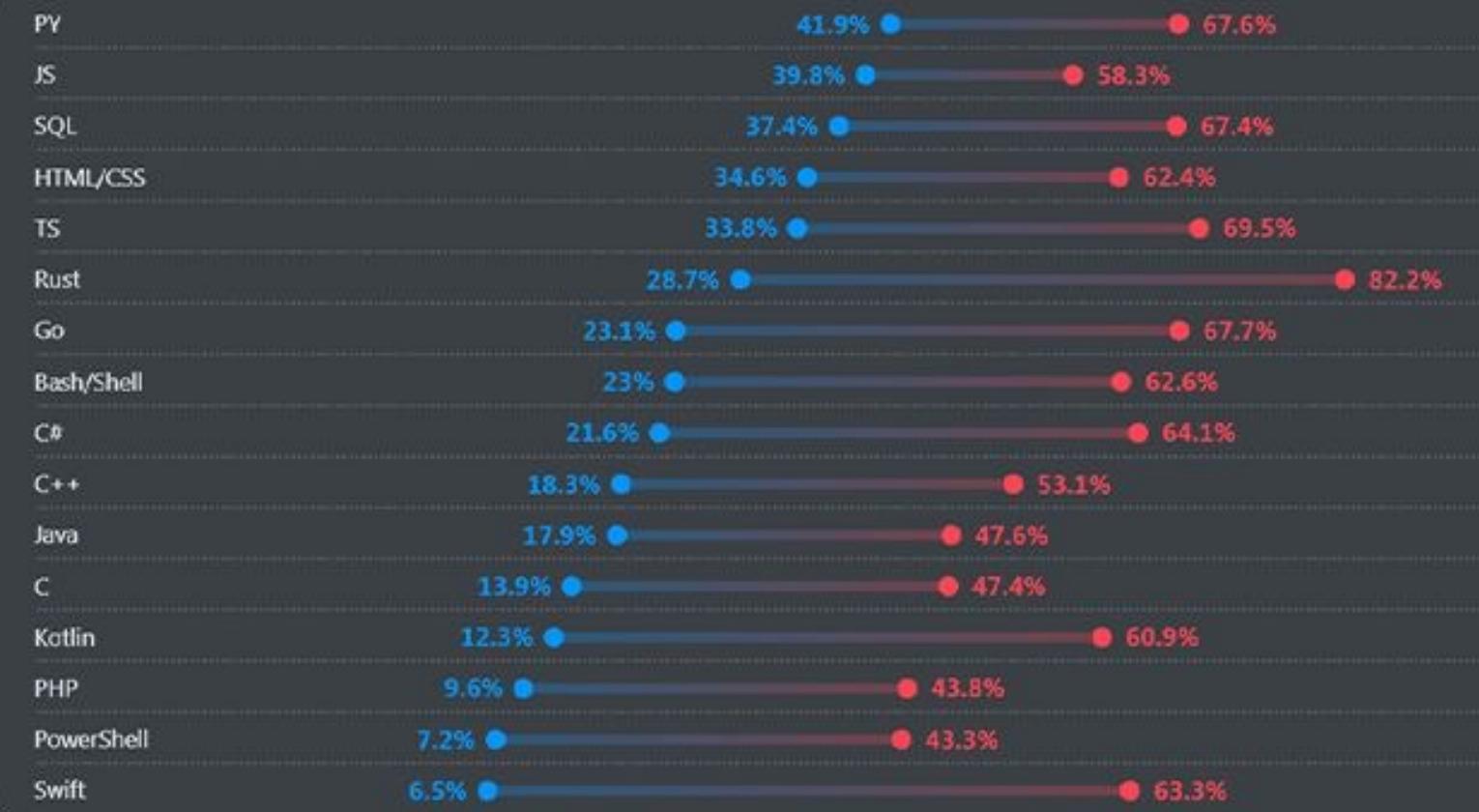
Major Driver
of the programming
language market is
Growth In AI
Expansion

Key Trend

Innovative Programming Languages And
Development Tools Driving Growth And
Profitability

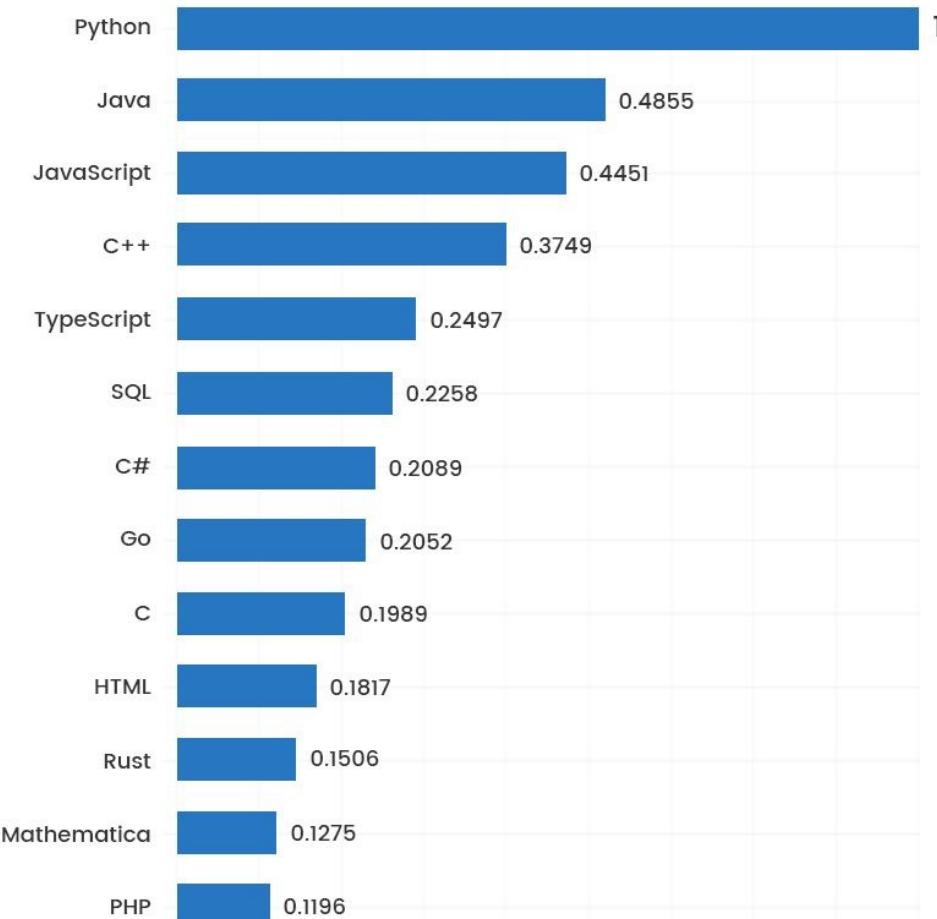


Disclaimer: This slide is a copyrighted communication service of The Business Research Company. The Business Research Company is not liable for damages resulting from the use of this information. All rights reserved. The Business Research Company is a registered trademark of The Business Research Company, Inc. The Business Research Company is not affiliated with any other company.



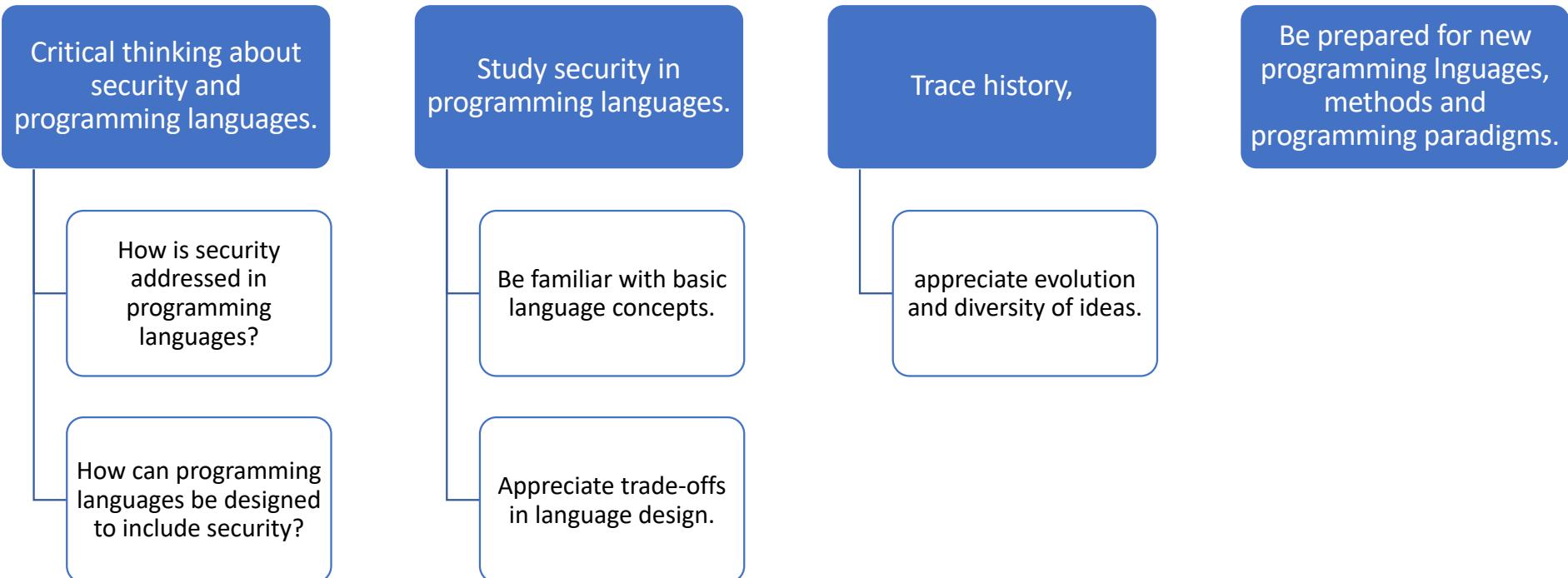
Source : survey.stackoverflow.co

Top Programming Languages



Source : spectrum.ieee.org

The LBTS goals



What makes a good programming language for security?

- Clarity: clean where security is thought about
- Naturalness for the application. Programming feels natural
- Support of abstractions for security. from the very beginning, from design
- Ease of program verification: code will be developed together with a chatbot. But are you sure it works?
- Effective programming language toolchain.
 - F. Logozzo “shift-left” Static detection <- Run-time vulnerability
- Cost of use.
 - Cost of detecting vulnerabilities

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, Peter W. O'Hearn:
Scaling static analyses at Facebook. Commun. ACM 62(8): 62-70 (2019)

Available on TEAMS



What makes a secure programming language successful?

- Expressive power of security abstractions
- Ease of use for the novice.
- Many useful and certified libraries
- Excellent toolchain (including open-source tools)
- Inertia.
- Note the recent trend of big companies to create/control their own languages: C# (Microsoft), Hack (Facebook), Go/Dart (Google), OSwift (Apple), Rust (Mozilla), Python (Dropbox)

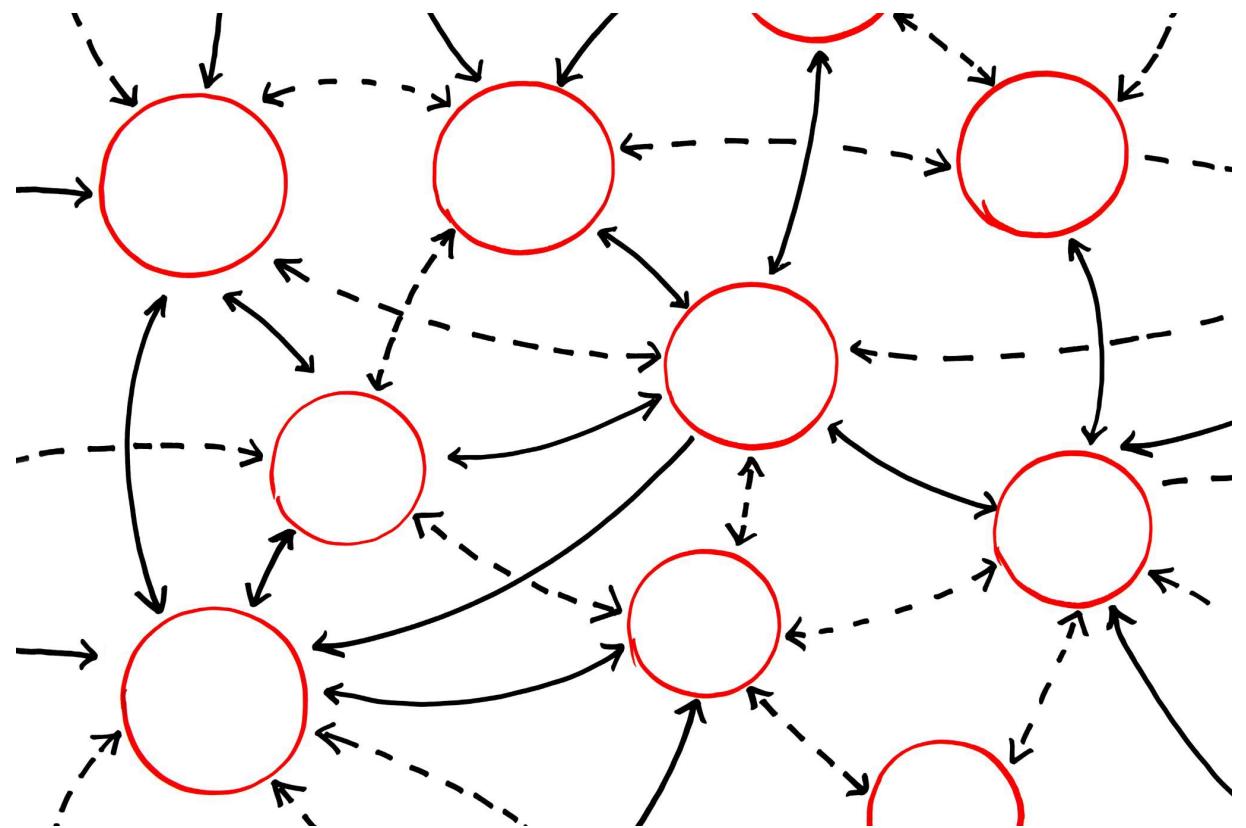
↑ set of tools used to develop, compile and debug programs (compilers, interpreters etc)

↳ There is a trend of big companies developing their own prog-languages.

Programming Languages

- Why so many programming languages?
 - Evolution.
 - Special purposes.
 - No one language is good at expressing all programming styles.
- What makes languages evolve?
 - Changes in hardware or implementation platform
 - Changes in attitudes to safety and security
 - New ideas from academic or industry

Execution Model and Programming Paradigms



Programming language design starts from
abstract machines

Abstract Machine?

△ The design of a prog. lang. starts from def. of abstract machine

In practice, an **abstract machine** is not necessarily a piece of software but rather a **conceptual model**—a formal set of rules that define how a programming language operates. It describes how the language's constructs (variables, loops, function calls, memory management, etc.) behave without being tied to any specific hardware.

An abstract machine typically consists of:

- **A memory model** – How **data is stored**, accessed, and manipulated (stack, heap, registers, etc.).
- **A computation model** – How **operations are performed**, such as arithmetic, branching, and control flow.
- **An execution model** – How **instructions are processed**, including evaluation order, concurrency, and **function calls**.

Example 1: The C Abstract Machine (Low-Level, Hardware-Oriented)

The C standard defines an **abstract machine** that closely resembles real hardware but still has some flexibility for different architectures.

