CS 421 Lecture 4: Overview of language implementation

- Lecture outline
  - Static vs. dynamic languages
  - Program execution and run-time systems
  - Compiler structure
  - Some history

Change of pace

- No more OCaml! (* for now… *)
- Different ways to design and implement programming languages
- Brief history of PLs

Language implementation overview

- Language types
  - Static, vs.
  - Dynamic
- Implementation approaches
  - Compile to machine code, vs.
  - Compile to virtual machine code, vs.
  - Directly execute ("interpret")
- Run-time support
  - "Raw" machine, vs.
  - Extensive run-time support (*e.g.*, garbage collection)

Language types

- Static, a.k.a. “compiled,” a.k.a. “conventional”
  - Examples: C, C++, Fortran
  - Static type-checking
  - “Manual” memory management
  - Run-time values not “tagged” – *i.e.*, cannot determine type of value at run time
- Dynamic, a.k.a. “interpreted”
  - Examples: Java, OCaml, Python, Lisp
  - Often lack static type-checking (Python, Lisp), but sometimes have it (Java, OCaml)
  - Automatic memory management, a.k.a. garbage collection
  - Run-time values are “tagged” – *i.e.*, can determine properties of values at run time

Type checking – static vs. dynamic

- When is type-checking done?
  - Statically, *i.e.*, at compile time
  - Dynamically, *i.e.*, at run time. (Values must be tagged in some way.)
- How strong?
  - Strong: no type errors possible, *e.g.*, if program has expression "`x.a`", then `x` is *definitely* an object of a class that has a field named `a`.
  - Weak: programmer may bypass type system
- These are properties of the language, *i.e.*, specified in the language's definition.

Type checking (cont.)

- Java:

```
int f (int x) { return x+1; }
… f(new C()) …
```

- Ocaml:

```
let f x = x+1;;
… f true …
```

- C or C++:

```
int f (int x) { return x+1; }
… f((int)new C()) …
```

- Python:

```
Def f (x):
    return x+1
… f([]) …
```

- Note: Not all errors are *type* errors – *e.g.*, hd [], or 5/0. Call those *value errors*. In Java and Ocaml, no type errors can occur at run time; in Python, both value and type errors can occur; in C or C++, type errors cannot normally occur, but you can cause them by injudicious casting.

Automatic memory management

- Consider these programs:
  - C:

```
for (i = 0; i <= Max; i++)
   x = malloc(sizeof (float));
```

  - Java:

```
for (i = 0; i <= Max; i++)
   x = new C();
```

- Suppose Max is a very large number. What will happen?
- Automatic memory management, also called *garbage collection*.

Run-time tags

- Suppose you want to write a function `classOf(x)` that returns the name of `x`'s class, where `x` is a pointer to an object. It would be like this:
  - C++:

```
void f (void *x) { cout << classOf(x); }
```

  - Java:

```
void f (Object x) { println(classOf(x)); }
```

- Is it possible?
- In Java, can see not only the type of a variable, but the name and fields of its class, and other aspects of run-time state. This is called *reflection*.

What compilers do

- Compilers translate high-level language programs (C, C++, Java, Python, Ocaml, …) to an executable form.
  - Conventional: Translate to machine language; load and run.
  - "Dynamic:" Translate to "virtual," or "abstract," machine language; virtual machine emulator loads and executes virtual machine code.

Compiling to machine code

- Compiler knows the target machine code.
- Generates machine instructions, *e.g.*, C compiled for x86:

```c
int f (int x) {
   return x+1;
}
```

```
.globl f
    .type f, @function
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
```

- Execute directly on machine of correct type.

Compiling to a virtual machine

- Compiler translates to a made-up machine language for which no machine actually exists.
- Generates virtual (or abstract) machine instructions, *e.g.* Java:

```
int f (int x) {
   return x+1;
}
```

```
iload_1
iconst_1
iadd
ireturn
```

- A program reads that code and then executes it one instruction at a time ("emulates" the non-existent machine)

Interpreter

- Alternate implementation method: Don't translate the program at all. Execute the program by traversing its abstract syntax tree and executing each part. The program that does this is called an *interpreter*.
- Hardly ever used any more.
  - At least for general-purpose programming languages.

