CSE 307: Principles of Programming Languages

Course Review

R. Sekar

Course Topics

- Introduction and History
- Syntax
- Values and types
- Names, Scopes and Bindings
- Variables and Constants
- Expressions
- Statements and Control-flow
- Procedures, Parameter Passing
- Runtime Environments
- Object-Oriented Programming
- Functional Programming
- Logic Programming

Programming Language

- A notational system for describing computation in a machine *and* human readable form
- Human readability requires:
  - abstractions
    - data abstractions, e.g., primitive and compound, structured data types, encapsulation, ...
    - control abstractions, e.g., structured statements (if, switch, loops,...)
    - procedure abstractions
  - promotes reuse and maintainability

Computational Paradigms

Imperative: characterized by variable assignment and sequential execution

- procedural
- object-oriented

Declarative: Focus on specifying what, reduced emphasis on how. No variable assignment or sequential execution.

- functional
- logic

History of Programming Languages (1)

1940s: Programming by “wiring,” machine languages, assembly languages

1950s: FORTRAN, COBOL, LISP, APL

1960s: PL/I, Algol68, SNOBOL, Simula67, BASIC

1970s: Pascal, C, SML, Scheme, Smalltalk

1980s: Ada, Modula 2, Prolog, C++, Eiffel

1990s: Java, Haskell

2000s: Javascript, PHP, Python, ...

History of Programming Languages (2)

FORTRAN: the grandfather of high-level languages

- Emphasis on scientific computing
- Simple data structures (arrays)
- Control structures (goto, do-loops, subroutines)

ALGOL: where most modern language concepts were first developed

- Free-form syntax
- Block-structure
- Type declarations
- Recursion

History of Programming Languages (3)

LISP: List-processing language — Focus on non-numeric (symbolic) computation

- Lists as a “universal” data structure
- Polymorphism
- Automatic memory management
- Mother of modern functional languages
- Descendants include Scheme, Standard ML and Haskell
  - Some of these languages have greatly influenced more recent languages such as Python, Javascript, and Scala

History of Programming Languages (4)

Simula 67:

- Object orientation (classes/instances)
- Precursor of all modern OO-languages
  - Smalltalk
  - C++
  - Java

Prolog:

- Back-tracking
- Unification/logic variables

History of Programming Languages (5)

C: "High-level assembly language"

- Simplicity
- Low-level control that enables OSes to be implemented mostly in C
  - Registers and I/O
  - Memory management
  - Support for interspersing assembly code

Java: A simple, cleaner alternative to C++

Reliability: Robustness/Security built from ground-up

Internet focused: "write once, run every where"

Bundled with runtime libraries providing rich functionality

Draws on some concepts from functional languages

Programming Language Design

The primary purpose of a programming language is to support the construction of reliable (and efficient) software

Efficiency was the singular focus in the early days (1940s and 1950s). As programs became more complex, other criteria gradually assumed more importance:

- Readability
- Language complexity
- Reliability
- Expressive power
- Maintainability
- Portability

Language Design Principles

Generality: a few general features and mechanisms, rather than many special cases

Orthogonality: language constructs can be combined in any meaningful way, rather than being restricted to certain combinations

Uniformity: Similar things should look and behave in similar fashion, and dissimilar things should be clearly distinguishable

Syntax Vs Semantics

- Syntax describes the structure of a program
  - Determines which programs are legal
  - Consists of two parts
    - Lexical structure: Structure of words

      Distinguish between words in the language from random strings
    - Grammar: How words are combined into programs

      Similar to how English grammar governs the structure of sentences in English
- Programs following syntactic rules may or may not be semantically correct.
  - Compare with grammatically correct but nonsensical English sentences
- Formal mechanisms used to describe syntax and semantics to ensure that a language specification is unambiguous and precise

Lexical Structure of Languages

- Lexical analysis or scanning recognizes lexemes or “words” in the language:
  - keywords
  - identifiers
  - literals
  - operators
  - comments
- Based on regular expressions
- Ambiguity resolution
  - Prefer longest match Prefer first match over subsequent matches

Syntactic Structure

“How to combine words to form programs”

- Context-free grammars (CFG) and Backus-Naur form (BNF)
  - terminals nonterminals productions of the form *nonterminal* $\longrightarrow$ sequence of *terminals* and *nonterminals*
- EBNF and syntax diagrams

Syntax-related topics

- Derivations
- Parse trees
- Ambiguity
- Resolution of ambiguity
  - operator precedence and associativity
  - shift over reduce
- Construction of parsers
- Abstract syntax trees (AST)

Data Types

A data type is a set of values, together with a set of operations that operate uniformly over this set.