- **Memory Model:**
 - C assumes a linear memory model with explicit stack and heap management.
 - A pointer in C is a direct memory address.
 - You can manually allocate (`malloc`) and deallocate (`free`) memory.
- **Computation Model:**
 - Operations correspond directly to CPU instructions.
 - Integer overflow is **undefined behavior**, meaning the compiler can optimize in ways that assume overflow won't happen.
- **Execution Model:**
 - Function calls are implemented with a **stack** (stack frames).
 - The order of evaluation of expressions (except for `&&`, `||`, `,`) is **not guaranteed** by the standard, giving compilers room for optimization.

Example 2: The Java Virtual Machine (JVM) (High-Level, Abstracted from Hardware)

Java programs run on the **Java Virtual Machine (JVM)**, which defines a different kind of abstract machine.

- **Memory Model:**

- Java has a **garbage-collected heap**, so programmers don't manually free memory.
- Java objects live in the heap, and references (not direct memory addresses) **are used to access them**.

- **Computation Model:**

- The JVM has a **stack-based bytecode execution model** rather than direct CPU instructions.
- Integer overflow is **well-defined** (wrap-around for int, exceptions for BigInteger).

- **Execution Model:**

- Java ensures **strict evaluation order**, meaning expressions are always evaluated left to right.
- The JVM manages **thread synchronization** using memory barriers, ensuring predictable behavior across different CPUs.

So an abstract machine is like a set of rules in your world and everything obeys those rules

Exactly! An abstract machine is like the **laws of physics** in a programming language's world—everything obeys those rules, and they define how things behave.

For example, in the **C abstract machine**, if you access memory outside an allocated region, the behavior is **undefined**—anything might happen, from crashing to corrupting data. But in the **JVM abstract machine**, accessing out-of-bounds memory always results in a **predictable exception** (ArrayIndexOutOfBoundsException).

Abstract Machines

* What happens in the operational environment when a program is executed.

The **abstract machine** describes the fundamental operations, memory management, and execution rules that the language follows.

Key Aspects:

- **Instruction Set:** the basic operations the language supports (e.g., arithmetic, control flow, memory access).
- **Memory Model:** how data is stored, accessed, and managed.
- **Execution Semantics:** how instructions are processed, including evaluation order, function calls, and concurrency handling.
- **Operational Semantics:** how each language construct modifies the machine's state.

Execution Semantics

This describes how a program runs step by step—how instructions are processed, in what order, and how different parts of the program interact. It's about control flow and how the abstract machine organizes execution.

Example 1: Evaluation Order (C vs. Java)

Consider this expression:

```
C  
int x = f() + g();
```

 Copy code

- In C, the order in which f() and g() are evaluated is not defined by the C abstract machine.
 - f() might be called first, then g(), or vice versa—it depends on the compiler.
- In Java, evaluation order is strictly left to right.
 - f() is always executed before g().

So, in C's execution semantics, the compiler has flexibility, while in Java's execution semantics, the order is fixed.

Example 2: Function Calls (Stack Frames in C vs. Tail Recursion in Scheme)

In most languages, when a function is called, a new stack frame is created.

```
C  
int factorial(int n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1);  
}
```

 Copy code

- In C, each function call adds a new frame to the call stack. If n is large, you might get a stack overflow.
- In Scheme (Lisp), which is a functional language, the abstract machine optimizes tail recursion so that recursive calls reuse the same stack frame, avoiding overflow.

This difference is part of each language's execution semantics—how function calls behave internally.

Operational Semantics

This is about **how each individual language construct changes the abstract machine's state** (memory, registers, program counter, etc.).

Think of it like a **CPU's instruction set**—each instruction changes the state of **registers** or **memory**, and operational semantics define that for a programming language.

Example 1: Assignments and Memory Changes (C vs. Python)

C

```
int a = 5;  
a = a + 3;
```

 Copy code

In C:

1. **a = 5**; stores 5 in a specific **memory address**.
2. **a = a + 3**; loads a into a CPU register, adds 3, then stores the new value back.

Python

```
a = 5  
a = a + 3
```

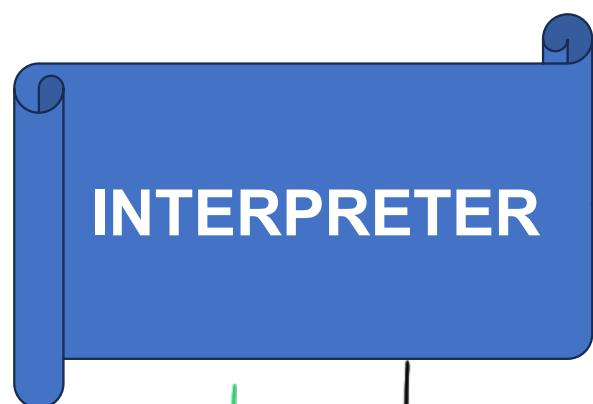
 Copy code

In Python:

1. **a = 5** creates a reference to the **object** 5 in memory.
2. **a = a + 3** doesn't modify the existing value—Python **creates a new object 8** and **makes a reference** to it.

The **operational semantics in C** modify the **same memory location**, while in **Python**, **variables behave more like pointers to objects** rather than direct memory locations.

Abstract Machines



The **abstract machine** describes the fundamental operations, memory management, and execution rules that the language follows.

Key Aspects:

- **Instruction Set:** the basic operations the language supports (e.g., arithmetic, control flow, memory access).
- **Memory Model:** how data is stored, accessed, and managed.
- **Execution Semantics:** how instructions are processed, including evaluation order, function calls, and concurrency handling.
- **Operational Semantics:** how each language construct modifies the machine's state.

Embody capable to
natively execute programs in your language

The two form the foundation of an interpreter

1. Interpreter: "Execute Now, Translate On the Fly"

An **interpreter** reads code **line by line**, translates it into machine instructions **on the fly**, and **immediately executes it**.

How It Works (Step-by-Step)

1. **Reads the source code** (e.g., `print("Hello")`).
2. **Understands and translates** it into machine-level instructions.
3. **Executes the command immediately** without generating a separate executable file.
4. Moves to the next line and repeats.

2. Compiler: "Translate Everything First, Execute Later"

A **compiler** translates the entire program into machine code **before** running it. The resulting **binary (executable)** can be run **without needing the compiler again**.

How It Works (Step-by-Step)

1. **Reads the entire source code** at once.
2. **Translates it into machine code** (produces an .exe or binary file).
3. **Optimizes the code** for performance.
4. **Saves the compiled file** to disk.
5. Later, when you run the program, the CPU executes the pre-compiled machine code.

Abstract Machines

① Helps in defining the *bolts*

- Provides a foundation for language specification and compiler design.
- Enables portability by defining execution independently of physical hardware.
- Helps in reasoning about program correctness,
① security, and optimization.
- *Abstract machine*: the abstract framework that dictates how a programming language operates at a fundamental level, bridging the gap between high-level code and actual execution.

Abstract Machines: Evolution (incomplete)

Fortran: Flat registers; memory arranged as linear array

Lisp: cons cells, read-eval-print loop (the interpreter)

Algol family: stack of activation records; heap storage

Simula: Object references

ML: SECD machine (functions as basic control structure)

Java: Java virtual machine (JVM)

C#: Common Language Run-time

LLVM: Intermediate Representation (IR) as a compiler target for multiple languages.

RUST: Rust Abstract Machine defines safe and unsafe behaviors

AN EXAMPLE: The SEDC machine

The **SECD machine** (Stack, Environment, Control, Dump) is the abstract machine designed to execute functional programming languages. Four key components:

1. **Stack (S)** – Holds intermediate results. Machine has a mechanism to save intermediate results
2. **Environment (E)** – Stores variable bindings. (associate variables to their values)^①
3. **Control (C)** – Holds the program (code) to execute.^②
4. **Dump (D)** – Saves the execution state during function calls^③

① Mechanism that allows to bind identifiers to values

② Contains remaining instructions to still execute and contains the rules to execute the program

③ Used to save current state during function calls.

SECD: execution

Execute the function **fun x. x+2** on the actual parameter 3

Using a C-like syntax

```
int add2(int x) { return x + 2;}
```

```
int main() {
    int result = add2(3); // Apply the function to 3
    printf("%d\n", result); // Output: 5
    return 0;
}
```

We have to translate to the language of Ab-Machine

SECD Step1: Translation

From **(fun x. x+2)3** to the SECD instructions

[3, [LDF [x, ADD 2], AP]]

\uparrow Load function \uparrow Call the function to an argument

- LDF [x, ADD 2] \rightarrow Load the function (ADD primitive op)
- AP \rightarrow Apply the function to argument (AP – Application primitive op)
- ADD 2 \rightarrow Perform the addition.

This is the inverse polish notation

When you write a program in a high-level language (like Python, C, or Java), the code you write is not in the language of the abstract machine (or its instruction set). High-level languages are designed to be human-readable and abstract away a lot of the low-level operations that happen in the machine.

For example, in Python, you might write:

Python

Copy code

```
x = 5 + 3
```

This is a high-level statement. Python itself doesn't know how to execute $5 + 3$ directly as the abstract machine would. Python needs to translate this into something that the abstract machine can actually understand and execute.

The abstract machine might understand operations like:

- Load a constant (5) into a register.
- Add a constant (3) to the register.
- Store the result in memory or a variable.

But the high-level code you wrote doesn't directly correspond to this sequence of low-level operations—it's more abstract, meaning that translation is still necessary.



So when we are talking about the instruction set of the abstract machine we are talking about things like "load this there", or "access this memory location"?

Exactly! When we talk about the instruction set of the abstract machine, we're referring to a set of low-level operations or commands that the abstract machine understands and can execute. These instructions are typically very simple and correspond to basic tasks that a computer performs, like moving data, performing arithmetic, or controlling the flow of execution.

SECD Step 2: execution

Step	Stack	Environment	Control	Dump	Description
1	You have no argument, result []	[]	The program to be exec. [3, LDF [x, ADD 2], AP]	[]	Start with empty stack & environment.
2	[3]	[]	[LDF [x, ADD 2], AP]	[]	Push 3 onto the stack.
3	[3]	[]	[LDF [x, ADD 2]]	[]	Load function fun x. x+2
4	[]	[{x → 3}]	[x, ADD 2] *	[S, E, C]	Apply function (AP), move 3 into the environment.
5	[3]	[{x → 3}]	[ADD 2]	[S, E, C]	Lookup x (which is 3) and push onto stack.
6	[5]	[{x → 3}]	[]	[S, E, C]	Perform 3 + 2 and push result.

* Parameter passing: The actual parameter is 3, so we bound x to 3 and 3 is not in stack anymore. This is call by value (on the top of the stack)

Abstract Machines

- Abstract Machine: a **virtual system that represents the behavior of a physical machine by identifying**

Resources

- The **set of resources required to execute programs**

Instructions

- A **set of instructions specifically designed to operate with these resources**

What your slide is describing is the concept of an **abstract machine** in terms of how it mimics a **physical machine** for executing programs. Let's break it down:

"Abstract machine: a virtual system that represents the behavior of a physical machine..."

- An **abstract machine** is a **conceptual model** or **virtual system** that simulates the behavior of a real computer system.
- It **doesn't correspond directly** to the physical hardware but **mimics** its behavior for the purpose of executing programs.
- Think of it as a **virtualized** version of the actual machine that **abstracts** away the hardware-specific details. It's like creating a software model of how a computer works, which can then be used to execute code without having to rely on the specific physical hardware.

"...by identifying resources (the set of resources required to execute programs)..."

- The **resources** are the components that the **abstract machine** needs to function. This could include things like:
 - **Memory** (for storing data and instructions),
 - **Registers** (for storing intermediate values),
 - **CPU** (the processor that executes the instructions),
 - **Input/Output devices** (for handling data from the outside world).
- These resources are **defined within the model** of the abstract machine, and the machine's behavior is built around how these resources interact with each other to execute programs.

"...and **instructions** (a set of instructions specifically designed to operate with these resources)."

- The **instructions** are the commands that the **abstract machine** can execute using its resources.
- These instructions are designed to perform basic operations, like moving data between memory and registers, performing arithmetic, or controlling the flow of the program.
 - For example, an instruction might tell the abstract machine to:
 - ▶ Load data from memory,
 - ▶ Perform an addition with data stored in registers,
 - ▶ Store the result back into memory.
- These instructions are the **low-level operations** that drive the machine's behavior, and they are part of the **abstract machine's instruction set**.

Putting It All Together

The **abstract machine** is like a **software-based simulation** of a computer system. It defines two main things:

1. **Resources**: What the machine needs (memory, registers, CPU, etc.).
2. **Instructions**: The commands or operations that can be performed using these resources.

By specifying these resources and instructions, the **abstract machine** can execute programs in a way that is similar to how a physical machine would, but in a more **virtualized** or **conceptual** sense.

In Simpler Terms

Imagine you're writing code for a fictional computer system. The **abstract machine** defines what kind of **resources** (like memory or registers) are available and what **instructions** can be used to manipulate those resources. It's like building a computer in software, and then running your programs on it, instead of using actual hardware.

you operate in terms of

Programming Paradigms: Imperative

Procedural

- Fortan, Algol, Pascal, C, Ada, ...

Object Oriented

- Simula, SmallTalk, C++, Java, C#, Scala, RUST ..

Scripting

- Perl, PHP, Python, Javascript, Hack

Objects encapsulate data , and you exploit the methods given by object to interact with data.

Imperative Programming Paradigm

Imperative programming is a programming paradigm where you specify how to perform a task or solve a problem by describing the steps (or instructions) that the computer must follow. It's like giving the computer a set of commands, step-by-step, to achieve a desired outcome.

In imperative programming, you define a sequence of statements or instructions that tell the machine what to do and how to do it. These instructions change the state of the program (like updating variables, memory, etc.).

- You're directly controlling the flow of execution of the program.
- You're managing state changes, which means you explicitly tell the machine what to do with data at every step.

Example: If you wanted to add two numbers, you would write instructions like:

1. Set a variable a to 5.
2. Set a variable b to 3.
3. Add a and b, and store the result in c.

This is imperative programming because it specifies how to achieve the goal by telling the machine the exact steps.

Procedural Programming

Procedural programming is a subset of imperative programming. It focuses on organizing code into procedures or functions—blocks of reusable code that perform specific tasks.

In procedural programming, you break the program into functions that operate on data. These functions can be called in different parts of the program, allowing you to reuse code and keep things modular.

- Procedures/functions are the core structure.
- The focus is on how to break tasks down into smaller steps (procedures), and call them to perform actions.

Object-Oriented Programming (OOP)

Object-Oriented Programming is another programming paradigm that is based on objects and classes. In OOP, you organize your program around objects, which are instances of classes. Each object represents a real-world entity and contains state (data) and behavior (methods/functions).

Instead of writing code that operates on data directly, like in procedural programming, OOP models real-world entities as objects and defines the operations that can be performed on those objects through methods.

Scripting Programming

Scripting programming refers to writing scripts, which are programs that are usually interpreted (not compiled) and are often used to automate tasks, manipulate data, or interact with other programs.

A script is typically written in a scripting language (like Python, JavaScript, or Bash) and is designed to perform a series of tasks quickly and easily. While scripting can be imperative, procedural, or even object-oriented, the key feature is that scripts are often used for automation or solving specific, smaller tasks rather than building large, complex systems.

