CS 315 – Programming Languages
Introduction

Why do we use Programming Languages (PLs)

- To tell the computer what to do
- To *instruct* the computer what to do
- *instructions*

Forms of PLs

- binary (native to the machine) 101001 . . . 101 → generally powers of 2 times.
- assembly (assembler converts assembly to binary) add $s0, $s1, $2
- high-level (compiler converts high-level to assembly) int a = b+c;

Why do we have so many different PLs?

- Evolution: assembler
  → structured → C, Pascal
  → object-oriented → C++, Java, Swift
  → functional → Haskell
  The need for new abstractions arise as new problems are tackled.
- Different languages make different trade-offs and are thus suitable for different tasks
  - C for low level programming
  - Java for enterprise programming
- Special purpose: domain-specific langauges
  SQL (database), XML (semi-structured data), XSLT (XML transformations), Verilog (Hardware design), etc.
- Personal preference: like fashion → not always! You cannot code a kernel with Javascript

What are PL evaluation criteria?

- Expressive power: How easy is to to express ideas
  - Object-Oriented: models real world
  - C pointers: maps to hardware's memory model
- Ease of use for the novice
  - C++: difficult; Pascal: easy
- Efficiency of implementation
  - Python: slow; Java: fast; C: faster
- Open source availability
  - A critical aspect for a large number of open source projects
  - License type is important as well (Copyleft GNU vs BSD clang)
- Standardization
  - Lack of it causes interoperability issues
  - What this covers is important
    - e.g., C++ name mangling is not standardized, and thus causes interoperability issues
- Economics, Patronage, Inertia
  - Fortran: IBM, ADA: DoD, Java: Sun/Oracle, C#: MS

Dynamically Typed: In Python, you never declare anything. An assignment statement binds a name to an object, and the object can be of any type. If a name is assigned to an object of one type, it may later be assigned to an object of a different type.

Statically typed: In Java, all variable names (along with their types) must be explicitly declared. Attempting to assign an object of the wrong type to a variable name triggers a type exception.

Important PL evaluation metrics

- Readability
- Writability
- Orthogonality
- Performance
- Reliability

The first two (readability and writability) often support each other, but may also be at odds:

- E.g.: Java is high on readability (verbose syntax provides hints on semantics), which makes it low on writability (too verbose, takes time)
- E.g.: Perl is very high on writability (various shortcuts and implicit variables help with getting a lot done via small amount of code), but very low readability (sometimes jokingly referred to as a write-once language)
- E.g.: Having too many core features in a language makes the language high on writability (developers can use the features they are most familiar with), but low on readability (readers need to know all features)

→ Fall 2013 midterm question.

Orthogonality: A PL is orthogonal if its features can be composed without special cases.

Almost no language is purely orthogonal. For instance, in Java Generics cannot be combined with primitive types (e.g., can't have ArrayList<int>).

The last two (reliability and performance) are often at odds as well:

- Improved reliability comes at the cost of additional checks. Sometimes these checks needs to happen at runtime, which brings additional cost. E.g.: Array bounds checking in Java. (or null checks to throw a nullptr exception)

Performance typically refers to running time performance, that is, the time it takes to execute the program. For compiled languages, compilation time performance might be of interest as well, as it impacts how fast the edit-debug cycle is. Performance may also refer to memory usage.

Reliability is about the error checking facilities present in the language. These could be runtime or compile-time error checks. Some examples follow:

- Type checking

Python: Flexible due to missing type checks, but more error prone due to possibility of run-time errors. It is important to note that the runtime errors happen when the program is run and only if the problematic code line is exercised with the right set of values.

```
def sum(x, y):
    return x+y
```

```
sum(5, "15")
# gives run-time error
```

Java: Safe, but not flexible.

```
int sum(int x, int y)
{    return x+y;    }
```

```
sum(5, "15")
# gives compile-time error
```

C++: Flexible usage could be provided at the cost of reability and writability for the function definition.

`template<typename T>`
`T sum(T const & x, T const & y)`
`{ return x+y; }`
	`sum(5, "15")`
	`# gives compile-time error`

C++14: What if we want to add things of different types? This should be valid as long as there is a + operator defined for them. The above sum function for C++ cannot do this, yet the one for Python can. Below is a C++14 function that can achieve flexibility similar to the Python variant, but with compile-time checks rather than run-time checks.

two different types are defined.

```
template<typename T1, typename T2>
auto sum(T1 const & x, T2 const & y)
{    return x+y;    }
```

- Exceptions: Another reliability feature. Compare C-style error checking via return codes versus Java/C++ style exception handling. Exceptions simplify handling of error cases. We will study this topic at length later.

