Interpreting Functions

- In the last lecture we talked about interpretation, and we had come to the problem of interpreting functions.
- Functions need an environment where they can store the values of arguments and local variables.

Environments in Symbol Tables

- One solution:
  - Use a symbol table to store the values of symbols.
  - Create a new local symbol table for the symbols in a function before each function call.
  - Use function calls in Java to implement function calls in our interpreted language.
- Problem:
  - Symbol information (such as the type of a symbol) has to be recreated at each call.

Static Vs Dynamic

- Information about a program can either be static or dynamic.
- Static information is computed during compile-time and does not change during runtime.
- Dynamic information is computed during runtime and might change during the execution.

Static Vs Dynamic (Values & Symbols)

- There are similarities between `Value` and `Symbol`; both have information about an identifier.
- `Symbol` store static information about the identifier, that is, information that does not change over time.
- `Value` stores dynamic information about the identifier, an identifier has different values over time.

Static Vs Dynamic (Compile-time)

- Scopes are static. (It is fixed during interpretation.)
- The abstract syntax tree contains static information.
- Most compiler passes works recursively on the abstract syntax tree and computes static information.

Static Vs Dynamic (Runtime)

- Environments are dynamic. (For each call to a function we create a new environment.)
- The interpreter computes dynamic information.
- The interpreter is not a simple recursion on the abstract syntax tree. In a function call it jumps from the call to its definition to execute the function body.

Static Vs Dynamic (Type Checking)

- For statically typed languages (Java, C, ML) type checking is done statically as part of the analysis.
- For dynamically typed languages (Erlang, Scheme), type checking is done dynamically, at runtime.
- (Some LISP interpreters also did name analysis at runtime – giving you dynamic scoping.)

Environments

- We have both static and dynamic information about identifiers.
- We want two different structures to keep track of the static and the dynamic information, a symbol table and an environment.
- The environment can be implemented as an array of values, the position in the array of an identifier can be stored in the symbol table.

Extending Symbol

```
class Symbol {
  …
  abstract Value load(Environment env);
  abstract void store(Environment env,
                      Value val);
}
```


Local Variables

```
class LocalSymbol extends Symbol {
   int offset;
   Value load(Environment env){
     return env.values[offset];
   }

   void store(Environment env,Value val){
     env.values[offset] = val;
   }
}
```


Global Variables

```
class GlobalSymbol extends Symbol {
   Value val;
   Value load(Environment env){
     return val;
   }

   void store(Environment env,Value val){
     this.val = val;
   }
}
```


Identifiers, version 2

Now we can get the value of an identifier in the interpreter:

```
…
void caseIdent(Tree.Ident tree) {
  // Try to find the symbol.
  Symbol sym = lookup(tree.name);
  if (sym == null) { // Not found.
      error();
  } else {
    result = tree.symbol.load(currentEnv);
  }
}
```

Function Calls

```
void caseFunCall {
  // call interpreter recursively on
  //   function arguments;
  Arguments args = interpret_args(tree.args);

  // Create a new Environment
  currentEnv = new Environment(currentEnv);

  // Store the arguments in the new environment.
  insert_args(args, currentEnv);

  // Call the interpreter recursively on the
  //   body of the called function, using the new
  //   environment.
  result = interpret(find_code(tree.funName));

  // Restore the environment.
  currentEnv = currentEnv.outer;
}
```


Summary Interpreter

- In general the interpreter can be implemented more or less as the Evaluator visitor, but control flow, and especially function calls complicates things a bit.
- We have to keep track of both static and dynamic information about identifiers.

Virtual Machines

- A virtual machine is a simple way to get a portable compiler.
- A virtual machine often resembles real physical machines but they are implemented in software instead of hardware.
- By implementing the VM in a widespread portable language like C (or Java) programs compiled to the machine can be executed on many different platforms.

Virtual Machines (cont)

- A virtual machine is basically just an interpreter for virtual machine code.
- The same techniques can be used to implement a VM as an interpreter.
- The virtual machine code is usually linear (an array of instructions) as opposed to a tree as we used in the interpreter.

Virtual Machines (cont)

- When designing a VM one has some design choices similar to those for the intermediate code.
  - Should the machine be used on several different physical architectures and operating systems? (JVM)
  - Should the machine be used for several different source languages? (CLI/CLR (.NET))
  - Should the machine be efficient and reliable? (BEAM – Erlang).

Virtual Machines (cont)

- I will not go into the details of how to implement a VM.
- If you are interested you can look at the EVM:
  http://lampwww.epfl.ch/teaching/CompilationA/2003/html/project/files/EVM.java
- Or better yet, take my course "Compilation Avancée" next summer semester.

Part VII

Other Topics in Compilation

- Code Generation
  - Register Allocation
  - Instruction selection
  - Instruction scheduling
- Optimization
  - Constant propagation and folding
  - Loop optimization
  - Inlining & Partial evaluation
- Advanced language features
  - Concurrent programming languages
  - Functional languages
  - Object Oriented languages
  - Logic programming languages
  - Advanced type systems
- Runtime Environments
  - Concurrency & Parallelism
  - Memory managements & Garbage Collection
  - Efficient Virtual Machines

Part VII

Summary: Compilation

- Scanner
  - Converts a stream of characters into a stream of tokens.
  - Can be written by hand or with a scanner generator.
- Parser
  - Converts the stream of tokens to an abstract syntax tree.
  - Recognizes syntax errors.
  - Can be written by hand or with a parser generator.

Part VII

Summary: Compilation

- Analysis
  - Associates definitions and uses of names by attaching symbols to the AST.
  - Uses scopes to store Symbols.
  - Recognizes undefined variables.
  - Attaches types to the AST.
  - Recognizes type errors.

Part VII

Summary: Compilation

- Code generation
  - Turns the AST (or some other intermediate representation) into (virtual) machine code.
- Interpretation
  - Executes the user program