Programming Paradigms: Declarative

You have no notion of state
↑
(objects are representing states)

Functional

- Lisp, ML, Ocaml, F#, Haskell, ...

Logic

- Prolog, CLP, Datalog, ...

Declarative Programming Paradigm

Declarative programming is a paradigm where you focus on **what** the program should accomplish, **not** how to achieve it. In other words, instead of specifying the exact steps to perform a task, you describe the desired **outcome** or **result**, and the system figures out the steps to get there.

The key difference from imperative programming (where you specify step-by-step instructions) is that in declarative programming, you're more concerned with **describing the problem** than solving it with specific instructions.

For example, in a declarative approach, you may say:

- "Find me all the even numbers in this list."
- You don't need to specify **how** the numbers should be searched, sorted, or iterated over—**the system decides that for you**.

Declarative programming abstracts away the details of how to perform computations and allows the system to focus on the logic or meaning of the task. There are several types of declarative languages, and functional and logic programming are two prominent examples.

Functional Programming Languages

Functional programming is a subparadigm of declarative programming that emphasizes **functions** as the primary building blocks. In functional programming, you **describe** computations as the evaluation of **mathematical functions** rather than a sequence of instructions.

Functional programming avoids changing state or mutable data, which is common in imperative programming. Instead, it relies on **pure functions** (functions that always produce the same output for the same input and have no side effects) and **immutable data**.

Logic Programming Languages

Logic programming is another declarative programming paradigm where you express the problem in terms of logical relationships and rules, and the program's execution is based on **logic inference** (deducing facts from rules).

In logic programming, you typically define a **set of facts** and **rules**, and the system tries to find solutions by applying **these rules** to the facts. The main language used in logic programming is **Prolog**.

Programming Paradigms

- See en.wikipedia.org/wiki/Programming_paradigm for more detail

```
--> mirror_mod.mirror_object
    if not object_to_mirror:
        mirror_mod.mirror_object = None
    else:
        mirror_mod.mirror_object = object_to_mirror
    if operation == "MIRROR_X":
        mirror_mod.use_x = True
        mirror_mod.use_y = False
        mirror_mod.use_z = False
    elif operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    elif operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    #selection at the end -add
    if ob.select==1:
        mirror_ob.select=1
        context.scene.objects.active = mirror_ob
        print("Selected" + str(modifier))
        mirror_ob.select = 0
    bpy.context.selected_objects.append(mirror_ob)
    data.objects[one.name].select = 1
    print("please select exactly one object")
--> OPERATOR CLASSES ----

<types.Operator>:
    X mirror to the selected object.mirror_mirror_x"
    "mirror X"
```

How programming languages are implemented?

Interpreters are based directly on the abstract machine of the language

Two major strategies (roughly speaking!!!!)

- Interpreters (take source code and run it)
- Compilers (translate source code, run result) ①

Interpreters run programs “as is”

- Little or no static analysis and static optimization

Compilers do extensive static analysis:

Most implementations use mixed techniques (interpreter+compiler)

① Take the source code and compile it in the language from the abstract machine provided by the assembler machine
You are given -

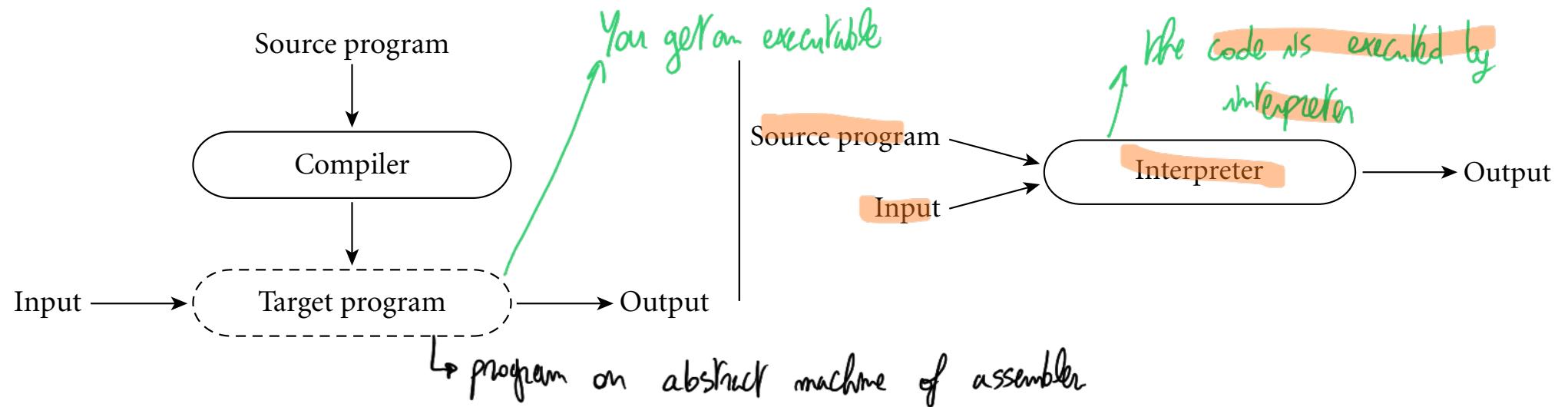
Interpreters typically don't perform **static analysis** or **static optimization** in the same way that **compilers** do because their design and execution models differ fundamentally from those of compilers. Here's why:

1. Dynamic Execution vs. Compilation:

- **Interpreters** execute programs **line-by-line** (or in some cases, block-by-block), **meaning** they **read the source code at runtime and immediately execute it**. They **don't have a full view of the entire program ahead of time**. This makes it difficult for interpreters to perform optimizations like a compiler might, which works with the entire program's source code.
- **Compilers**, on the other hand, process the entire program **before execution** and convert it into an intermediate representation or machine code. This gives compilers the ability to **analyze the entire code and perform static optimizations**—optimizations that don't require runtime information and can be determined from the code itself.

2. Nature of Static Analysis:

- **Static analysis** involves **analyzing the code without actually running it**, checking for things like:
 - **Dead code** (code that is never executed)
 - Constant folding (simplifying expressions like $2 + 3$ into 5)
 - **Type checking and variable analysis**
- This requires the ability to understand the code in its entirety, which is something a **compiler** can do because it can see all the code upfront.
- In contrast, an **interpreter** executes code incrementally and may not have access to the full context or the ability to analyze the entire program in advance.



COMPIILATION VS INTERPRETATION

Compilers

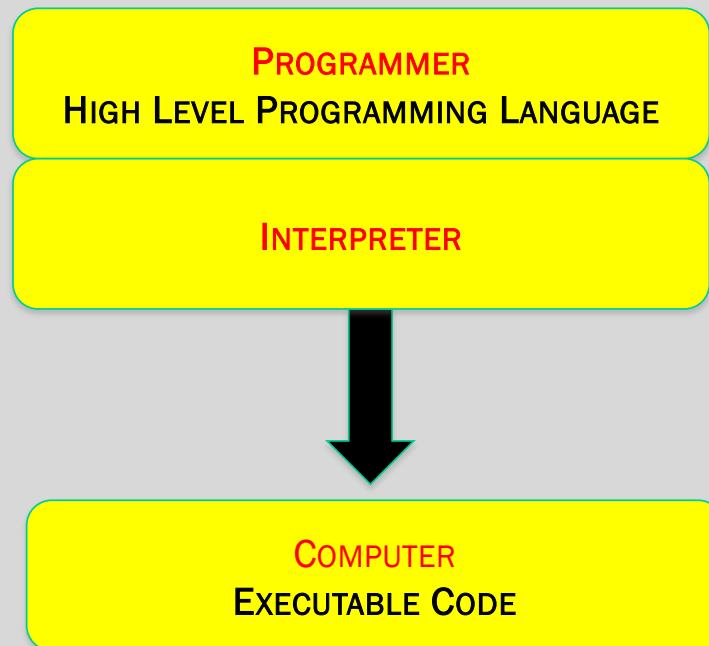
PROGRAMMER
HIGH LEVEL PROGRAMMING LANGUAGE



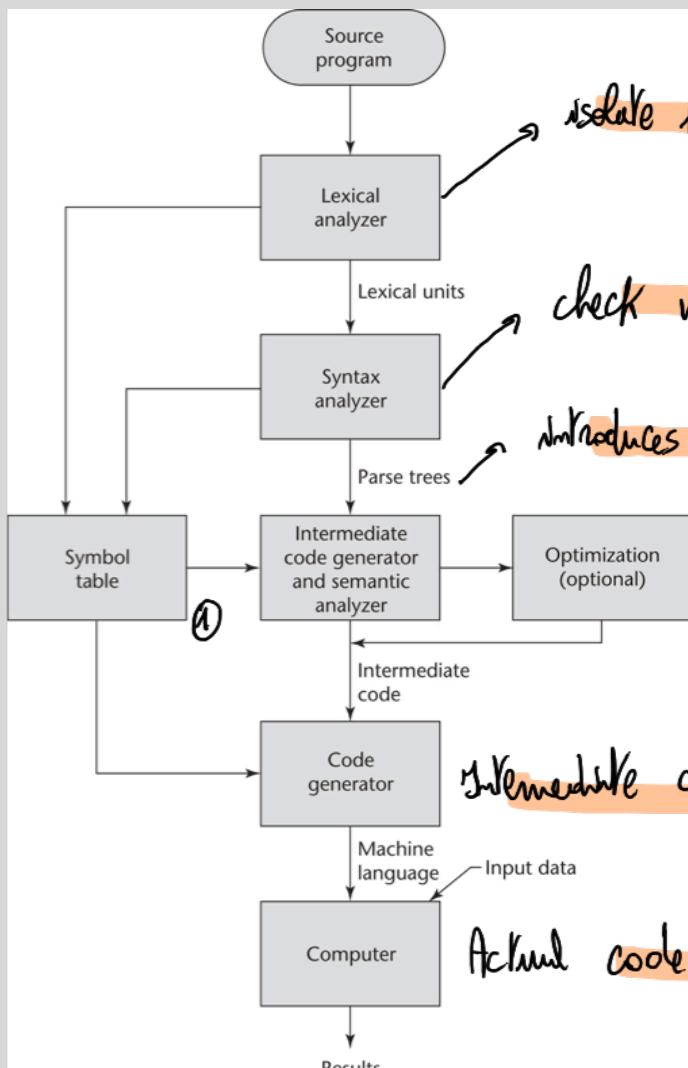
Compilation
Translate the source code into the
object (aka executable) code

COMPUTER
EXECUTABLE CODE

INTERPRETERS



Interpreter:
Execute he instruction of the
High Level Code
*(fetch execute cycle at the programming
language level)*



Copyright ©2016 Pearson Education, All Rights Reserved

isolate from source program the tokens: represent in simpler terms

check whether or not the list of tokens follows the syntax of the language

introduces a tree like module that is an intermediate representation of prog.

THE STRUCTURE OF A COMPILER

Used to do extensive semantic analysis (variables declared before using them, for ex.)

①

Intermediate code generation

Actual code generation

A **lexical analyzer** (also known as a **lexer** or **scanner**) is the first stage in a **compiler**. Its main job is to read the raw source code and transform it into a sequence of **tokens**, which are the meaningful units or building blocks of the program.

Here's a breakdown of what a lexical analyzer does:

1. Breaking the Source Code into Tokens

The primary role of the lexical analyzer is to **break down** the program's source code (which is just a stream of characters) into **tokens**. Tokens are the smallest units of meaning in the programming language, such as:

- **Keywords**: Reserved words like if, else, while, etc.
- **Identifiers**: Variable names, function names, etc.
- **Operators**: +, -, *, /, etc.
- **Delimiters**: Punctuation marks like ;, {, }, (), etc.
- **Literals**: Constants like numbers (42, 3.14) or strings ("hello").

The lexer reads the source code from left to right and identifies these tokens by using the **lexical rules** of the language.

1. Parsing Tokens into a Syntax Tree (Abstract Syntax Tree - AST)

SYNTAX ANALYZER

The primary task of the parser is to organize the tokens into a tree-like structure called a **syntax tree** or **abstract syntax tree (AST)**. This tree represents the hierarchical structure of the program, showing how different parts of the code relate to each other.

- The tree has nodes representing language constructs (like expressions, statements, etc.), and edges that define relationships between them (for example, an assignment statement has a left-hand side and a right-hand side).
- For example, consider the expression `a = b + c;`. The parser would build an AST that reflects the structure of the assignment (=), the variables (a, b, c), and the operation (+).

2. Checking Syntax Validity

The **syntax analyzer** checks that the sequence of tokens respects the **grammar** of the programming language. The grammar defines the rules for forming valid expressions, statements, and other constructs in the language.

- The grammar of a language is often described using **Context-Free Grammar (CFG)** or **Backus-Naur Form (BNF)**, which formally specifies the valid syntax.
- For example, in many languages, a statement like `x = y + 2` must have:
 - A variable (x) on the left side of the assignment operator (=).
 - An expression (`y + 2`) on the right side.
- If the sequence of tokens does not follow these rules, the parser will generate a **syntax error**.

3. Building the Abstract Syntax Tree (AST)

Once the parser has checked that the syntax is correct, it builds an **abstract syntax tree** (AST). The AST is a simplified, tree-like representation of the program's structure. It captures the essential elements of the program and how they are related, omitting unnecessary details (such as parentheses in expressions).

- For example, the expression $x = y + z * 2$ would be parsed into a tree where the multiplication ($z * 2$) is evaluated before the addition ($y + (z * 2)$), showing the precedence and associativity of operations.
- In an AST, the operators (like $+$, $*$, $=$) are usually internal nodes, while the operands (like variables and constants) are the leaves.

A **symbol table** is a data structure used by a compiler to store information about the variables, functions, objects, and other symbols used in a program during its compilation. The purpose of the symbol table is to provide a centralized place where the compiler can keep track of symbols and their associated attributes as it processes the code.

The **symbol table** contains information such as:

- **Identifiers:** Names of variables, functions, classes, etc.
- **Data types:** The data type of each identifier (e.g., int, float, char, etc.).
- **Scope information:** The scope in which the symbol is valid, such as whether it is a local or global variable.
- **Memory location:** Where the symbol (variable, function, etc.) is stored, such as the memory address or offset.
- **Other attributes:** Depending on the language, this could include information like function parameters, return types, visibility (public/private), etc.

1. Semantic Analyzer

The **semantic analyzer** ensures that the program is **semantically correct**. While the syntax analyzer checks if the program follows the rules of the language's grammar, the semantic analyzer checks if the program's constructs make sense according to the language's meaning or logic.

This includes:

- **Type Checking:** Ensures that operations between variables make sense based on their types. For example, if you try to add a string and an integer, this would be flagged as a semantic error in most statically-typed languages.
- **Variable Declaration and Use:** Ensures that variables are declared before use and that functions or variables are used in the correct scope. For example, using a variable `y` without declaring it first would be a semantic error.
- **Function Call Validity:** Checks that function calls match the function's declared signature (correct number and types of arguments).
- **Constancy:** Ensures that constant values are not modified.