What method is best?

- In principle, either method can be used for any language.
- In practice, older languages (C, C++, Fortran) are usually compiled to machine language, while new ones (Java, OCaml, Python) use virtual machines.

Run-time systems

- Complete set of services available to running programs. Can range from raw machine to virtual machine:
  - "Raw" machine: Just O.S. services, *e.g.*, read/write files; allocate memory; spawn processes; *etc.*
  - Virtual machine: O.S. services, plus run-time type-checking; garbage collection; reflection

Executing C programs

- C programs are translated to machine language.
- Run on raw machine
  - No run-time type-checking – type errors can go undetected until they casue a machine-level problem, *e.g.*, null pointer dereference
  - No garbage collection, a.k.a. automatic memory management – memory allocated (malloc'd) is never available until it is expressly freed.

Executing Java programs

- javac translates Java programs to Java virtual machine (JVM) code
- JVM code executed by virtual machine (java)
  - VM knows types of all variables – run-time type checks
  - Garbage collection – no need to free memory
  - Reflection – can discover, *e.g.*, type class of an object, see what fields it has, *etc.*
- Many Java virtual machines translate JVM code to native machine code, either as soon as they are loaded or after they have executed for a while. This is called *just-in-time compilation*.

Executing OCaml programs

- Translated to virtual machine code
- Can compile programs into files, but normally programs are executed immediately
- Run-time system
  - G.C.
  - No run-time type checks

Executing Python programs

- Translated to virtual machine code
- Run-time system
  - G.C.
  - Run-time type checks

Language implementation overview (revisited)

- Language types
  - Static, vs.
  - Dynamic
- Implementation approaches
  - Compile to machine code, vs.
  - Compile to virtual machine code, vs.
  - Directly execute ("interpret")
- Run-time support
  - "Raw" machine, vs.
  - Extensive run-time support (*e.g.*, garbage collection)

Engineering trade-offs

- Different implementations present trade-offs between different values:
  - Fast response time
  - Fast execution time
  - Type-safety
  - Portability
  - Implementation complexity

- Desired features depend on the application domain: what is the language *for*?

History of languages – 1950s

- Late 1950s:

- FORTRAN
  - Not very high level
  - Compiler produced excellent code
  - No automatic memory management
  - No recursion
  - Static typing
  - "Compiled" language

- LISP
  - Fully-parenthesized syntax
  - Dynamically-allocated lists
  - Automatic memory management
  - Recursion
  - Dynamic typing
  - "Interpreted" language

History of languages – 1960s

- Compiled Languages
  - FORTRAN, PL/1, COBOL, ALGOL, PASCAL, SIMULA
  - Block structure
  - Recursion
  - No dynamic allocation
- Interpreted ("dynamic") languages:
  - LISP, APL, BASIC
  - Memory management
  - Run-time type checking

