CSCI 3136

Control Flow to Object-Orientation

Zeh

Alex Safatli

Monday, February 18, 2013

Contents

Control Flow

Mechanism

The successful programmer thinks in terms of basic principles of **control flow** and not in terms of syntax. The principle categories of control flow mechanisms are:

- **sequencing** (sequence of instructions),
- **selection** or **alternation**,
- **iteration** (run for a finite number of steps and substructures),
- **procedural abstraction** (ability to define functions and procedures — TOP-DOWN building more complex and macroscopic *primitives* from basic, microscopic *primitives*),
- **recursion** (can replace iteration for functional languages, etc.),
- **concurrency**,
- **exception handling** and **speculation** (how to implement methods of non-local transfer of control), and
- **non-determinism** (closely tied to concurrency).

Expression Evaluation

EXPRESSION EVALUATION is affected by order of evaluation: this may influence result of computation. For **purely functional languages**, interaction is avoided as much as possible with the outside world;

- computation is **expression evaluation**,
- the only effect of evaluation is the returned value — no side effects, and
- order of evaluations of subexpressions is *irrelevant* — this will not affect any other part of the program or will not depend on any other part (other than arguments passed in).

This opens the door to many things: optimization can make use of changing around the order of functions (without effect) and lazy expression evaluation (does not get evaluated unless needed). Usually this can only be achieved in small subsets of a program. If the outside world cannot be affected with, the program probably should not have been written in the first place. For **imperative languages**,

- computation is a series of changes to the values of variables in memory,
- this is *computation by **side effect*** (views of the world, memory content changes with expression evaluation — successive instructions will depend on these),
- the order in which these side effects happen may determine the outcome of the computation;
- there is usually a distinction between an **expression** (do because their evaluation result is important) and a **statement** (do because what is cared about is the way they affect the world).

Assignment is the *simplest* (and most fundamental) type of side effect that a computation can have. It is *extremely important* in imperative programming languages, and much less important in declarative (functional or logical) programming languages (values are important only).

```
A = 3
```

in Java, C++, C, ...

Expression Evaluation continued on next page...

REFERENCES AND VALUES. Expressions that denote values are referred to as **r-values**. Expressions that denote memory locations are referred to as **l-values**. The meaning of a variable name normally differs depending on the side of an assignment statement it appears on. On the right-hand side, it refers to the variable's value — it is used as an r-value. On the left-hand side, it is the variable's location in memory — it is used as an l-value.

NOTE Some languages *explicitly* distinguish between l-values and r-values. In BLISS, variables only refer to a memory location. Dereferencing is important. In some languages, a function can return an l-value (e.g. ML or C++).

There exists different *models of variables*. In a **value model**, assignments copies the value. In a **reference model**, a variable is always a reference — assignment makes both variables refer to the same memory location. We can then distinguish between variables referring to the same object and variables referring to different but identical objects.

NOTE In Java, the value model is used for built-in types and the reference model is used for classes. C passes by value for assignment unless explicitly referred to by pointer.

EVALUATION ORDERING WITHIN EXPRESSIONS. It is usually unwise to write expressions where a side effect of evaluating one operand is to change another operand used in the same expression. Some languages explicitly forbid side effects in expression operands. See examples.

Evaluation order is often left to the compiler (undefined in the language specification) — thus, such side effects may lead to unexpected results. Specifying order avoids this issue but leads to another problem... Evaluation order impacts register allocation, instruction scheduling, ... By fixing a particular evaluation ordering, some code improvements may not be possible. This impacts performance.

SHORT-CIRCUIT EVALUATION OF BOOLEAN EXPRESSIONS. If the value of the expression does not depend on a further calculation, that evaluation is skipped. This is a useful optimization, but if the evaluation has side-effects, the meaning of the code may be changed. Most languages use this, but some languages provide both regular and short-circuit versions. Leads to useful idioms (cleaner semantics).

Control

Sequencing comes naturally in imperative programming languages, without a special syntax to support it. On the other hand, mixed imperative/functional languages (LISP, Scheme, ...) often provide special constructs for sequencing. Ultimately, this brings up the issue of what the value is of a sequence of expressions/statements. Commonly, this is the value of the last subexpression. To build non-trivial programs, sequences of instructions have to usually be provided.

GOTO AND ALTERNATIVES. Use of `goto` is typically bad programming practice if the same effect can be achieved using different constructs. Early languages may not have had these constructs and `goto` was necessary. Sometimes it is unavoidable: breaking out of a loop, of a subroutine, or a deeply nested context. It confuses the readability of the code, however. Many languages provide alternatives: one-and-a-half loops, `return` statements, and structured exception handling.