2. Intermediate Code Generator

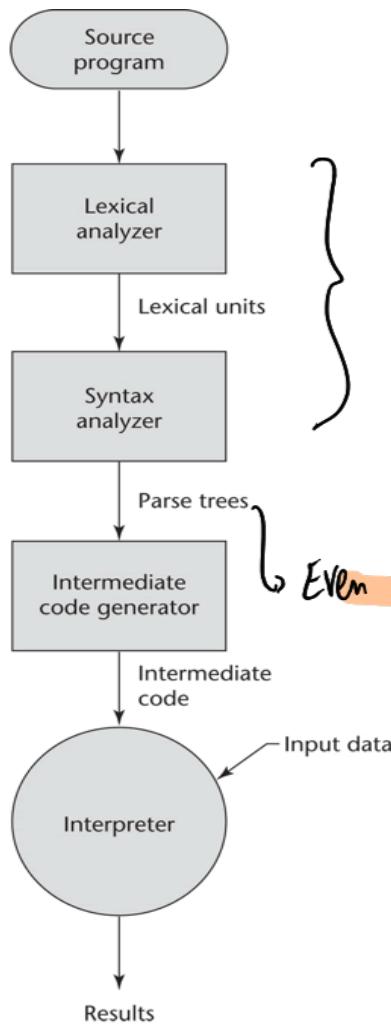
After the semantic analysis, the **intermediate code generator** takes the **abstract syntax tree (AST)** produced by the **syntax analyzer** and generates **intermediate code** (also known as **intermediate representation** or **IR**). This intermediate code is **not machine code**, but it is **an abstraction that is easier to manipulate than the original source code**.

- The intermediate code is **usually designed to be platform-independent** (i.e., not specific to any one machine or processor).
- It **represents the program at a lower level than the source code but is still high-level enough to be optimized or translated into machine code later.**

CODE OPTIMIZATION, WHICH IS OPTIONAL

4. Code Generator

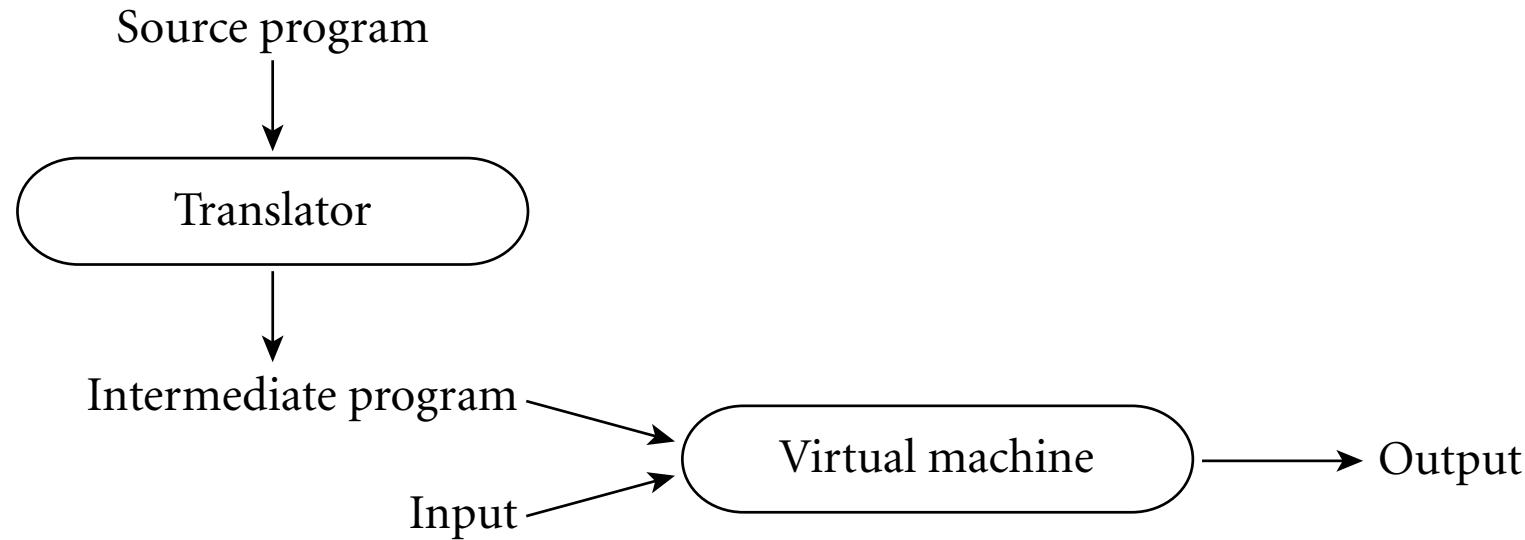
The **code generator** is the final component that takes the **intermediate code** (which has been optimized, if applicable) and generates the **final machine code** or assembly code specific to the target architecture (like x86, ARM, or another platform).



THE STRUCTURE OF AN INTERPRETER

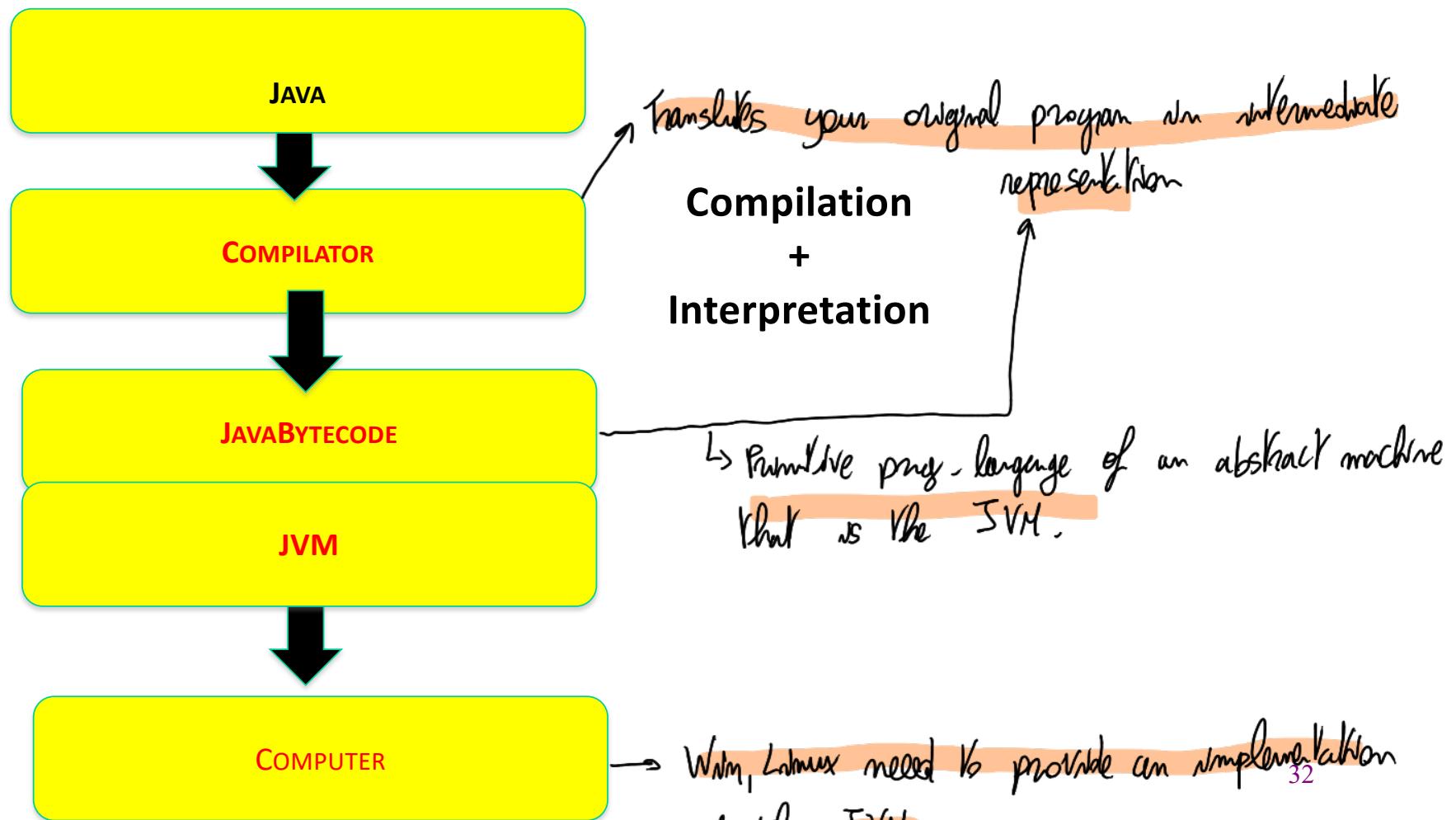
Even an interpreter has to check if program is well written. It executes the abstract syntax tree of a program, a more refined representation.

Java is an example



MIXED IMPLEMENTATION

Example: The Java Programming language



Java bytecode verifier verifies if code is compliant with the specifications of the JVM, the rules of Java.

JVMs are in the network! So you need this.

The **Java bytecode verifier** is a crucial component of the Java Virtual Machine (JVM). Its primary role is to ensure that the bytecode loaded into the JVM is **safe** and **valid** before it is executed. This is an important security feature because Java programs (or applets) can come from untrusted sources, and the verifier ensures that the bytecode does not perform illegal or harmful operations that could compromise the integrity of the JVM or the system on which it runs.

Here's what the Java bytecode verifier does in more detail:

1. Checking Bytecode Structure

The verifier checks that the bytecode is well-formed according to the Java Virtual Machine specification.

This includes:

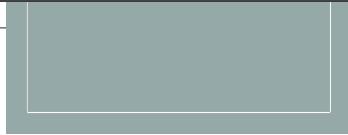
- Ensuring that the bytecode follows the correct syntax and formatting.
- Verifying that all bytecode instructions are valid for the version of the JVM being used.
- Ensuring that methods, fields, and other elements in the bytecode are correctly referenced and aligned with the class file structure.

this is because an attacker could modify the bytecode to be executed right? So if we only checked for validity for the java code we could have this problem

Exactly! You're on the right track. If we only checked the **validity of the Java source code** (before it is compiled into bytecode), we would be leaving the system vulnerable to malicious modifications in the compiled bytecode itself.

Compilation and Lexical Analysis





COMPILATION IN A NUTSHELL

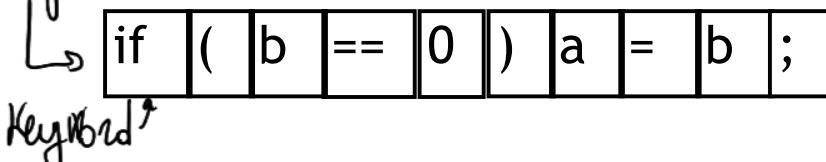
FRONT-END

Source code
(character stream)

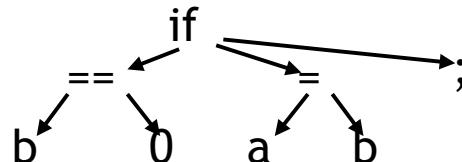
if (b == 0) a = b;

Basic elements of the language

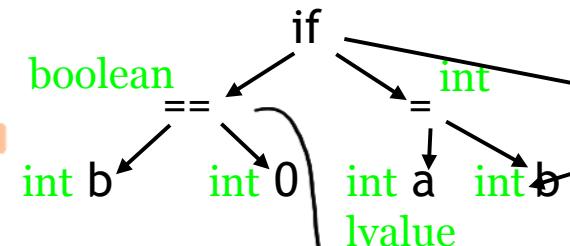
Token
stream



Abstract syntax
tree (AST)



Decorated AST



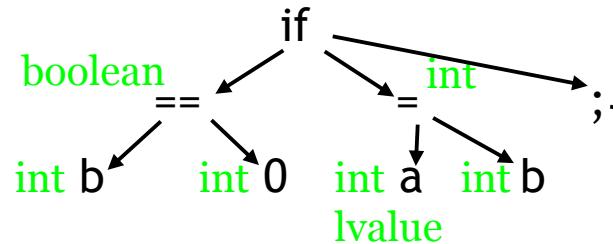
Contains information about types → result is a boolean!

Lexical analysis

Parsing

Semantic Analysis

BACK END



Intermediate Code Generation

```
SEQ(CJUMP(TEMP(b) == 0, L1, L2),  
    LABEL(L1),  
    TEMP(a) = TEMP(b)  
    LABEL(L2))
```

Code Generation

```
cmp rb, 0  
jnz L2  
L1: mov ra, rb  
L2:
```

Register allocation, optimization

```
cmp ecx, 0  
cmovz [ebp+8], ecx
```

We generate the
intermediate representation

And then the
code

LEXICAL ANALYSIS

Source code
(character stream)

if (b == 0) a = “hi”;

Token stream

if | (| b | == | 0 |) | a | = | “hi” | ;

Lexical analysis

Lexical analysis converts program source into tokens (keywords, variable names, operators, numbers, etc.)

Parsing

Semantic Analysis

TOKENS

- Identifiers: x y11 elsex _i00
- Keywords: if else while break
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1.
1e5 0.e-10
- Symbols: + * { } ++ < << [] >=
- Strings: "x" "He said, \"Are you?\""
- Comments: /** don't change this **/

All the basic elements in a language

Ad-hoc lexer: it identifies a token.

READ
IDENTIFIER
TOKENS

Reader input;

...

```
while (true) {  
    int c = input.read();  
    if (!isIdChar(c)) break;  
    id.append((char) c);  
}
```

Identifiers cannot start with a number.

Ad-hoc lexer

It reads, and discards, the character just after the identifier.

This might not be a problem if the character is whitespace, but in general the discarded character will be part of the next token in the input stream.

*a = 2 + 2
with no space*

Reader input;

...

```
while (true) {  
    int c = input.read();  
    if (!isIdChar(c)) break;  
    id.append((char) c);  
}
```

Tokens and Tokenize (in Ocaml)

```
type token =
| Tok_Num of char
| Tok_Sum
```

```
tokenize "1+2" =
[Tok_Num '1'; Tok_Sum;
Tok_Num '2']
```

A Lexical Analyzers in Ocaml (1)

```
(*  
  This function tokenizes a given string into a list of tokens.  
  It only recognizes single-digit numbers and the plus operator.  
  If an unrecognized character is encountered, it raises an exception.  
*)  
let tokenize (s: string) = (* returns token list *)  
  
(* Define a regular expression to match a single digit (0-9) *)  
let re_num = Str.regexp "[0-9]" (* single digit *)  
  
(* Define a regular expression to match the plus sign.*)  
  
let re_add = Str.regexp "+"
```

A Lexical Analyzers in Ocaml (2)

```
(* 'tok' is a helper function that takes the current position 'pos' in 's' *)
let rec tok pos s =
(* If the current position is at or beyond the end of the string, return an empty list *)
  if pos >= String.length s then []
    else (* Check if the character at position 'pos' matches a digit *)
      if (Str.string_match re_num s pos) then
        let token = Str.matched_string s in
(* Create a token for the digit and continue tokenizing from the next character *)
          (Tok_Num token.[0]) :: (tok (pos + 1) s)
        (* If it is not a digit, check if it matches the plus operator *)
        else if (Str.string_match re_add s pos) then
          (* Create a token for the plus operator and continue *)
          Tok_Sum :: (tok (pos + 1) s)
        else
          (* If the character does not match any known token, raise an error *)
          raise (IllegalExpression "tokenize")
```

A Lexical Analyzers in Ocaml (3)

```
(* This function tokenizes a given string into a list of tokens.  
It only recognizes single-digit numbers and the plus operator.  
If an unrecognized character is encountered, it raises an exception.  
*)  
let tokenize (s: string) = (* returns token list *)  
  
(* Define a regular expression to match a single digit (0-9) *)  
let re_num = Str.regexp "[0-9]" (* single digit *)  
  
(* Define a regular expression to match the plus sign.*)  
let re_add = Str.regexp "+"  
  
let rec tok pos s =  
  < code>  
  in  
  (* Start tokenizing from position 0 *)  
  tok 0 str
```

Issues

- What kind of token we are going to read from seeing first character?
 - if token begins with “i” is it an identifier? Or a type?
 - if token begins with “2” is it an integer? Or a float?
 - tokens with common prefixes are hard to tokenize correctly.
 - Either: complex code (hard to maintain), or backtracking⁽¹⁾
- More principled approach: **lexer generator** (e.g., lex, Jflex) generates efficient tokenizer automatically from a lexical specification.
- The theory helps a lot!!!