History of languages – 1970s

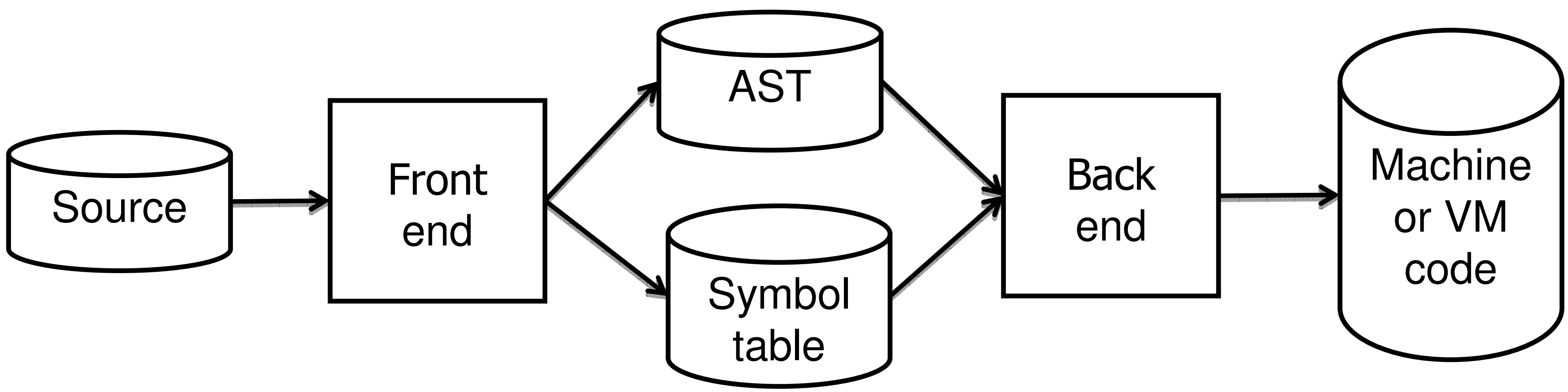
- Compiled languages:
  - C
  - OO languages:
    - Smalltalk – interpreted
    - CLU, ALPHARD, … – compiled
- Interpreted (“dynamic”) languages:
  - Scheme (variant of LISP), ML, PROLOG

History of languages – 1980s to present

- 1980s
  - C++ (compiled)
  - Objective C (compiled)
- 1990s
  - Java
  - Python, JavaScript, Perl
- 2000s
  - C/C++
  - Java/C#
  - Python, JavaScript, Ruby, …