Type checking: check the use of an expression w.r.t. specified declarations

Type inference: infer the types of expressions from their use

Type compatibility:

Data types

- Base types (aka primitive or simple types)
  - e.g., int, bool, real, enumerated, ...
- Type constructors to build compound types
  - cartesian product: labeled or unlabeled
    - unlabeled: tuples in ML, labeled: records in Pascal or ML, structs in C
  - union: tagged or untagged
    - tagged: algebraic data types in ML, variant records in Pascal; untagged: C/C++
  - array
  - pointer
  - function
- Recursive types
  - recursive type constructors supported in ML
  - simulated via pointers in imperative languages

Polymorphism

- Ability of a function to take arguments of multiple types
- Purpose: code reuse
- Parameteric polymorphism:
  - types parameterized with respect to other type
  - parameter types appear as type variables
  - example: template types in C++, generics in Java
- Overloading ("ad hoc polymorphism"): less general than parametric polymorphism but still allows reuse of client code.
  - Method overloading in Java, function or operator overloading in C++
  - virtual method calls in C++

Type Equivalence

- Type equivalence determines whether two types are the same:
  - structural equivalence
  - name equivalence
  - declaration equivalence
- ML: structural equivalence
- Pascal: declaration equivalence
- C/C++: name equivalence for structures/ unions, declaration equivalence for others

Type Checking

- Strong Vs Weak typing
  - Strong typing improves reliability by catching some runtime errors at compile time itself
- Static Vs Dynamic type checking
  - dynamic typing is more flexible, but less efficient since type info needs to be maintained and checked at runtime
- Type compatibility

Type Conversion

- Type coercion (implicit conversion)
- (Explicit) type conversion functions
- Type cast
- “Reinterpretation” in untaged unions union assigned with one type of value may be accessed using a different type

Names and Attributes

- Names are a fundamental abstraction in languages to denote entities
- Meanings associated with these entities is captured via attributes associated with the names
- Attributes differ depending on the entity:
  - location (for variables)
  - value (for constants)
  - formal parameter types (functions)

Static Vs Dynamic Binding

Binding: association between a name and its attributes.

Static binding: association determined at compile time

Dynamic binding: association determined at runtime

Examples:
- type is statically bound in most langs
- value of a variable is dynamically bound
- location may be dynamically or statically bound

Binding time: Earliest point in time when the association can be made. Also affects where bindings are stored

- Name → type: symbol table
- Name → location: environment
- Location →value: memory

Symbol Table

Maintains bindings of attributes with names:

$$\mathit{SymbolTable} : \mathit{Names} \longrightarrow \mathit{Attributes}$$

- In a compiler, only *static attributes* can be computed; thus:

  $$\mathit{SymbolTable} : \mathit{Names} \longrightarrow \mathit{StaticAttributes}$$

- While execution, the names of entities no longer are necessary: only locations in memory representing the variables are important.

  $$\mathit{Store} : \mathit{Locations} \longrightarrow \mathit{Values}$$

  (Store is also called as Memory)

- A compiler then needs to map variable names to locations.

  $$\mathit{Environment} : \mathit{Names} \longrightarrow \mathit{Locations}$$

Scope and Visibility

- *Scope* establishes the program region over which a binding (introduced by a declaration) is maintained
- *Lifetime:* the period during which a binding is effective.
- *Visibility:* Names may be hidden within their scopes due to redefinitions in inner blocks
    - *Scope qualifiers* (e.g., operator :: in C++) may be available to access hidden names

Variables and Constants

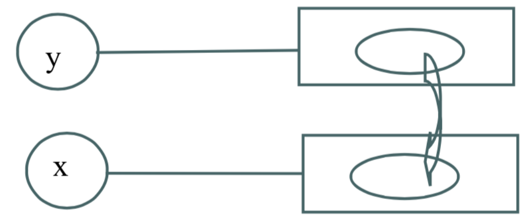
- Variables are stored in memory, whereas constants need not be.
  - Value of variables can change at runtime.
- Variables have a location (*l*-value) and value (*r*-value).
- Constants have a value, but no location.