⁽¹⁾ I start by guessing it is an identifier, then I get that and a whitespace. So I have to backtrack and do it again.

Theory to describe the tokens of a language

↑

Lexer generator: issues

-
- How to describe tokens *unambiguously*
 - How to break text up into tokens
 - How to tokenize efficiently tokens with similar prefixes



How to describe tokens

- Programming language tokens can be described using **regular expressions**
- Regular expression R describes a set of strings $L(R)$:
 $L(R)$ is the “language” defined by R
 - $L(abc) = \{ abc \}$
 - $L(\text{hello}|\text{goodbye}) = \{\text{hello, goodbye}\}$
 - $L([1-9][0-9]^*) = \text{all positive integer constants}$
 - $L(X(Y|Z)) = L(XY|XZ) = L(XY) \cup L(XZ)$
- Idea: define each kind of token using RE

Regular Expressions

- a an ordinary character stands for itself
- ϵ the empty string (actually written as empty string in lexer specs)
- $R|S$ any string from either $L(R)$ or $L(S)$:
$$L(R|S) = L(R) \cup L(S)$$
- RS string from $L(R)$ followed by one from $L(S)$:
$$L(RS) = \{rs \mid r \in L(R) \wedge s \in L(S)\}$$
- R^* zero or more strings from $L(R)$, concatenated (“Kleene star”)
$$\epsilon | R | RR | RRR | RRRR \dots$$

Examples

Regular Expression

a

ab

a | b

ϵ

(ab)*

Strings in L(R)

“a”

“ab”

“a” “b”

“ ”

“ ” “ab” “abab” ...

Implementing Regular Expressions

We implement a regular expression by turning it into a finite automaton

Finite automaton

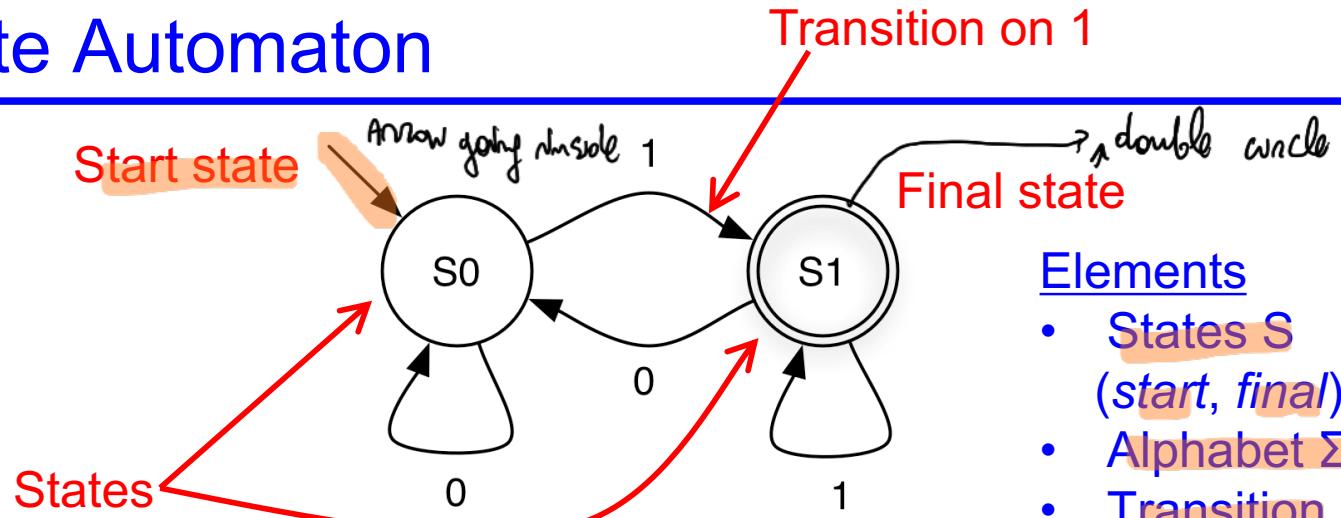
A “machine” for recognizing a regular language

Machine implementing regex: finite state
automata

The blackboard contains several mathematical notes and diagrams:

- At the top left, there is a summation formula: $\sum_{k=0}^{\infty} x^k \cdot 7^{k+3} = \frac{7x}{1-x}$.
- To the right of the summation, there is a square root calculation: $\sqrt{2656.56} = 51.52$.
- Below the summation, there is a diagram of a finite automaton with states labeled s_1 , s_2 , and s_3 . Transitions are shown with arrows labeled a and b . A final state is marked with a double circle.
- Next to the automaton, there is a formula: $D(x) = a + b + 4.31447$.
- Below the automaton, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- Further down, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- On the left, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- In the center, there is a complex fraction: $\frac{2x^2 + 3x^2}{y^2} = \frac{5x^2}{y^2}$.
- Below the fraction, there is a formula: $xy = c$.
- Next to the fraction, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- On the right, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- At the bottom left, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- On the far right, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- At the bottom, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.
- At the very bottom, there is a circle containing a shaded area with a radius labeled r . To its right, there is a formula: $\pi r^2 = \pi$.

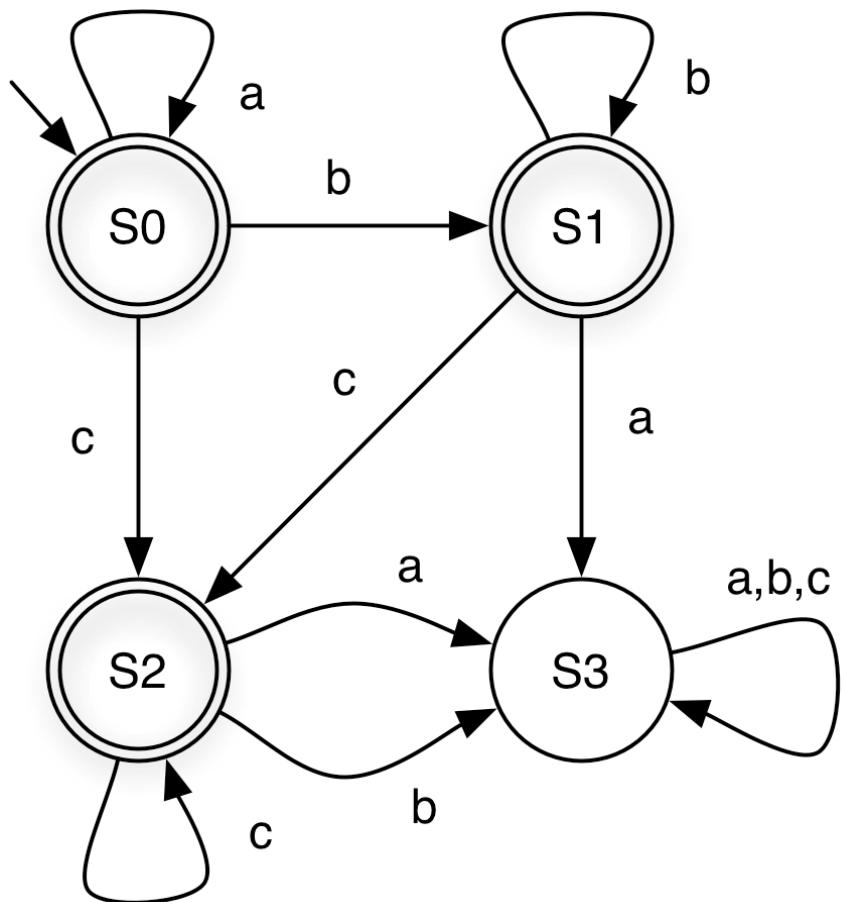
Finite Automaton



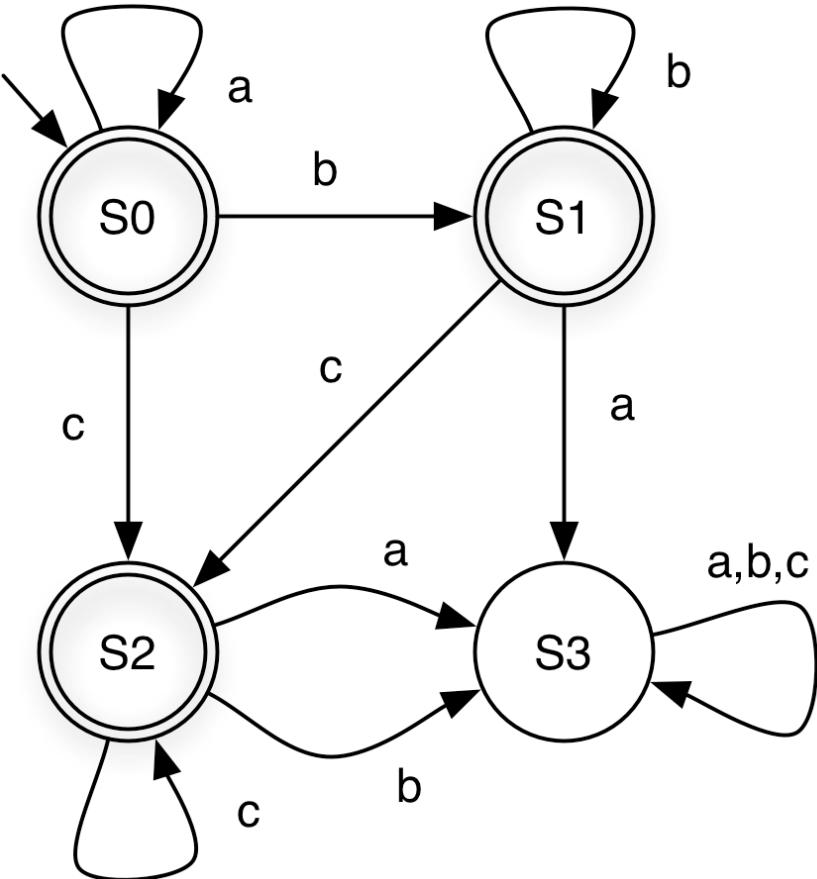
- Machine starts in **start** or **initial state**
- Repeat until the end of the string s is reached
 - Scan the next symbol $\sigma \in \Sigma$ of the string s
 - Take **transition edge** labeled with σ
- String s is **accepted** if automaton is in **final state** when end of string s is reached

- Elements
- States S
 - (start, final)
 - Alphabet Σ
 - Transition edges δ

Trans. function takes one state, a symbol and gives the next state.



string	state at end	accept s?
aabcc		



string	state at end	accept s?
aabcc	S2	Y

If we are in a final state,
the automata recognises the
string.

Regular expressions are mapped to non deterministic finite state automata, then

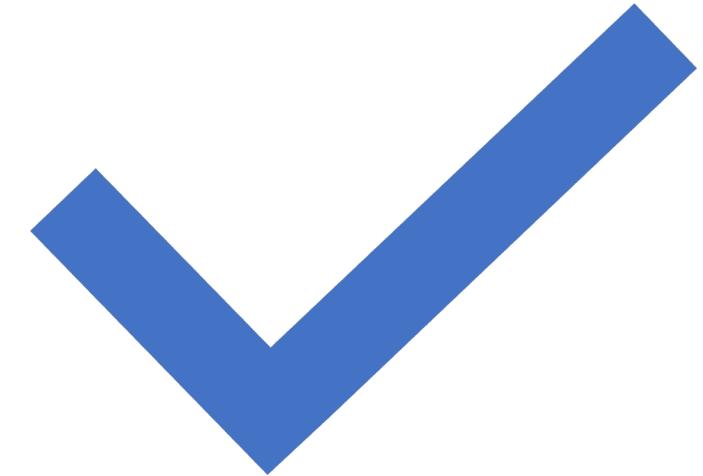
From a regular expression we can construct the minimal automata for that regex

Lexer specification = collection of all the automata recognising the regex for our tokens,

- Input to lexer generator:
 - List of regular expressions in priority order
 - associated action for each RE (generates appropriate kind of token, other bookkeeping)
- Output:
 - program that reads an input stream and breaks it up into tokens according to the REs (or reports lexical error)

Lexical Analysis

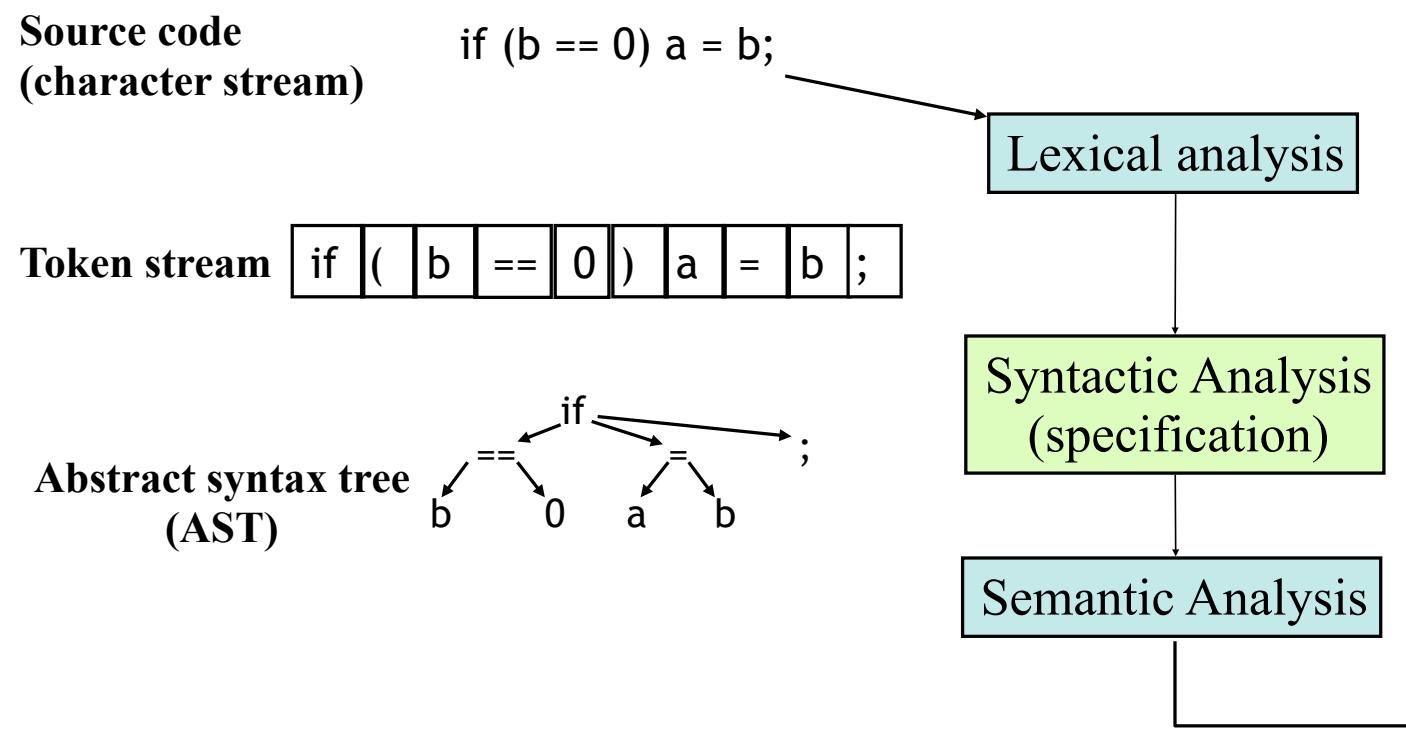
- Notes on Lexing are available on TEAMS,
- section READINGS