Selection or **alternation** involves the execution of certain blocks of code depending on state. Normally, `goto` is used to implement loops or alternation. Almost every programming language has a `if-then-else` statement, multi-way `if-then-else` statements (a convenience; is not more expressive or cheaper), or `switch` statements (looking for the first case that holds; more restrictive multi-way branching — single expression with different conditions... this is therefore less expressive but this is usually more readable).

Control continued on next page...

`switch` statements are a special case of if/then/elsif/else. The principle motivation of them is to generate *more* efficient code. Compilers can use different methods to generate efficient code: sequential testing, binary search, hash tables, or jump tables.

`if-then-else` is a number of condition tests with a large number of `gotos`. At the assembly level, there are no special constructs. Furthermore, this can be expensive due to the nature of instruction pipelines because the processor is faster than a memory bank. On the other hand, a switch statement can merely be represented using a **jump table** — values are extremely limited in range. The processor is not sitting idle as it may be in the former situation.

So what are the benefits and disadvantages of the different ways of implementing switch statements? **Jump tables** are fast (one lookup table) but may have a potentially very large table. **Hash tables** are fast, but are more complicated (at the assembly level) and elements in a range need to be stored individually (a large table, potentially). **Linear search** is potentially slow but has no storage overhead. And finally, **binary search** is fast (but slower than table lookup) and has no storage overhead. NO SINGLE IMPLEMENTATION IS BEST IN ALL CIRCUMSTANCES. COMPILERS OFTEN USE DIFFERENT STRATEGIES DEPENDING ON THE SPECIFIC CODE.

Iteration can be in one of two forms.

1. **enumeration-controlled loops**, and
2. **logically controlled loops**.

Enumeration-controlled loops include `for`-loops; one iteration is done per element in a finite set. The number of iterations is known in advance. These refer to FORTRAN loops such as `FOR i=1 TO 10 STEP 2`. Logically controlled loops include `while`-loops and are executed until a Boolean condition changes. The number of iterations is not known in advance.

Some languages *do not* have loop constructs (e.g., Scheme) — looping in these languages make no sense whatsoever; they use tail recursion instead. These may be inefficient, but when loops are done, they are typically the last instruction in a procedure, so there is no reason to keep the stack frame around.

Logically controlled loops include: **pre-loop** tests, **post-loop** tests (loop executed at least once), and **mid-loop** tests (one-and-a-half loops). The latter involves breaking out of the loop; this can be done with some manner of `break` statements or `continue` statements.

What are the trade-offs in iteration constructs? Logically controlled loops are very flexible but expensive. Any condition can be specified, and inside the iteration, any updates can be made to variables that are involved in the condition. Ultimately, they come at a cost: this is very little you can do to optimize. The `for`-loop in C or C++ is merely syntactic sugar for the common *init-step-test* idiom found in logically controlled loops.

Potentially more efficient are enumeration-controlled loops; optimizations can be done. If modifying the loop variable inside is allowed, or if modifying is not allowed are the two steps that must be considered here. The latter is more efficient.

LABELLED BREAK & CONTINUE. A **break statement** exists the nearest enclosing `for`, `do`, `while`, or `switch` statement. A **continue statement** skips the rest of the current iteration. Note that even without them, the same effect can be done using properly structured `if` constructs or by changing the loop condition — the cost is no different. Both statements may be followed by a label that specifies an enclosing loop or any enclosing statement — this is usually not doable with normal constructs. A loop may also have a `finally` part which is always executed no matter whether the iteration is executed normally or terminated.

GENERATORS. Going one level of abstraction higher, we have iterators and **generators**. Very often, `for` loops are used to iterate over sequences of elements (stored in a data structure, generated by a procedure, ...). Iterators/generators provide a clean idiom for iterating over a sequence without a need to know how it was generated.

Control continued on next page...

See example from Python — no storage is done; they are generated one at a time using a **code routine** but it is indistinguishable (`yield` merely suspends and does not return).

How do we implement iteration in another language without this feature? C++ and Java provide iterator classes that can be used to enumerate the elements of a collection, or programmatically generate a sequence of elements to be traversed. Loops do not have to care what a container looks like on the inside; from its point of view, it is containing them as elements.