- Aliasing: Two distinct names pointing to the same object/memory. This is bad for reliability, as it makes it difficult to track changes on the objects. Most Von Neuman languages provide this, as it provides increase performance (avoids copying).

- Run-time checks
  - Error bounds (Java arrays vs C arrays)
  - Null pointer checks (Java null pointer exceptions vs C segmentation faults (or overwring your own memory))

- Garbage collection: avoid memory leaks and dangling pointers.

Spectrum of Pls

- imperative – the how
  - Von Neuman (C, Ada, Fortran)
    - Scripting (Perl, Python, Ruby)
    - Object Oriented (C++, Java)
- declarative – the what
  - Logic (Prolog)
  - Functional (Lisp/Scheme, Haskell)
  - Data flow (id, val)
  - Template (XSLT)

There are four main variations:

- *Von Neuman*: Computation through modifications of variables that correspond to memory
  - *Scripting*: rapid prototyping and expressivity over speed of execution
  - *Object-oriented*: interactions among objects (state and behavior)
- *Functional*: Recursive definition of pure functions
- *Logic/constraint-based*: Find values that satisfy certain relationships/rules
- *Data flow*: Data flows across nodes forming a processing graph

Example from the three language variations: Let us compute the GCD (greatest common diviser)

Von Neuman / C ***assignments (= operator)***

```
int gcd(int a, int b)
{
  while (a!=b)
    if (a>b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Functional / Scheme ***recursive function definitions***

```
(define gcd
  (lambda (a b)
     (cond ((= a b) a)
           ((> a b) (gcd (- a b) b))
           (else (gcd (- b a) a))
)))
```

Logic / Prolog ***rules***