Exercized (Design Finite State Automaton)

- ▶ That accepts strings containing two consecutive 0s followed by two consecutive 1s
- ▶ That accepts strings with an odd number of 1s
- ▶ That accepts strings containing an even number of 0s and any number of 1s
- ▶ That accepts strings containing an odd number of 0s and odd number of 1s
- ▶ That accepts strings that **DO NOT** contain odd number of 0s and an odd number of 1s

What next



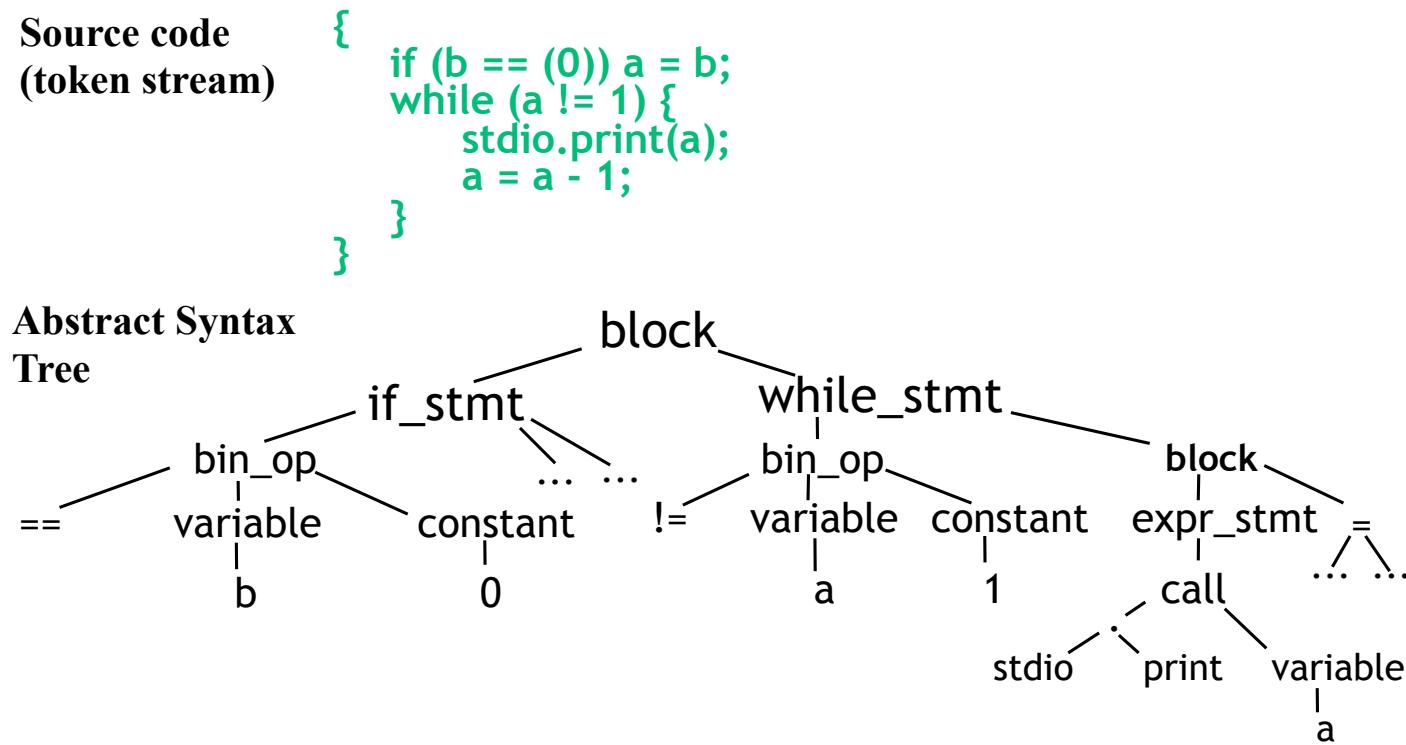
Syntactic Analysis and Parsing

Parsing

Goal: recognizing whether a program is grammatically well-formed.

We have the list of tokens. Still a symbolic representation.

Parsing



Parsing: structure

- **Input:** stream of tokens
- **Output:** abstract syntax tree
- **Implementation:**
 - Parse token stream to traverse concrete syntax (**parse tree**)
 - During traversal, build **abstract syntax tree**
 - Abstract syntax tree removes extra syntax
 - $a + b = (a) + (b) = ((a)+(b))$

↑ Tree you get from
traversing
of the grammar

What parsing does not do

- Doesn't check many things: type agreement, variables declared, variables initialized, etc.

int x = true;

int y;

z = f(y);

- Deferred until semantic analysis

Specify language syntax

- First problem: how to describe language syntax precisely and conveniently
- Why not use regular expressions (on tokens) to specify programming language syntax?

Limits of Regular Expressions

- Programming languages are not regular -- cannot be described by regular exprs
- Example: the language of all strings that contain balanced parentheses (easier than PLs)
$$() ()() ()()() ((())())()$$
- Problem: need to keep track of number of parentheses seen so far: unbounded counting
A finite state automata cannot do that.

We need a *different* theory

Context Free Grammars

- A specification of the balanced-parenthesis language:
 $S \rightarrow (S)S$
 $S \rightarrow \epsilon$
- The definition is recursive
- A **context-free grammar**
 - More expressive than regular expressions
 - $S = (S) \epsilon = ((S) S) \epsilon = ((\epsilon) \epsilon) \epsilon = ()$

If a grammar accepts a string, there is a *derivation* of that string using the productions of the grammar

What is a Context-Free Grammar (CFG)?

A **Context-Free Grammar (CFG)** is a formal grammar used to describe the syntax rules of a programming language, and it is particularly useful for defining programming languages with nested structures (like expressions, statements, and blocks). CFGs allow us to express more complex syntactic structures than regular expressions can, which is why they are often used in language design (including programming languages).

A **CFG** is made up of a set of rules or **productions** that define how **symbols** (both terminal and non-terminal) can be replaced or rewritten. These rules are used to generate strings in a language.

In a programming language, **syntax** refers to the set of rules that define the structure of valid statements or instructions. These rules govern how code must be written to be understood by the compiler or interpreter.

In C, some key syntax rules include:

1. **Keywords:** Reserved words with predefined meanings (e.g., `int`, `if`, `return`).

- Example:

The screenshot shows a code editor window with a dark theme. In the top left corner, there is a small icon labeled 'c'. In the top right corner, there are two buttons: 'Copia' (Copy) and 'Modifica' (Modify). The main area contains the following C code:

```
int main() {
    return 0;
}
```

2. **Variables and Data Types:** Variables must be declared with a type before use.

Components of a Context-Free Grammar:

1. **Non-Terminals:** These are **symbols** that can be replaced by other symbols according to the grammar rules. They are placeholders for syntactic categories like "expression," "statement," etc. Non-terminals are usually written in **capital letters** (e.g., `E`, `S`, `A`).
2. **Terminals:** These are the actual symbols of the language (the "alphabet" of the language). They represent the basic building blocks of the language, such as **keywords**, **identifiers**, **operators**, etc. Terminals are usually written in **lowercase** or as literal symbols (e.g., `if`, `+`, `x`).
3. **Production Rules:** These define how non-terminals can be expanded into other non-terminals or terminals. A production has the form:

Definition (hint)

- Terminals
 - Token or ϵ
 - Non-terminals
 - Syntactic variables
 - Start symbol
 - A special nonterminal is designated (S)
 - Productions
 - Specify how non-terminals may be expanded to form strings
 - LHS: single non-terminal, RHS: string of terminals or non-terminals
 - Vertical bar is shorthand for multiple prod'ns
- $$S \rightarrow (S) S$$
- $$S \rightarrow \epsilon$$

What Does the Start Symbol Mean?

In the context of **Context-Free Grammars** (CFGs), the **start symbol** is the **first non-terminal** from which the process of deriving valid strings in the language begins. This non-terminal is special because it represents the **highest-level structure** in the grammar, and the **goal of the derivation process** is to expand the start symbol into a valid string of **terminals** (the actual symbols of the language).

Derivation Process

Let's break it down using the **balanced parentheses example**.

- The start symbol, in your case, is **S**. This means that, to generate a valid string in the language of balanced parentheses, we begin with **S**.
- Derivation refers to the process of applying the production rules to replace non-terminals with other non-terminals or terminals, starting from the start symbol.

so to have a working grammar we need to specify what is the starting symbol we have to assume we work with when we start with a string, correct?

Yes, exactly! To have a **working grammar**, you need to specify a **starting symbol**. The starting symbol is essential because it serves as the **entry point** for the process of generating or parsing strings in the language defined by the grammar.

Example

$$S \rightarrow E + S \quad | \quad E$$

$$E \rightarrow \mathbf{number} \quad | \quad (S)$$

e.g. $(1 + 2 + (3+4))+5$

$$\begin{array}{l} S \rightarrow E + S \\ S \rightarrow E \\ E \rightarrow \mathbf{number} \\ E \rightarrow (S) \end{array}$$

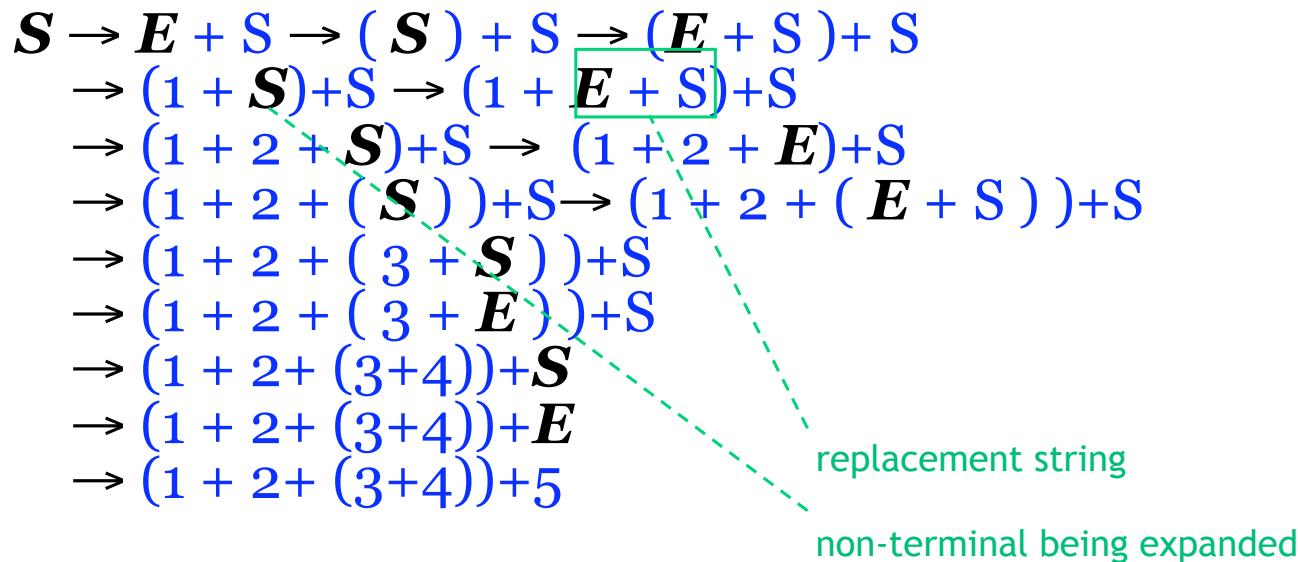


4 productions
2 non-terminals (S, E)
4 terminals: (,), +, number
start symbol S

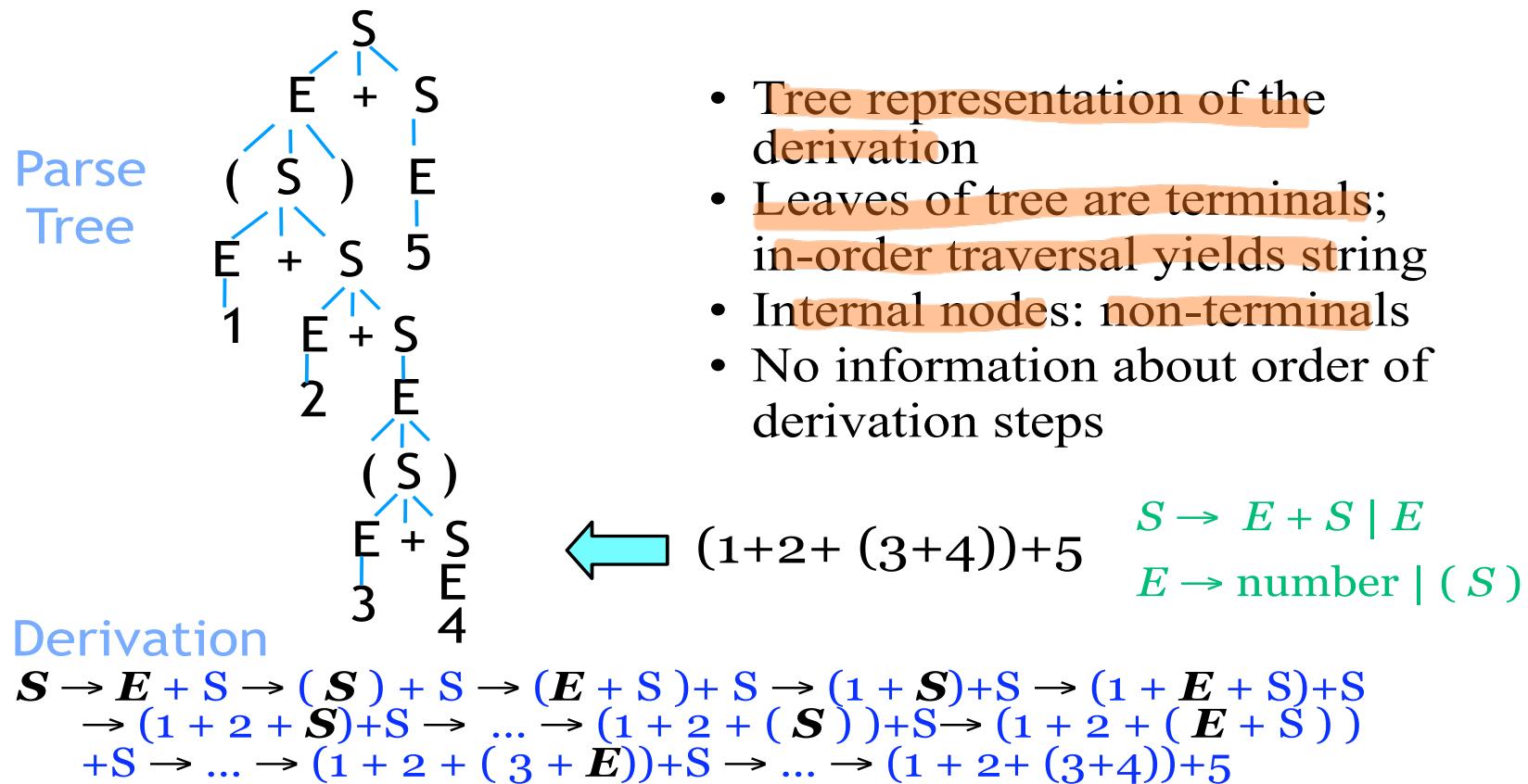
Derivation

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{number} \mid (S) \end{aligned}$$

