Class 4 - Overview of language implementation

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

Overview of today’s class

- 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, aka “compiled,” aka “conventional”
  - Examples: C, C++, Fortran
  - Static type-checking
  - “Manual” memory management
  - Run-time values not “tagged” — e.g. cannot determine type of value at run time
- Dynamic, aka “interpreted,”
  - Examples: Java, OCaml, Python, Lisp
  - Often lack static type-checking (Python, Lisp) (but sometimes have it: Java, OCaml)
  - Automatic memory management, aka garbage collection
  - Run-time values “tagged” — e.g. 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 used 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 the 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. (Or, dynamic languages are "interpreted" — loaded and executed without translating to an executable form; they may translate to such a form internally.)

Compiling to machine code

- Compiler knows machine it is compiling for.
- Generates machine instructions, e.g. C compiled for x86:

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

$\Rightarrow$

```
.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 virtual machine

- **Compiler translate 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) {              iload_1
   return x+1;       ⇒       iconst_1
}                            iadd
                             ireturn
```

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

Interpretation

- Alternate implementation method: Don't translate program at all. Execute program by traversing tree and executing each part. The program that does this is called an *interpreter*.
- Hardly ever used any more. (Languages that produce no executable files are often called "interpreted," but usually do actual compilation internally. It is sometimes possible to save this internal form in a file, e.g. Python "pyc" files.)

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

- Run-time system = 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 cause a machine-level problem, e.g. null dereference
  - No garbage collection, aka 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

Overview of today's class (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.

History of languages — 1950's

- **Late 1950's:**

FORTRAN
Not very high level
Compiler produced
excellent code
No automatic memory mgt
No recursion
Static typing
“Compiled” language

LISP
Fully-parenthesized syntax
Dynamically-allocated lists
Automatic memory mgt
Recursion
Dynamic typing
“Interpreted” language

History of languages — 1960's

- **Compiled languages:**

- **Interpreted (“dynamic”) languages:**

History of languages — 1970's

- **Compiled languages:**

- **Interpreted ("dynamic") languages:**

History of languages — 1980's-present

- **1980's**

- **1990's**

- **2000's**

Compilers

- Compiler structure:

- Abstract syntax tree = tree representation of program

- Symbol table = properties of names defined in program - type of variables; argument types of functions; etc.

Compiler front end

- **Front end divided into three phases:**

History of front ends

- **1950's — lexing, parsing by *ad hoc* means**
- **Mid-50's — Chomsky hierarchy**

History of front ends (cont.)

- **1960's — Application of Chomsky hierarchy**

- **1970's — Knuth discovers LR(k) grammars**

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