```
gcd(A,B,G) :- A=B, G=A
gcd(A,B,G) :- A>B, C is A-B, gcd(C,B,G)
gcd(A,B,G) :- B>A, C is B-A, gcd(A,C,G)
```

Why do we study PLs?

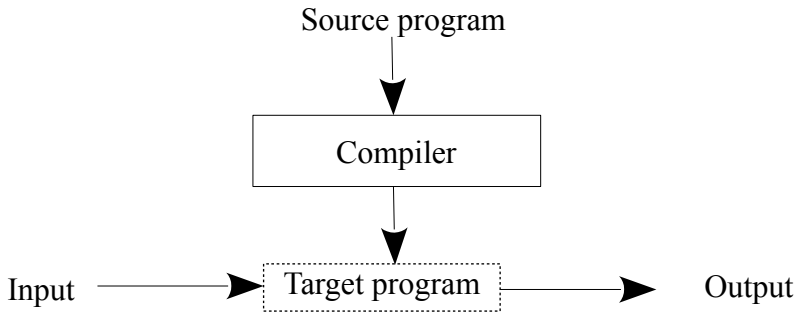
- Choose the most appropriate language for the task
  - Building an OS? Performance and low-level abstractions, e.g., C or C++
  - Enterprise applications? Readability, novice friendly, e.g., Java, C#
  - Scientific applications? Matrix manipulations, symbolic execution, e.g., Matlab, Python
- Learn new languages easily
- Simulate useful features in languages that lack them
  - OO-style programming in C via function pointers as struct members
  - naming conventions to emulate namespaces in C
- Design & implement your own language
  - General steps: parse, analyze, optimize, generate
  - Does not need to be a general purpose language
    - Could be a DSL (domain-specific language)
    - E.g., a language for specifying rules, configurations, etc.

Compilation & Interpretation

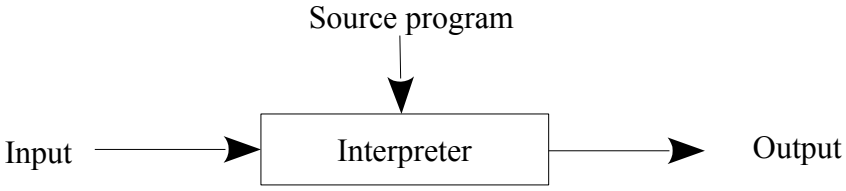
We will look at three different modes:

- Pure compilation
- Pure interpretation
- Mixed compilation/interpretation

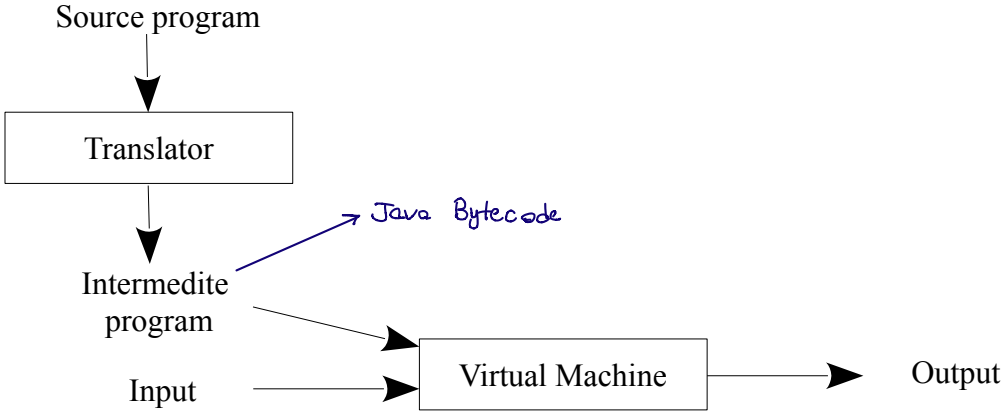
Pure compilation



Pure interpretation



Mixed compilation/interpretation



Pure compilation (e.g., C, C++)

- Advantages:
  - Good runtime performance: Compiles down to machine code, which could be specific to the instruction set of the machine at hand. No interpretation involved during runtime.
- Disadvantages:
  - Longer edit-debug cycle due to full compilation needed. → modern compilers/linkers take care of that actually.
  - Software distribution is harder due to platform dependence. You need separate versions of the program distributed for different hardware architectures. → one day we will ditch the x86 architecture it is going to be so much painful.

Pure interpretation (e.g., Python with CPython)

- Advantages:
  - Short edit-debug cycle: no compilation step is needed.
  - Platform independence simplifies software distribution, as there will be a single version of the program distributed for all platforms. However, there should be an interpreter available for each platform supported. Given that interpreter is not provided by the software developer, this is not a burden, unless availability of interpreters is a problem.
- Disadvantages:
  - Low runtime performance
  - Have to distribute source code

Mixed compilation/interpretation (e.g., Java, Python with PyPy)