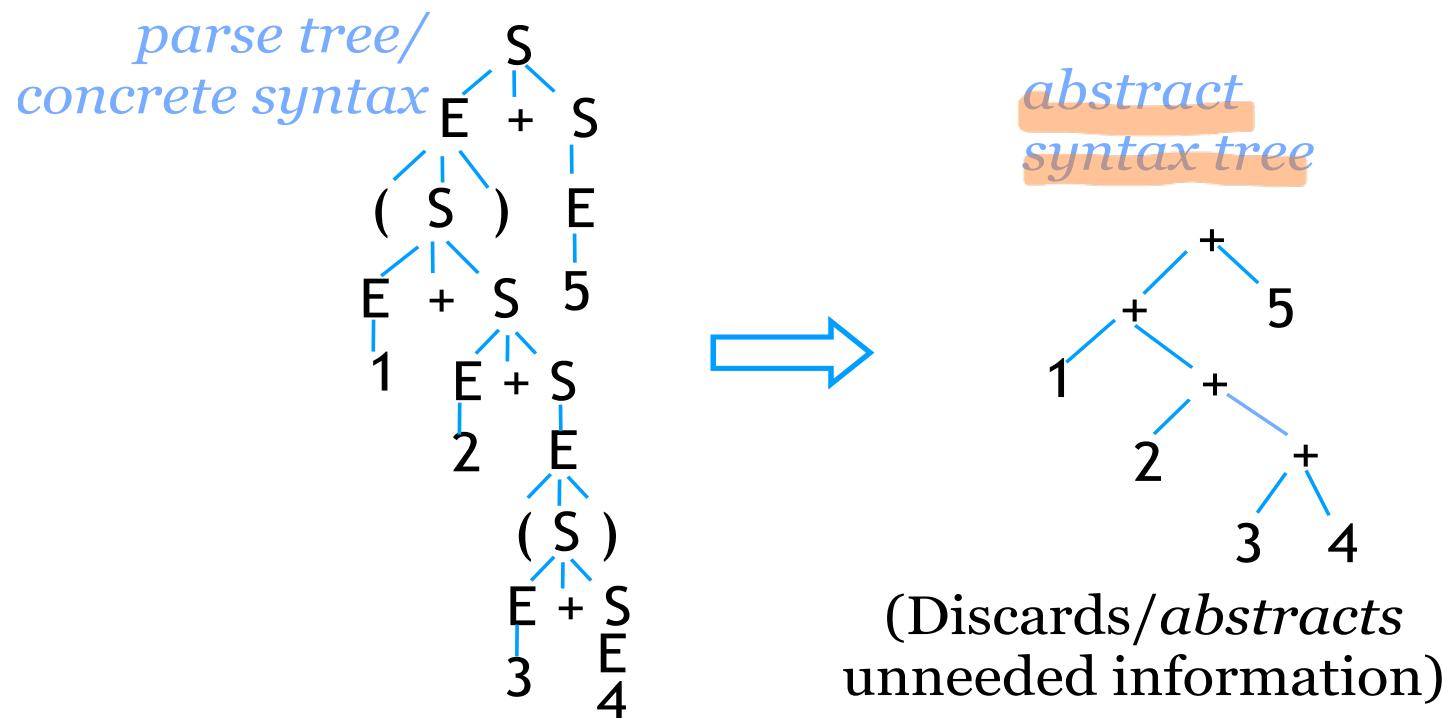
Derive $(1+2+(3+4))+5$:



From derivation to parse tree



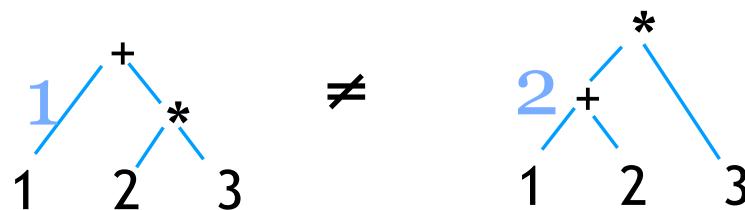
From parse tree to abstract syntax tree



Ambiguity

$$S \rightarrow S + S \mid S * S \mid \text{number}$$

- Consider expression $1 + 2 * 3$
- Derivation 1: $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$
- Derivation 2: $S \rightarrow S * S \rightarrow S * 3 \rightarrow S + S * 3 \rightarrow S + 2 * 3 \rightarrow 1 + 2 * 3$



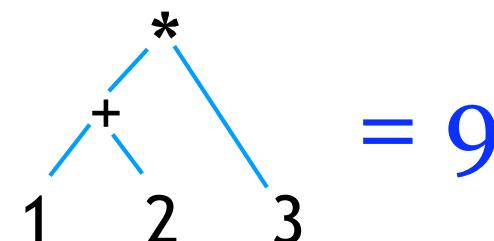
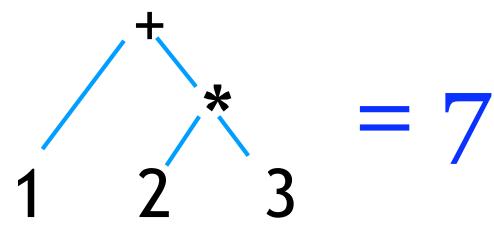
Ambiguity

Different derivations produce different parse trees:

ambiguous grammar

Different parse trees correspond to different evaluations!

Meaning of program not defined



Yes, exactly! When a grammar allows for different derivations that produce different parse trees for the same string, the grammar is **ambiguous**. This means that there is more than one way to interpret the structure of a given input, which leads to **uncertainty in meaning**—a serious problem in programming languages.

Why is Ambiguity a Problem?

If a grammar is ambiguous, it means that a program written using that grammar could have multiple valid interpretations. Since programming languages need **precise meaning**, ambiguity is unacceptable. Different parse trees might lead to different **execution behaviors**, making it unclear what the program is supposed to do.

The Problem:

- The **same input** (`id + id * id`) results in **two different parse trees**.
- This **means two different orders of operations**, which leads to different results.
- Since programming languages need a single, well-defined meaning for expressions, **this ambiguity must be resolved**.

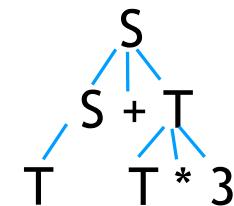
Eliminating Ambiguity

Top level mode is always "+".

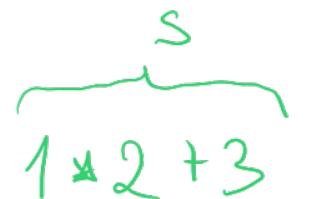
- Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left

$$S \rightarrow S + T \mid T$$

$$T \rightarrow T^* \text{num} \mid \text{num}$$



- T non-terminal enforces precedence¹₂
- Left-recursion : left-associativity



How Do We Fix Ambiguity?

1. **Modify the Grammar:** We can explicitly enforce operator precedence by changing the grammar.

For example:

```
r Copia Modifica
```

```
E → E + T | T
T → T * F | F
F → id
```

This ensures that `*` has higher precedence than `+`, forcing multiplication to be evaluated first.

2. **Use Parsing Rules:** Some parsers (like LR parsers) resolve ambiguity by setting precedence and associativity rules.
3. **Use Extra Information (e.g., Precedence Tables):** Instead of relying only on the grammar, we can use precedence rules to enforce meaning.

1. Operator Precedence Formalized via Grammar Structure

The key idea is that **operators with higher precedence must appear deeper in the parse tree**. This ensures that they are processed first in a bottom-up parsing approach.

Take our example grammar:

```
r Copia Modifica
```

```
E → E + T | T
T → T * F | F
F → id
```

- The rule `T → T * F` is **nested inside the `E` rule**.
- This means that **before an addition (`E → E + T`) can be applied, all multiplications (`T → T * F`) must have been fully reduced**.

This **forces** multiplication to be evaluated before addition because:

1. `T` must be completely parsed before returning control to `E`.
2. `T` itself cannot return to `E` until all its `T * F` structures are resolved.

Thus, **operators at deeper levels in the parse tree always get processed first**.

1. Left Recursion in Grammars

A **left-recursive** grammar rule is one where a **non-terminal appears as the leftmost symbol** in its own production.

Example of left recursion:

The screenshot shows a dark-themed text editor window. At the top, there are two buttons: 'Copia' (Copy) and 'Modifica' (Modify). The main area contains the following grammar rules:

```
E → E + T | T
T → T * F | F
F → id
```

- Here, $E \rightarrow E + T$ is **left-recursive** because E appears on the **left side** of the production rule.

2. What is Left Associativity?

Left associativity means that when multiple operators of the **same precedence** appear in a row, the operations are grouped **from left to right**.

For example, consider:

The screenshot shows a dark-themed text editor window. At the top, there are two buttons: 'Copia' (Copy) and 'Modifica' (Modify). The main area contains the following expression:

```
a + b + c
```

- With **left associativity**, it must be parsed as:

The screenshot shows a dark-themed text editor window. At the top, there are two buttons: 'Copia' (Copy) and 'Modifica' (Modify). The main area contains the following expression with parentheses:

```
(a + b) + c
```

- First, $a + b$ is computed.
- Then, the result is added to c .

With the left-recursive rule:

r

Copia Modifica

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

Step 1: Expanding E using left recursion

r

Copia Modifica

$E \rightarrow E + T$
 $E \rightarrow (E + T) + T$
 $E \rightarrow ((T + T) + T)$

Recursive Descent Parsers

- Top-down parsers build the parse tree from the top down as they read in input.
- *Recursive-descent* parser is structured as a set of potentially mutually recursive functions.

Simplest example of parser is Recursive-descent parser. We construct a procedure associated to a production rule.

1. What is a Top-Down Parser?

A **top-down parser** constructs the **parse tree** starting from the **root** (the start symbol) and **works its way down to the leaves** (the input tokens).

- It starts with the **start symbol** of the grammar.
- It tries to expand it using **production rules**.
- It continues until it matches the **input tokens**.

This is called **top-down parsing** because it begins with the highest-level structure and **breaks it down step by step**.

2. What is a Recursive Descent Parser?

A **recursive descent parser** is a type of **top-down parser** that uses a **set of recursive functions**, where each function corresponds to a **non-terminal** in the grammar.

- When a **non-terminal** needs to be expanded, the parser **calls a function** to handle it.
- If a **function calls itself** (directly or indirectly), that's **recursion**.
- Multiple functions can **call each other**, which is why your slide says they can be **mutually recursive**.

3. Example of a Recursive Descent Parser

Consider this simple grammar for arithmetic expressions:

```
r  
E → T + E | T  
T → id
```

A **recursive descent parser** would have:

- A function `parseE()` for `E`
- A function `parseT()` for `T`

C-like Pseudocode

```
c Copia Modifica

void parseE() {
    parseT(); // Parse the first term (T)
    if (nextToken == '+') {
        match('+');
        parseE(); // Recursively parse the rest (E)
    }
}

void parseT() {
    match("id"); // Match an identifier
}
```

Here's what happens when parsing "id + id + id":

```
SCSS Copia Modifica

parseE()
├─ parseT() → matches "id"
├─ match('+')
└─ parseE()
    ├─ parseT() → matches "id"
    ├─ match('+')
    └─ parseE()
        ├─ parseT() → matches "id"
```

This naturally **follows the recursive structure of the grammar!**



Recursive Descent Parsers

The S-expression grammar

$$S \rightarrow x \mid (L)$$

$$L \rightarrow \epsilon \mid S L$$

S-expressions were invented by John McCarthy for the programming language Lisp

The S-expression

(let ((x 2)) (+ x 1))

is roughly equivalent to the OCaml expression

let x = 2 in x + 1.

1. What is an S-Expression?

An **S-expression** (Symbolic Expression) is a simple, nested list-like notation used in **Lisp** and other functional programming languages. It represents both **code** and **data** in a uniform way.

- **Atoms** (like `x`, numbers, etc.) are basic elements.
- **Lists** are enclosed in parentheses `(...)` and can contain **other lists or atoms**.

2. What Does the Given Grammar Mean?

The slide provides a **context-free grammar** for defining S-expressions:

nginx Copia Modifica

```
S → x | (L)
L → ε | S L
```

- **S** (an S-expression) can be:
 - An **atomic symbol** (`x`).
 - A **list** (`(L)`), which contains an **L** inside parentheses.
- **L** (a list) can be:
 - **Empty** (`ε`, meaning an empty list `()`).
 - A **sequence of one or more S -expressions**.

Expression: `(let ((y (+ x 1))))`

1. Start with the top-level expression:

The whole expression is wrapped in parentheses, so this fits the rule `S → (L)`, where **L** is a list of expressions.

So, we have:

- `S → (L)`

Now, we need to parse the contents of the list **L**, which is the part after the opening parenthesis and before the closing parenthesis: `(let ((y (+ x 1)))`.

2. Recognize the first element in **L** (the `let` keyword):

In this case, the first element is `let`. The `let` keyword is treated as an **S-expression**—a part of the list **L**.

So, the `let` itself is just another part of the list that the parser will process. The grammar doesn't specify exactly how `let` works, but we still recognize it as an atom within the list.

So, we now have:

- $S \rightarrow (S \ L)$ where $S = \text{let}$ and $L = ((y \ (+ \ x \ 1)))$.

3. Next, parse the inner list $((y \ (+ \ x \ 1)))$:

The `let` keyword in Lisp typically introduces a list of bindings (variable-value pairs). Here, the binding is $((y \ (+ \ x \ 1)))$.

The inner part $((y \ (+ \ x \ 1)))$ fits the rule $L \rightarrow S \ L$ (where each item in the list is an S expression, and the next item is another L).

Breaking it down:

- First element $S = (y \ (+ \ x \ 1))$
 - This is another S-expression (a list), so it fits the rule $S \rightarrow (L)$.

We now need to parse this list L (which is $(y \ (+ \ x \ 1))$):

- $L = (y \ (+ \ x \ 1))$, and y is a variable, so it's an S (an atom).
- $(+ \ x \ 1)$ is another S-expression, a list. It fits $S \rightarrow (L)$.

We have a sequence of tokens. Peek takes the next token and provides the type of the token.

Recursive descent parser for S-expressions

```
void parseS() throws SyntaxError {  
    switch (peek()) { // peek reports the type of the next token  
        // the lexer is ready to return,  
        case ID: // S → x  
            scan(ID); // Whatever x is, it can be an identifier.  
            return;  
        case LPAREN: // S → ( L )  
            scan(LPAREN); // left parenthesis  
            parseL();  
            scan(RPAREN); // right parenthesis  
            return;  
    }  
    throw syntaxError(peek()); }  
}
```

Recurrsively you match the entire program.
If you can't match, there is a syntax error, because you are not following grammar.

Recursive descent parser for S-expressions

```
void parseL() throws SyntaxError {
    switch (peek()) {
        case SEQ: // L → S L
            parseS();
            parseL();
            return;
        case EPS: // L → ε
            return;
    }
    throw syntaxError(peek());
}
```

Summary

- Recursive descent parsing is simple, but it doesn't work for all grammars, and for more complex grammars, predicting the correct production involves some subtleties.
- More on a course on compiler construction!!



