An Introduction To
Programming Language Concepts

Day 1

August 27

Language Design Considerations

Code will be designed to support network applications

Code must support safety and security

Properties of code can be proven mechanically
- a bad state can never be entered on any input
- a user cannot read before writing to a location

Code must run efficiently

Prototyping should be relatively rapid
- worry over pointer ops or structures slows development

Chance of run-time errors should be minimized

Maintenance costs should be low
- problems should be found before deployment
- a maintainer can understand code not seen before

Concepts to be Considered

values and types:

static, dynamic, inferred, persistent, transient, complete, safe

variables and storage:

simple, complex, local, global, heap, lifetime

bindings and scope:

association between identifier and entity (value, object)

procedural abstraction:

separate function from program mechanics

data abstraction:

classes, subclasses, inheritance

paradigm:

imperative, functional, declarative, object oriented

Programming Paradigms

Imperative (C, Java)

Statement execution results in state changes and is determined via control structures

Side effects make it difficult to assure correct computation

Decomposing for parallel execution is difficult due to possible need for process communication

Efficient since state changes are usually incremental

Any variable may be referenced, control may be transferred to any arbitrary point, and any variable binding may be changed so the whole program may need to be examined to understand even a small portion of code.

Appearance of Fortran and Lisp at same time shows early machine architectures did not force imperative style

Programming Paradigms

Procedural (C)

Imperative but using procedures or functions

a coder may reason about a program sans data structures

but statement execution is still deterministic

Object Oriented (C++, Java, Scheme)

Data types may be defined by the coder

data is hidden from general access

methods are developed to operate on data

a non-owner manipulates data through those methods

Data subtypes may also be defined, with permission

inheritance, multiple inheritance

Constructs for polymorphism

code still works in the presence of new data types

Programming Paradigms

Functional (Scheme, Haskell)

Computation is the evaluation of math and logic functions

State is not changed - no side effects
no assignment statement

Developed from lambda calculus

Referential transparency
execution results are the same, regardless of time
that is, results only depend on argument values
loops are discarded in favor of recursive calls

Anonymous subroutines

Lazy evaluation (sometimes manually via above)

Procedures are first class objects (as is data)

Programming Paradigms

Declarative (SQL, Prolog, Cryptol)

- Set of constraints must be true under some interpretation
- No control structures used in coding
- Side effects are minimized or eliminated
- Can be used as a functional specification (modeling)

Values and Types

What is a data type?

A set of values

A collection of operations on those values
application of operations must be uniform for all values

There may be a type hierarchy
e.g. integer may be a subtype of rational

The type of a value returned by an op may be a supertype

There are built-in primitive types (cannot be decomposed)
e.g. int, string, boolean

Types may be grouped as tuples

There may exist recursive types (lists, trees)
defined in terms of themselves

Type Systems

What is a type system?

A grouping of values into types
  syntax and grammar of available types &
    portions of parser handling type declarations &
    portions of compiler checking types of parameters
  enables coders to describe data effectively and to prevent operations that make no sense during run-time

Types of operands must be checked before op is applied

Statically typed language (Java, Haskell):
  all variables and expressions have fixed type
  types are checked at compile-time

Dynamically typed language (Scheme, Python):
  variables and expressions do not have fixed type
  types are checked at run-time

Static vs. Dynamic Type Systems

Static typing is more efficient

Dynamic typing requires run-time type checks and forces all values to be tagged to make the type checks possible
Static typing requires only compile-time type checks, and does not force values to be tagged

Static typing is more secure

A compiler may be able to guarantee that a program contains no type errors. Dynamic typing provides no such security

Dynamic typing is more flexible

Needed by applications where data types are unknown in advance

Type Completeness

Operations on typed entities:

1. Can be assigned a value
2. Can be composed with other entities in composites
3. Can be passed as an argument to a function
4. Can be returned by a function as a result

Typed entity is first-class if all above can be done

Java:

primitives, class objects are first class
classes are first class via reflection (as ByteArray)

Scheme, Haskell:

data and procedures are first-class and may be exchanged
e.g. `(if ... then sin else cos)(x)`

Type Safety

What is it?