This is nowhere near as convenient as in Python where **coroutines** are being used. Telling an iterator to "increment" will force it to do something and *then* return. In the generator in Python, nothing has to be done to remember where the suspended `yield` statement was in the code.

Iteration can be done without iterators; in C, the code for this would be:

```
for (it=begin(coll); it != end(coll); it = next(it)) {
   /* Do something with it. */
}
```

ITERATION IN FUNCTIONAL LANGUAGES. Functions as first-class objects allow passing a function to be applied to every element to an "iterator" that traverses the collection. An example in Haskell would be:

```
--- print first ten Fibonacci numbers,
--- each multiplied by 2.
main = do
   mapM_ (\x -> putStr $ show x ++ " ") fib2
   putStrLn ""
   where
      fib2 = map (* 2)
         $ take 10 fibonacci

--- the infinite sequence of Fibonacci numbers
fibonacci = 1:1:(zipWith (+) fibonacci (tail fibonacci))
```

If we want to iterate the way we were talking about before this, we need **loops**. Realize that functional languages *do not have loops* – they do not exist. We will see later why this is. Therefore, the resort is to turn things *inside out*: instead of getting elements one at a time and then doing something to them, a function is instead to an iteration procedure which applies this to every element in the object. **Ruby** is another language that implements this but is *not* functional.

Recursion

Every iterative procedure can turned into a **recursive** one:

```
while (condition) { S1; S2; ... }
```

becomes

```
procedure P() {
   if (condition) { S1; S2; ...; P(); }
}
```

The converse is *not* true (e.g. quicksort, mergesort, fast matrix multiplication, ...). This is the truth on the level that we want this transformation to be *easy*. If we simply have no choice, doing this involves a **stack**. A procedure could be implemented as a loop that takes the top-most stack frame and works with it. The stack can always be explicitly

Recursion continued on next page...

be maintained iteratively, but it is not easy. The more abstract answer to this answer can be seen by looking at the processor. The processor is a **finite state machine**: it maintains extra information by storing the right data in memory. Therefore, anything recursive, one going through the compiler, will actually be translated into something that is iterative.

So, *why do functional languages not support iteration*? What is being said here that it is impossible in a purely functional language. One of the core features of a purely functional language is that functions *cannot* have **side effects**. Therefore, the only way a function interacts with the outside world is through the arguments passed in and through the return value. Loops are strictly imperative: it relies on updating an **iterator variable**.

Would C still be capable of high performance without iteration — without loops? If the condition is true for an extremely long time, a large amount of stack frames will *not* be efficient. Therefore: A NAIVE IMPLEMENTATION OF RECURSION IS OFTEN LESS EFFICIENT THAN A NAIVE IMPLEMENTATION OF ITERATION.

An optimizing compiler often converts recursion into iteration when possible:

- **Tail recursion**: there is no work to be done after the recursive call (standard method for implementing iteration in functional languages). Recursion is done at the end. Therefore, the above is being done in reverse. Compiler will turn this into a *loop*.
- Work to be done after the recursive call may be passed to the recursive call as a **continuation**. This is another way to avoid the cost of continuously calling recursively. A continuation is a snapshot of the state of a given time, so the effect is there is a jump back to where the procedure is called from. In theory, there is no need to track a call stack, so it should be more efficient. But, continuations are even more costly than call stacks.

NOTE When n calls are made, n frames are made in a given closure.

See following slides for examples of (tail) recursion. In the two examples, the first implementation is not tail-recursive but can be converted into one. Note that **tail-recursive functions** *imitate* iteration (always), and in the absence of side effects they use **linear space**. Using garbage collection, the space bound is, in fact, *constant*.

Applicative and Normal Order of Evaluation

Applicative-order evaluation is an evaluation where arguments are evaluated before a subroutine call. This is default in most programming languages.

Normal-order evaluation is where arguments are passed unevaluated to the subroutine. In a way, they are passed "textually". The subroutine will evaluate them as needed. This is useful for infinite or lazy data structures that are computed as needed. Examples are macros in C or Haskell. The advantage of doing something only as you need it is so big that quite a number of modern languages make this the default. Haskell is *lazy* – if it uses an expression once, it then remembers the value and does not recalculate (acts as somewhere in the middle).

This is fine in functional languages but problematic if there are side effects. Why? Side effects will happen multiple times and this is normally not what one person would expect. If the expression `i++` is constantly evaluated, this could be a problem. This also has the potential for inefficiency. Why? How can this be avoided? We do not want to reevaluate expensive expressions over and over again; caching the value avoids this (only an option if there are no side effects).