PARSING

Note on parsing
are available on
TEAMS

READINGS
section

What next?

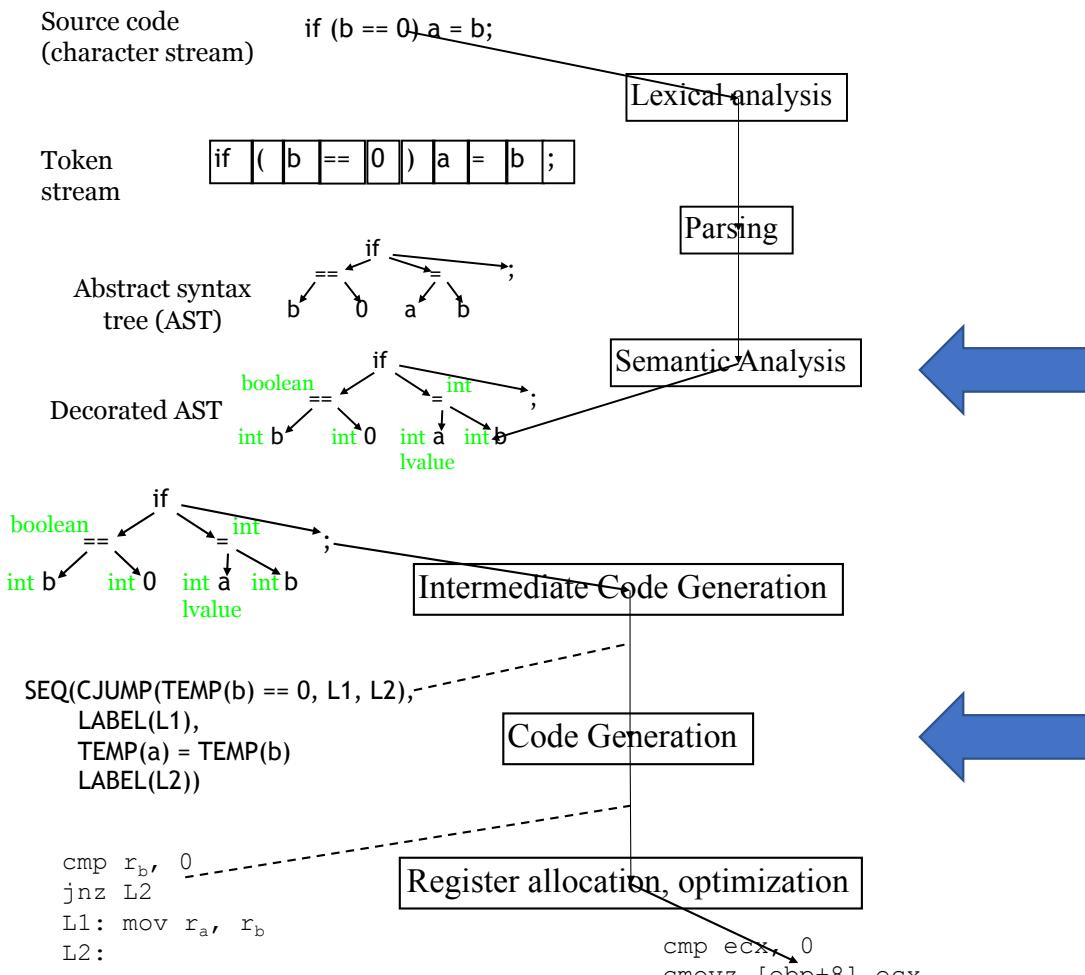
This is not a course
on compiler
construction

Outlining the
features of lexing
and parsing is
enough!!!

What Comes Next After Parsing



FRONT END DOES LEX. ANALYSIS AND PARSING.
BACK END SEMANTIC ANALYSIS AND CODE GEN.



Types and type checking
Control Flow Integrity
Information flow analysis
Taint Analysis
Abstract Interpretation

Run-time structure organization
Trusted Execution Environments

Here compiler has to ensure that the code generated works according to the trusted execution environment.

Abstract Syntax Trees



Abstract syntax trees are particularly relevant because they show the meaningful semantic structure of programs



From now on we will always consider abstract syntax, without considering the detailed aspects such as operator precedence, ambiguity, etc.

Abstract Syntax Trees in Java



Abstract syntax trees in Java

- We could encode the abstract syntax of a language in Java.
- How?
 - each syntactic category is an abstract class
 - each syntactic constructor is a subclass that extends the abstract class

1. "Each syntactic category is an abstract class":

- In the context of an AST, the **syntactic categories** are the different kinds of constructs you can have in your programming language (e.g., expressions, statements, variables, operators, etc.).
- **Abstract class** is used to define a common base for all of these categories. It's called abstract because it provides a blueprint, but you don't directly create instances of it. Instead, you define specific subclasses that represent concrete examples of that category.

Example:

- Suppose you have a language with `Expression` and `Statement` as two syntactic categories.
- You can create an **abstract class** `Expression` and another one for `Statement`.

java

Copia Modifica

```
abstract class Expression {  
    // Common methods and attributes for all expressions  
}  
  
abstract class Statement {  
    // Common methods and attributes for all statements  
}
```

2. "Each syntactic constructor is a subclass that extends the abstract class":

- Now, each specific type of expression or statement (e.g., a `Literal`, `Addition`, `Variable`, or `IfStatement`) is modeled as a **subclass** that extends the abstract class. These subclasses provide the **concrete details** for the specific constructs in the language.
- For example, a `Literal` or an `Addition` expression would be specific subclasses of the abstract `Expression` class.

Example:

- For the `Expression` category, you might have a `Literal` for numbers or variables and an `Addition` for adding two expressions.

```
class Literal extends Expression {  
    int value;  
    Literal(int value) {  
        this.value = value;  
    }  
}  
  
class Addition extends Expression {  
    Expression left, right;  
    Addition(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

In this case:

- `Literal` is a subclass of `Expression` that represents a literal value (e.g., a number).
- `Addition` is a subclass of `Expression` that represents an addition operation, holding two `Expression` objects (the left and right operands).

3. Why is this useful?

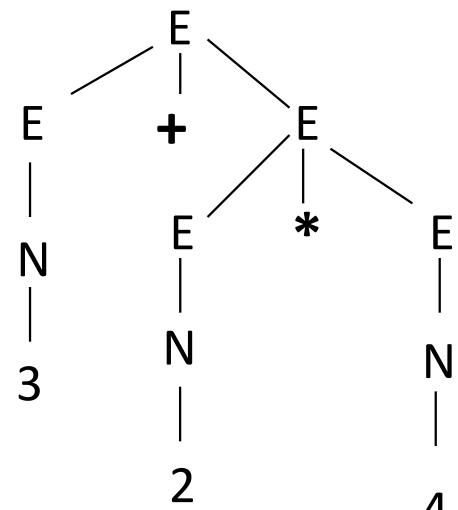
- This kind of design helps to structure the abstract syntax tree for a language and gives a clear, object-oriented way to represent different language constructs.
- By using classes and inheritance, you can encapsulate behavior that is common to all expressions (like evaluating an expression) in the abstract class, and then let the subclasses define their own behavior (like evaluating an addition operation or a literal).

Example

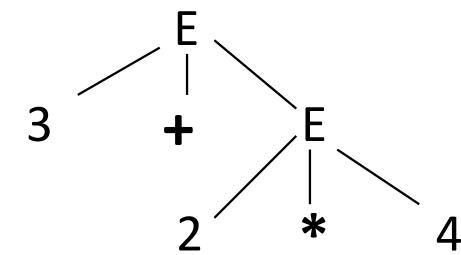
 The language of expressions

 Exp ::= Number | Exp + Exp | Exp * Exp

 Number ::= 0 | 1 | 2



PARSE TREE



ABSTRACT SYNTAX TREE

Expression AST in Java

```
public interface Expression {  
    public int eval(); // eval is the method to evaluate the expression  
                      // aka the interpreter of expressions  
}
```

Expression AST in Java

```
public class Add implements Expression{  
  
    private final Expression leftExpression;  
    private final Expression rightExpression;  
  
    public Add(Expression leftExpression, Expression rightExpression) {  
        this.leftExpression = leftExpression;  
        this.rightExpression = rightExpression;  
    }  
    @Override  
    public int eval() {  
        return leftExpression.eval() + rightExpression.eval();  
    }  
}
```

↓ eval is evaluate the left exp and add it to the right

Expression AST in Java

```
public class Product implements Expression{  
  
    private final Expression leftExpression;  
    private final Expression rightExpression;  
  
    public Product(Expression leftExpression, Expression rightExpression) {  
        this.leftExpression = leftExpression;  
        this.rightExpression = rightExpression;  
    }  
    @Override  
    public int eval() {  
        return leftExpression.eval() * rightExpression.eval();  
    }  
}
```

Expression AST in Java

```
public class Number implements Expression{  
  
    private final int n;  
  
    public Number(int n){  
        this.n = n;  
    }  
    @Override  
    public int eval() {  
        return n;  
    }  
}
```

AST in C

```
typedef struct AST AST;

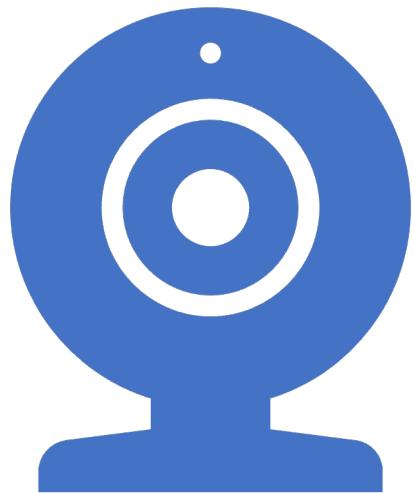
struct AST {
    enum {
        AST_NUMBER,
        AST_ADD,
        AST_MUL,
    } tag;
    union {
        struct AST_NUMBER { int number; } AST_NUMBER;
        struct AST_ADD { AST *left; AST *right; } AST_ADD;
        struct AST_MUL { AST *left; AST *right; } AST_MUL;
    } data;
};
```

C example

4 + 2 * 10 + 3 * (5 + 1)

```
AST *term =  
    AST_NEW(AST_ADD,  
            AST_NEW(AST_NUMBER, 4),  
            AST_NEW(AST_ADD,  
                    AST_NEW(AST_MUL,  
                            AST_NEW(AST_NUMBER, 2),  
                            AST_NEW(AST_NUMBER, 10),  
                            ),  
                    AST_NEW(AST_MUL,  
                            AST_NEW(AST_NUMBER, 3),  
                            AST_NEW(AST_ADD,  
                                    AST_NEW(AST_NUMBER, 5),  
                                    AST_NEW(AST_NUMBER, 1),  
                                    ),  
                            ),  
                    ),  
                );
```

```
void ast_print(AST *ptr) {
    AST ast = *ptr;
    switch (ast.tag) {
        case AST_NUMBER: {
            struct AST_NUMBER data = ast.data.AST_NUMBER;
            printf("%d", data.number);
            return;
        }
        case AST_ADD: {
            struct AST_ADD data = ast.data.AST_ADD;
            printf("(");
            ast_print(data.left);
            printf(" + ");
            ast_print(data.right);
            printf(")");
            return;
        }
        case AST_MUL: {
            struct AST_MUL data = ast.data.AST_MUL;
            printf("(");
            ast_print(data.left);
            printf(" * ");
            ast_print(data.right);
            printf(")");
            return;
        }
    }
}
```



Abstract Syntax Trees in OCaml

Abstract syntax trees in OCAML

- The abstract syntax of a language is easily expressed with Ocaml algebraic data types
 - each syntactic category becomes an Ocaml algebraic data type

BNF

Exp =

| n
| Exp + Exp
| Exp * exp

Algebraic Data Type

Type exp =

| Number of int
| Add of exp * exp
| Product of exp * exp

1. Abstract Syntax in OCaml

- OCaml is a functional programming language that supports **algebraic data types** (also known as **variant types**).
- Algebraic data types are a powerful way to define types that can have different variants, and these variants can hold data. They're great for representing tree-like structures, such as **abstract syntax trees**.

2. BNF (Backus-Naur Form)

You start by defining a **grammar** using **BNF** notation, which is often used to describe the syntax of a programming language.

In the BNF example:

```
plaintext Copiar Modificar
Exp =
| n           // A number
| Exp + Exp  // An addition expression
| Exp * Exp  // A multiplication expression
```

This says that an expression (**Exp**) can be:

- A **number** (`n`),
- An **addition** of two expressions (`Exp + Exp`),
- A **multiplication** of two expressions (`Exp * Exp`).

This is a very basic arithmetic grammar where we have numbers, addition, and multiplication.

3. Algebraic Data Type in OCaml

Now, we translate the **BNF** grammar into an **algebraic data type** (ADT) in OCaml.

```
ocaml Copiar Modificar
type exp =
| Number of int
| Add of exp * exp
| Product of exp * exp
```

Let's break this down:

- `type exp` defines a new type called `exp`. This is the type we'll use to represent expressions.

- The three variants are:
 - `Number of int`: This represents a **number**. It takes an `int` (integer) value, and it corresponds to the `n` rule in the BNF grammar.
 - `Add of exp * exp`: This represents an **addition expression**. It takes two `exp` values (the left and right operands), corresponding to the `Exp + Exp` rule in BNF.
 - `Product of exp * exp`: This represents a **multiplication expression**. Similar to `Add`, it takes two `exp` values (the left and right operands), corresponding to the `Exp * Exp` rule in BNF.

4. How Does This Relate to Abstract Syntax Trees?

In the context of abstract syntax trees, we're defining the structure of the expressions.

For example:

- A `Number` node in the tree will contain an integer value (like `3` or `5`).
- An `Add` node in the tree will have two child nodes: one for the left operand and one for the right operand.
- A `Product` node will also have two child nodes representing the left and right operands.

So, when you create an abstract syntax tree for the expression `3 + 5 * 2`, you would have a structure like this:

```

mathematica
Copia Modifica

      Add
      /   \
Number  Product
|       /   \
3      Number  Number
|           |
5      2
  
```

5. Example of Using This in OCaml

Let's say you want to build the abstract syntax tree for the expression `3 + 5 * 2` in OCaml. You could write something like this:

```
ocaml          Copia  Modifica

type exp =
| Number of int
| Add of exp * exp
| Product of exp * exp

(* Build the abstract syntax tree for 3 + 5 * 2 *)
let expr = Add (Number 3, Product (Number 5, Number 2))
```

- Here, `expr` represents the abstract syntax tree where `3 + 5 * 2` is encoded.
- The `expr` tree corresponds to the structure I showed earlier, with `Add` at the top, having `Number 3` as its left child and `Product (Number 5, Number 2)` as its right child.

6. Summary

- Abstract syntax trees in OCaml are easily represented using algebraic data types.
- Each syntactic category in your grammar (like `Exp` for expressions) becomes an OCaml algebraic data type.
- The BNF grammar defines the structure of expressions.
- OCaml's algebraic data types provide a way to define this structure in the code.
- The abstract syntax tree represents the program's structure, where each node in the tree corresponds to a part of the grammar.

In simple terms, the slide is showing how to take a grammar and represent it in OCaml using algebraic data types. This allows you to build, manipulate, and evaluate programs represented as abstract syntax trees.



Suggestion

A crash course on OCAML

<https://cs3110.github.io/textbook/cover.html>