Compilers

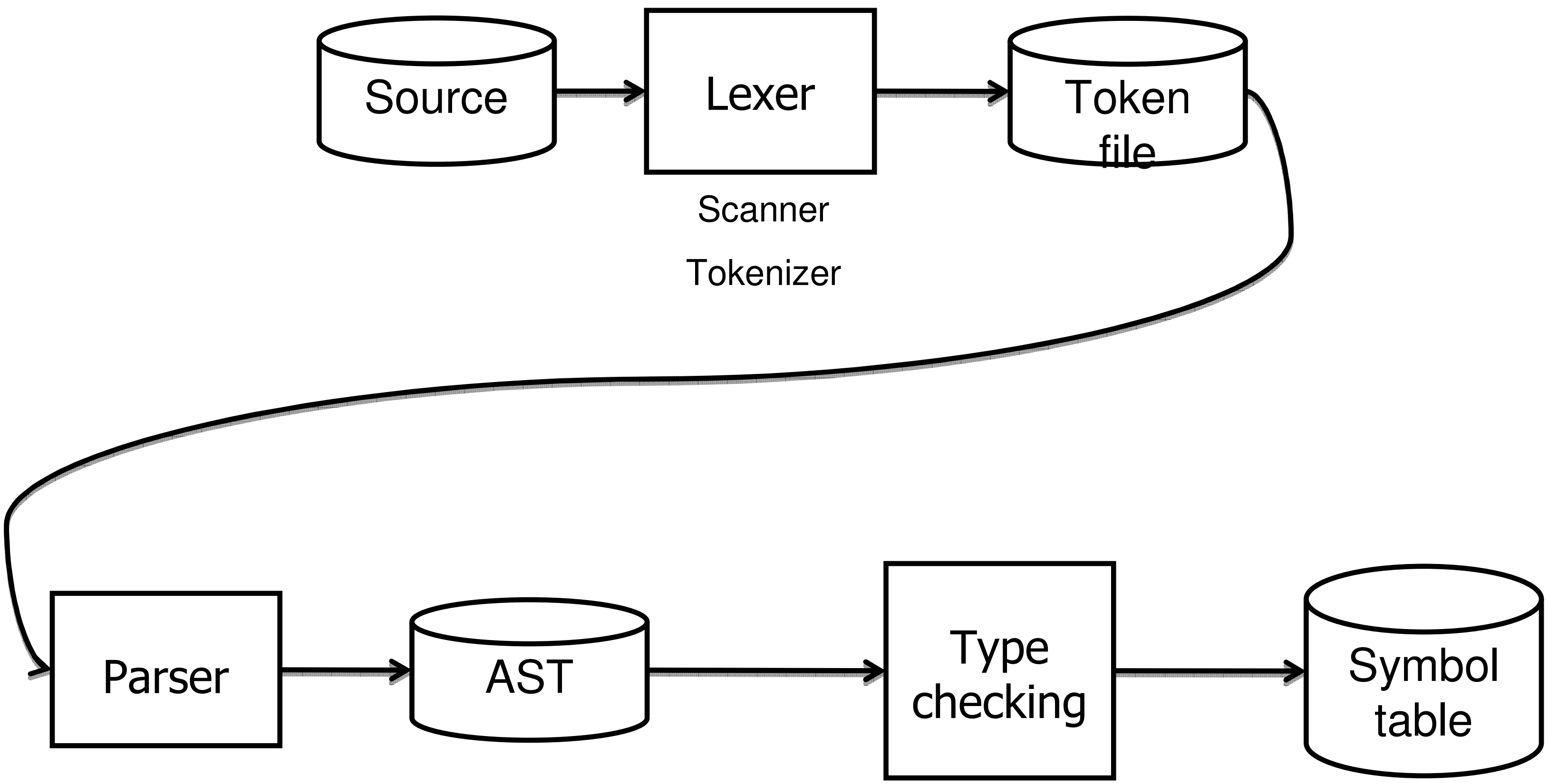
- Compiler structure



- Abstract syntax tree = tree representation of a program
- Symbol table = properties of names defined in a program
  - Type of variables
  - Argument types of functions
  - Etc.

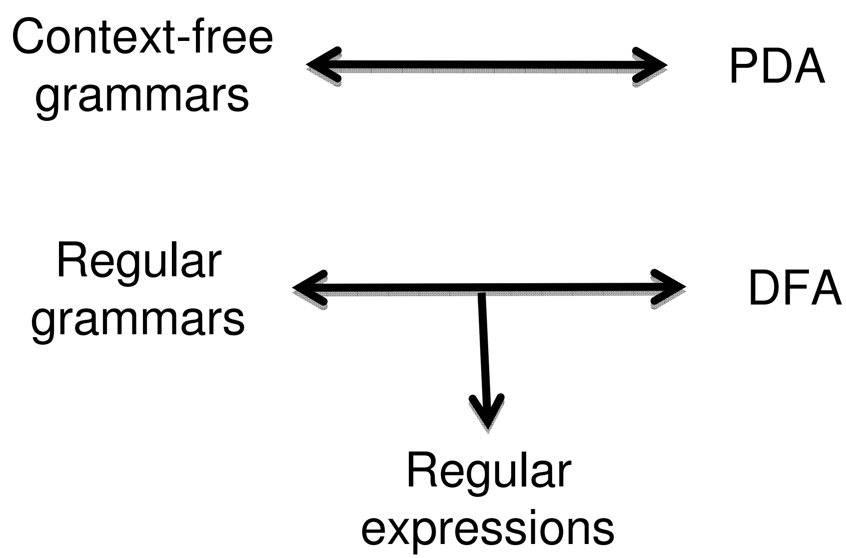
Compiler front end

- Front end divided into three phases:



History of front ends

- 1950s – lexing, parsing by *ad hoc* means
- Mid-50s – Chomsky hierarchy:



History of front ends (cont.)

- 1960s – Application of Chomsky hierarchy
  - CFGs for describing programming languages
  - Automatically obtain parser – "compiler-compilers", a.k.a. "parser generators"
  - Regular expressions for lexers
- 1970s – Knuth discovers LR(k) grammars
  - Large class of grammars that can be parsed efficiently – yacc

Summary

- Compiler front end analyzes program, produces AST and symbol table
- Compiler back end produces target machine code or virtual machine code
  - If machine code, program is executed directly, probably with minimal run-time support by O.S. services
  - If virtual machine code, program executed by emulator, probably with automatic memory management, possibly run-time type-checking, reflection