The absence of erroneous or undesirable behavior caused by a discrepancy between differing data types

C++ is not type-safe

```
int main() {
    int ival = 5;
    void *pval = &ival;
    double dval = *((double*)pval);
    cout << dval << endl;     // 5 is not output (try it)
    return 0;
}
```

Java is supposed to be type-safe

If your application has been compiled without unchecked warnings using javac -source 1.5, it is type safe - Oracle

Haskell is pretty much type-safe, Scheme is not

Storage

Copy vs. reference on assignment

Java copies primitive values only

Forgetting the `set!` family, Scheme does not assign

Haskell does not assign, new states are created by copying

Garbage Collection

Scheme, Haskell, Java have garbage collection

- as long as there is an active reference, variable is alive

Memory Safety

Avoid jumps to invalid data addresses and manipulation of code addresses

- C/C++ are highly unsafe - dangling pointers, etc.
- Java is fairly safe - no deallocator
- Haskell is safe - no assignment (has pointers though)

Storage

Stack

Store states of subroutines that are suspended

Multi-threaded programs have ≥ 1 independent stacks

Heap

Store all variables that are alive

Can be organized as a tree of stacks, sharing some state

Bindings

What is it?

A fixed association between an identifier and an entity such as a value, variable, or procedure

What is a set of bindings?

An environment or namespace

Why is an environment important?

Each statement is executed in an environment

What is the scope of a binding?

That portion of the code over which the binding has effect
Typically decided by the block structure of a language
Blocks may be nested

Bindings

In different blocks:

If identifier `I` is declared in two different blocks, `I` denotes a different entity in each block

If identifier `I` is declared in two nested blocks, `I` denotes one entity in the outer block and a different entity in the inner block. The inner declaration hides the outer one

Static vs. Dynamic Scoping:

Static: procedure executes in enviroment of its definition
  Can be determined at compile time

Dynamic: procedure executes in enviroment of its caller
  Cannot be determined at compile time

Nearly all languages are statically scoped

Lisp is dynamically scoped by accident

Procedural Abstraction

What is it?

Separates what a program does from how it does it

Why is it important?

Separates concerns of implementer from those of app coder
Important for success of large scale programming projects

App programmer: knows procedure's observable behavior
Implementer: knows algorithms needed to make it work

Procedural Abstraction

Function procedure:

Procedure only returns a value - no change to environment

Procedure has a body that is an expression

Function call is an expression that takes a value by evaluating the procedure body

Proper procedure:

Procedure has a body that is a command

Execution of the body changes variable values

Selector procedure:

Procedure returns a reference to a variable

Considered an abstraction over variable access

Procedural Abstraction

Argument:

Value that is passed to a procedure

Actual parameter:

An identifier through which a value is passed to a procedure

Formal parameter:

Identifier in the argument list of a procedure

What can be passed:

Any first-class object
Variables and pointers to variables (usually)
Procedures and pointers to procedures (maybe)

Variable Passing

Call-by-value:

A copy of the value of the variable is made and passed to the corresponding formal parameter - variable's value is unchanged after the procedure returns

Call-by-reference:

The location of the variable is passed to the corresponding formal parameter - the variable may be updated several times in the body of the procedure

Call-by-value-result:

Same as call-by-value except that the variable takes the value of the corresponding formal parameter at termination

Call-by-name:

The address of a function that computes the value of the actual parameter - effectively substitutes the parameter

Parameter Passing

Normal order:

Parameters are evaluated as needed in the body of the procedure

Applicative Order:

Parameters are evaluated before the procedure is invoked

Strictness:

If an argument is evaluated before the body of a procedure is entered, the procedure is strict in that argument

Non-strictness:

If a procedure is entered before an argument has been evaluated, the procedure is non-strict in that argument