- Advantages:
  - Platform independence similar to pure interpretation
  - Compilation times are often faster than pure compilation, resulting in better edit-debug cycles, but still not as good as pure interpretation.
  - Performance can get closer to pure compilation via the use of JIT
    - JIT: Just-in-time compilation: run-time compilation of intermediate representation of the code down to machine code. This is often done in an on-demand fashion to avoid a lengthy pause at start-up. In other words, parts of the program will be compiled as needed, and reused when the same code is executed later.
- Disadvantages:
  - Most of the work is pushed to the runtime virtual machine. This can result in pauses during runtime, making it not suitable for real-time applications.
  - Making it perform close to pure compilation requires a very large development effort.
- E.g., Java has bytecode as its intermediate code, and JVM as its virtual machine. Most JVM implementations provide a JIT compiler embedded into the JVM.

Boostrapping

Assume we are designing a new language called Z. We want to make your language *self-hosting*, that is, the compiler for the Z language should be implemented in the Z language itself. Doing this often requires a cumbersome translation step, where the first Z compiler is written in a language for which a compiler already exists (e.g., C), and then it is rewritten in the Z language and compiled with the first Z compiler. The rewriting of an entire compiler is a bit of a pain, so one approach to cope with this is to do the following: You divide the Z language into two: Z0 and Z1. Z0 is a simple version of Z that corresponds roughly to a subset of the C language that we name C0.

1. You first write a compiler for Z0 in C0 and compile it with a C compiler to obtain the first binary Z compiler.
2. You convert the source code of the first Z compiler from C0 to Z0 and compile it with the first binary Z compiler from the first step. You obtain the second binary Z compiler.
3. You expand the source code of the Z compiler, still using Z0 features, to recognize the features of the Z1 language and compile the resulting code with the second binary Z compiler from the previous step. You obtain the third Z compiler binary.
4. You update the source code of the Z compiler, this time to make the code more maintainable by using some of the Z1 features that will facilitate that. You compile the resulting compiler code using the third Z compiler binary from earlier and obtain the fourth Z compiler binary.