Allocation

- Static
- Automatic or stack-based
  - for procedures and nested blocks
  - variables allocated on activation records on stack
  - lifetime of variables follows the LIFO property
- Dynamic or heap-based
  - for objects that do not possess the LIFO property

L-value, R-value and Assignment

- In an assignment x = y
  - we refer to l-value of x on the lhs ("l" for location or lhs of assignments)
  - r-value of y on the rhs ("r" for right-hand-side of assignments)
  - **Storage semantics:** update *location* of x with the *value* of y



- Accessing a value stored at a location is called "dereferencing".
  - C/C++/Java: l-values of variables on rhs are implicitly dereferenced to get their r-values.
  - In ML, dereferencing should be explicit, as in `x := !y`

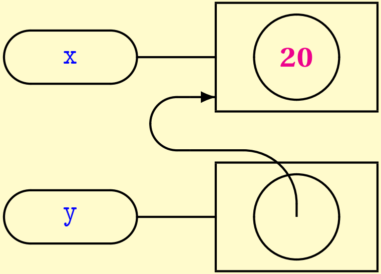
Pointers

- C/C++ “address-of” operation to explicitly turn a reference into a *pointer*.

  e.g. `&x` evaluates to the location of `x`.

  Example:

```
int x;
// x's location stores int
int *y;
// y's location stores
// pointers to int
x = 20;
y = &x;
```

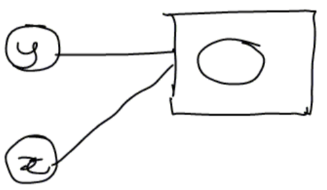


- The “*” operator is used to dereference a pointer

  e.g. in the above example, the value stored at `*y` is **20**

L-value and R-value (Continued)

- **Pointer semantics**
  - x simply “points” to y
  - more efficient if the size of y is large
  - but causes confusion in languages with assignment



- Java uses storage semantics for basic types, and pointer semantics for objects
- C/C++ use value semantics for all types
- In a language free of side-effects (i.e., memory updates), both semantics are equivalent.

Other topics in Variables and Allocation

- Arrays Vs Pointers
- Aliases
- Dangling pointers
- Garbage

Expression Evaluation

- Order of evaluation
  - More optimization opportunities if reordering of expressions is permitted, but the downside is that such reordering can change the expression value in the presence of expressions with side-effects.
- Strict (evaluate all subexpressions before applying an operator or function) vs lazy evaluation
  - All languages support lazy evaluation of if-then-else
  - Many languages require lazy (short-circuit) evaluation of boolean operations
  - Some languages support lazy evaluation of user-defined functions.
- Interpreter for expression evaluation

Control-flow statements

- if-then-else
- switch statement
  - issues, implementation techniques
- loops: while, for, repeat, ...
  - precise semantics
  - simulating one loop construct by another
- break, labeled break (Java), and continue
- goto
  - Allowable targets
    - Relates to scopes of labels and variables
  - goto's considered harmful
- Interpretation of assignments and control-flow statements

Procedure calls: Terminology

- Formal and actual parameters
- Caller Vs Callee
- Functions Vs Procedures
- Procedure activation
- Local Vs nonlocal variables
- Parameter Passing

Parameter passing mechanisms

- Call-by-value: evaluate actual, assign to formal parameter, execute body
- Call-by-reference: actual parameters must be l-values; evaluate and pass these l-values into the procedure
  - Needs care in the presence of aliasing
- Call-by-value-result: similar to CBV, but in addition copies formal parameters back to actual parameters after executing callee
- Call-by-name and call-by-need (lazy evaluation)
- All of these mechanisms can be understood precisely using
  - inlining the procedure body
  - ... after uniquely renaming local variables to avoid name capture
  - ... and substituting formals by actuals
- macros similar to call-by-name without local variable renaming.