5. Finally, you recompile the Z compiler source code with the fourth Z compiler binary to obtain the fifth Z compiler binary.

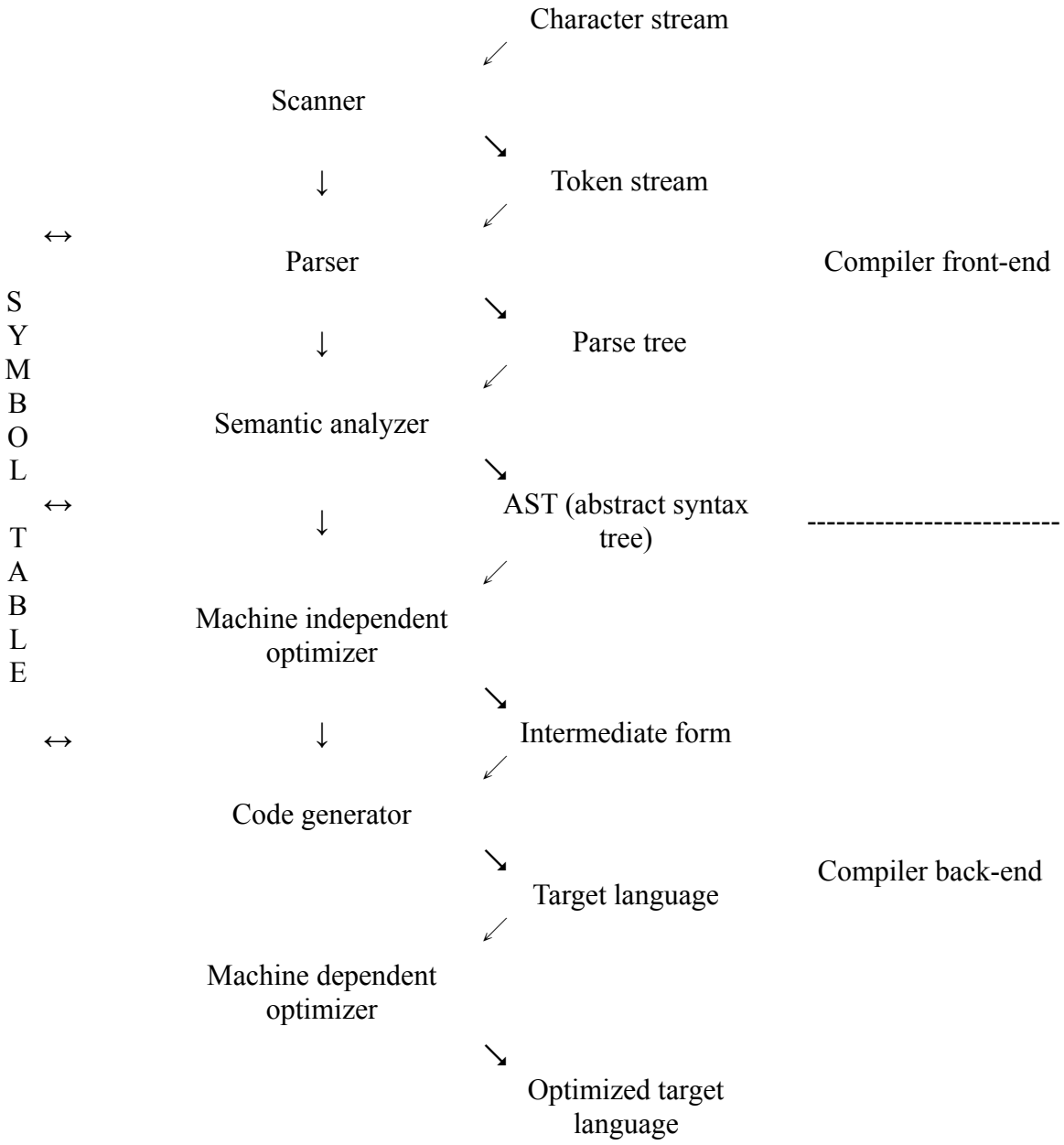
Let $B_x^{y(z)}$ represent a compiler binary that

- can recognize/compile language x
- was implemented in language y
- was compiled with a compiler that was implemented in language z

Assume that initially we have a second-order self-hosting C compiler at hand, that is $B_C^{C(C)}$. The following table details the steps outlined above to reach $B_{Z1}^{Z1(Z1)}$.

Step	Language of the Z Compiler source code	Development task performed	Language recognized by the Z Compiler	Compiler used to compile the Z Compiler	Output binary
1	C0	Write C0 code for the compiler	Z0	$B_C^{C(C)}$	$B_{Z0}^{C0(C)}$
2	Z0	Translate the C0 code into Z0	Z0	$B_{Z0}^{C0(C)}$	$B_{Z0}^{Z0(C0)}$
3	Z0	Extend the compiler code to recognize Z1	Z1	$B_{Z0}^{Z0(C0)}$	$B_{Z1}^{Z0(Z0)}$
4	Z1	Update compiler code to use Z1	Z1	$B_{Z1}^{Z0(Z0)}$	$B_{Z1}^{Z1(Z0)}$
5	Z1	Nothing	Z1	$B_{Z1}^{Z1(Z0)}$	$B_{Z1}^{Z1(Z1)}$

An Overview of Compilation



Scanner (lexical analyzer): characters → tokens

This transformation is done based on regular expressions describing the tokens

Parser: Organizes tokens into a parse tree

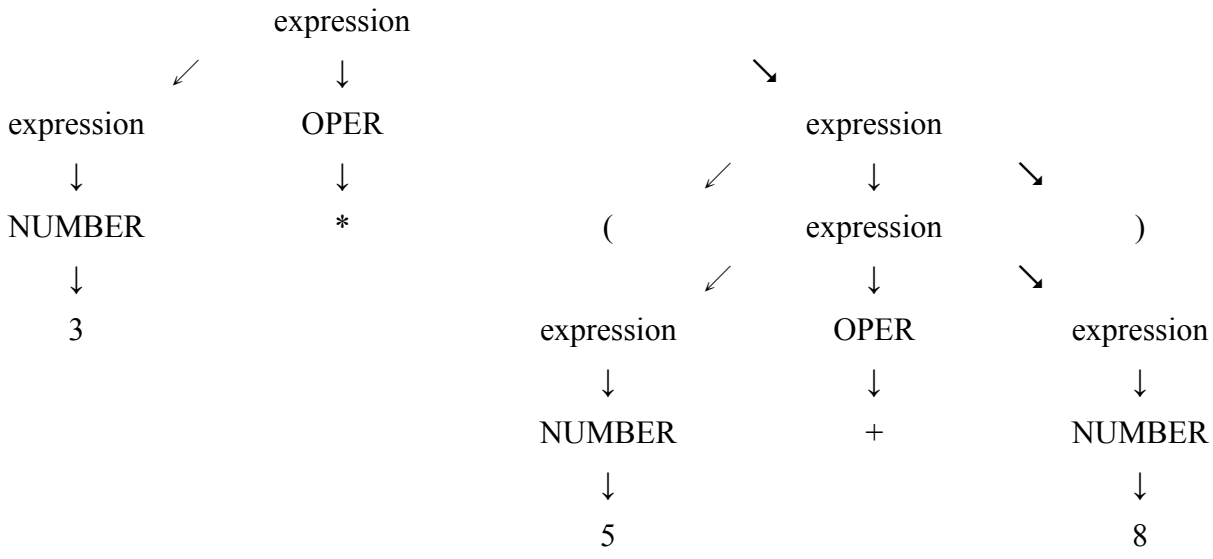
This transformation is done based on a grammar describing the language syntax

Example:

Grammar:
expression → expression OPER expression
| NUMBER
| '(' expression ')'

Tokens:
OPER: '+' | '-' | '*' | '/'
NUMBER: [0-9]+

3 * (5 + 8)
becomes



From now on, we will see a line of code as a tree and not as a sequence of characters.

Semantic analysis:
- Simplify tree
- Find types
- Find defs/refs (use symbol table)
- Check for correctness
  - e.g., every identifier is defined before it is used

These are static checks done at compile-time.

Code generation: Converts AST into target code (e.g., assembly)

Optimizations: These are used to improve the runtime performance of the code. Machine independent optimizations do not rely on the machine's instruction set. For instance, folding constants (converting 3+5 into 8), dead code elimination, or various loop optimizations are not dependent on the machine. Whereas some optimizations, like using SIMD instructions to speed up the code, are machine dependent.