Exceptions

- Streamlines error-handling
  - Enables exceptions to be dealt with by higher level procedures, while inner-level function may not deal with them at all
    - less cumbersome to write, as compared to return-value based error-handling
    - robust programs can be written even if some functions do not handle errors satisfactorily
- Implicit (rather than explicit) control-flow transfer mechanism
  - Other implicit CFTs: returns, calls using function pointers, etc.
- Resumption model (OS interrupts, UNIX signals) vs termination model (in almost all programming languages)

Implementing exceptions

- Exception handling uses dynamic scoping
- when exception is raised, the most recently encountered handler for the exception is invoked
  - if a handler exists in the current block, invoke it. Otherwise exit the current block and search recursively in the outer block
  - if current block is a procedure, exit the procedure, return to the calling environment, continue recursive search
  - requires information to be stored on the AR that records each exception handler encountered at runtime. This info is used at runtime during the above-mentioned search
  - in C++, code compiled with exception handling enabled contains code for such a search, while code compiled without this option does not. So, exceptions cannot be propagated across code compiled without this option.

Components of Runtime Environment (RTE)

Static area: allocated at load/startup time.

- Examples: global/static variables and load-time constants.

Stack area: for execution-time data that obeys a last-in first-out lifetime rule.

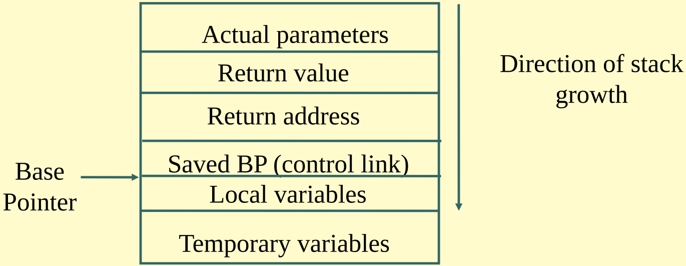
- Examples: nested declarations and temporaries.

Heap: a dynamically allocated area for "fully dynamic" data, i.e. data that does not obey a LIFO rule.

- Examples: objects in Java, lists in OCaml.

Procedures and the environment

- An Activation Record (AR) is created for each invocation of a procedure
- Structure of AR:



Actual parameters

Return value

Return address

Saved BP (control link)

Local variables

Temporary variables

Base Pointer

Direction of stack growth

Access to Local Variables

- Local variables are allocated at a fixed offset on the stack
  - Accessed using this constant offset from BP
    - Example: to load a local variable at offset 8 into the EBX register (x86 architecture)

      ```
      mov 0x8(%ebp),%ebx
      ```

- Example:

```
{int x; int y;
   { int z; }
   { int w; }
}
```

Steps involved in a procedure call

- Caller
  - Save registers
  - Evaluate actual parameters, push on the stack
    - Push l-values for CBR, r-values in the case of CBV
  - Allocate space for return value on stack (unless return is through a register)
  - Call: Save return address, jump to the beginning of called function
- Callee
  - Save BP (control link field in AR)
  - Move SP to BP
  - Allocate storage for locals and temporaries (Decrement SP)
  - Local variables accessed as [BP-k], parameters using [BP+l]

Steps in return

- Callee
  - Copy return value into its location on AR
  - Increment SP to deallocate locals/temporaries
  - Restore BP from Control link
  - Jump to return address on stack
- Caller
  - Copy return values and parameters
  - Pop parameters from stack
  - Restore saved registers

Resolving Nonlocal References

- Static/lexical scoping: meaning given by declarations in the blocks lexically surrounding the procedure
  - In languages with nested procedures, we need to maintain an access link in the current AR that points to the AR for the lexically surrounding scope. This link is used to lookup nonlocal references
  - In languages where functions can be created on the fly, ARs may need to be allocated on the heap
- Dynamic scoping: meaning given by most recently encountered declaration at runtime
  - we can simply follow the control links up the stack until we get to a scope that declares the nonlocal identifier.

Heap management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality
- Models
  - Explicit allocation and free (C, C++)
  - Explicit allocation, automatic free (Java)
  - Automatic allocation and free (OCAML)

Heap management: Concepts and Issues

- Issues
  - Fragmentation: internal vs external
  - Compacting to reduce wastage of space
- Garbage collection techniques
  - Reference counting
  - Mark-and-sweep
  - Copying collectors
  - Generational garbage collection
  - Conservative garbage collection

ADT

- Type is characterized by a set of operations
- Encapsulation: Only way to access the data is through these operations
  - access to internal representation of ADT is restricted
- Information hiding:
  - Semantics of operations don't depend on implementation
  - implementation can be changed without affecting “client code”, i.e., code that uses this ADT
- Supports following design goals
  - modifiability/maintainability, reusability, security

Modules and Name spaces

- Module
  - A way group "semantically related" code that may or may not operate on a single type
  - Export datatypes, variables, constants, functions
  - Ideal to support
    - separate compilation
    - library facilities
    - namespace separation (to avoid name clashes)
  - Examples: OCAML Modules, Java packages
- Namespaces: more narrowly focused on the namespace pollution problem.
  - Example: C++ namespaces, Java packages

Object-Oriented Languages: Goals

- Reuse
  - can we reuse implementation of a class even if we do not have access to its source code?
  - polymorphism
- Maintainability
  - minimize unnecessary dependencies by separating implementations of different system components, and making them as independent of each other as possible.

OO-Languages: Key Concepts

- Object = data + methods
- Encapsulation, information hiding, protection
- Subtype principle:
  - an object of a subtype may be used anywhere its supertype is legal

  with dynamic dispatching of methods
- Refinment and reuse via implementation inheritance

OO-Languages: Terminology

- Class
  - class (aka static) variables
  - class (aka static) functions
- Object (instance of a class)
  - Members
    - data (aka instance variables)
    - functions (aka methods or messages)
- Visibility
  - public, private, protected, ...

OO-Languages: Terminology (Continued)

- Subtype, subclass, derived class
- Supertype, super class, base class
- Subtype principle
- Virtual Vs Deferred methods

Inheritance

- Interface Inheritance (aka subtyping)
  - A is a subtype of B if A supports all of the interfaces supported by B, i.e., it supports all operations supported by B
  - Enables reuse of client code
    - if a function takes a parameter of type B, then this same function would work on all objects of type A. Thus, the function need not be reused for all subclasses of B.

- Implementation Inheritance
  - If C2 is a subclass of C1, C2 can potentially reuse all of the implementations of the methods in C1.
  - C2 need not support the same interface as C1, and hence may not be a subtype
  - Conversely, it is possible for C2 to be a subtype of C1, yet not reuse any of the implementations provided by C1
  - Here, reuse is enabled in a derived class of C1, rather than in a client of C2
  - This is a less critical feature of an OO language as opposed to interface inheritance.

Multiple Inheritance

- A class is a subclass of more than one class (or reuses implementaion of multiple clsses)
- Introduces ambiguities in the presence of âĂIJdiamond hierarchyâĂİ
- Many of these problems are solved if only the interface is inherited, but not necessarily the implementation

Dynamic Binding

- A function f may take parameters of class C1
- The actual parameter passed into the function may be of class C2 that is a subclass of C1
- Methods invoked on this parameter within f will be the member function supported by C2, rather than C1
- To do this, we have to identify the appropriate member function at runtime, based on the actual type C2 of the parameter, and not the (statically) determined type C1

Implementation of OO-Languages

- Data
  - nonstatic data (aka instance variables) are allocated within the object
    - the data fields are laid out one after the other within the object
    - alignment requirements may result in âĂIJgapsâĂİ within the object that are unused
    - each field name is translated at compile time into a number that corresponds to the offset within the object where the field is stored
  - static data (aka class variables) are allocated in a static area, and are shared across all instances of a class.
    - Each class variable name is converted into an absolute address that corresponds to the location within the static area where the variable is stored.

Implementation of Dynamic Binding

- All virtual functions corresponding to a class C are put into a virtual method table (VMT) for class C
- Each object contains a pointer to the VMT corresponding to the class of the object
- This field is initialized at object construction time
- Each virtual function is mapped into an index into the VMT. Method invocation is done by
  - access the VMT table by following the VMT pointer in the object
  - look up the pointer for the function within this VMT using the index for the member function

Implementation of Inheritance

- Key requirement to support subtype principle:
  - a function f may expect parameter of type C1, but the actual parameter may be of type C2 that is a subclass of C1
  - the function f must be able to deal with an object of class C2 as if it is an object of class C1
    - this means that all of the fields of C2 that are inherited from C1, including the VMT pointer, must be laid out in the exact same way they are laid out in C1
    - all functions in the interface of C1 that are in C2 must be housed in the same locations within the VMT for C2 as they are located in the VMT for C1