LAZY EVALUATION IN SCHEME. By default, Scheme uses *strict* evaluation. If we try to define the infinite sequence of *natural numbers*, in order to build the list it has to build consecutive lists endlessly. For example, this code runs forever:

```
(define naturals
   (letrec ((next (lambda (n)
                    (cons n (next (+ n 1))))))
      (next 1)))
```

Nevertheless, it is sometimes convenient to build infinite data structures. Conceptually, the "infinite list" is built and only a portion of it is worked with (a prefix of finite length). Using lazy evaluation explicitly, this can be avoided (see slide) by using `delay` (you can evaluate it when you have to).

Conceptually, the `naturals` structure is merely a set of two pointers: `car` and `cdr` where `car == 1` and `cdr == delay (next 2)`. Consequently, `cdr`, once evaluated, points to another set of pointers, `car == 2` and `cdr == delay (next 3)`.

LAZY EVALUATION IN HASKELL. Lazy evaluation is the default in Haskell. If maximum efficiency is needed, strict evaluation of function arguments and data structure members can be forced. All of the `delay` functionality is done under-the-hood. See slide. Forcing strict-ness involves using the `'seq'` operator.

How are `delay` and `force` implemented? `delay` can be a **special form** or **macro** (applicable order; simple textual replacement) that wraps the expression in an **anonymous function**. `force` simply evaluates the given function. See slide.

What is the problem with this implementation of `delay`? Note that it *evaluates* `delayed-exp` every time. This is inefficient. If costly, it will have to be paid every time.

A better implementation is what Haskell pretty much does by default (see slide). A closure is defined which *remembers* if the function was evaluated once before.

NORMAL-ORDER EVALUATION IN C/C++. Returning to **macros** — this is more cleanly applicative-order evaluation (complete textual replacement). However, the only problem with them is that they cannot be used recursively (they are not functions), and textual expansion may not mean what is intended; see slide.

Also note that if arguments are side effects, they could potentially be executed more than once. Furthermore, there is the possibility of name clashes with variables of the macro. In C++, *inline functions* are usually a better alternative.

Subroutines and Control Abstraction

Program Building Blocks

In general, when programs are built, abstractions are being built. Programming is very much about **building abstractions**. **Subroutines** are the main method to build control abstractions. The other form of abstraction we normally think about is **data abstraction**.

Subroutines are what we normally call **functions**, if they return a value, or **procedures** if they do not and thus are called for their side effects. Parameters to a subroutine are:

- **Formal parameters**: parameter names that appear in the subroutine declaration.
- **Actual parameters** or **arguments**: values assigned to the formal parameters when the subroutine is called.

We already discussed stack-based allocation of **activation records** and their maintenance by caller and callee before, at the beginning, at the end, and after the subroutine call. This is the *whole process of calling a subroutine*.

Note that even if a language is being used with static scoping, the dynamic chain and static chains both have to be maintained. An explicit reason for this is **exception handling**.

Program Building Blocks continued on next page...

CAN WE ACTUALLY AVOID THE COST OF THIS PROCEDURE? **Inline expansion** replaces a subroutine call with the code of the subroutine. This has advantages – we do not incur the overhead of a full-blown call sequence (faster code), encourages building abstractions in the form of many small subroutines, and it is related to but cleaner than macros.

However, there are also disadvantages to this as well. It could provide code bloating and *cannot be used* for recursive subroutines: a stack *has* to be kept.

WHAT HAPPENS WHEN A FUNCTION IS CALLED IN REGARDS TO PARAMETERS? **Parameter passing** refers to the execution of the named subroutine with its formal arguments bound to the provided actual arguments. How exactly? Parameter passing modes include the semantics:

- *by value* (on the outside, no effect of modification is seen),
- *by reference (or by sharing; actual parameter must be an **l-value** (reference to a memory address))*, or
- *by value/return* (on the outside, all modifications are seen; copy is copied back — this is useful if an error occurs).

The semantics are driven by the implementation, not the other way around.

NOTE FORTRAN has all parameters passed by reference; temporary variables are used to pass non-l-value expressions. PASCAL is call by value by default; keyword `var` before a formal parameter switches to call by reference. C is call by value always — arrays are passed by values as pointers and to simulate call by reference, **pointers** are used. Ada compilers determine if `in out` parameters are call by reference or call by value/return. C++ is the same as C but with the addition of reference parameters — references can be declared `const` so it has the efficiency of call by reference and safety of call by value. Java is call by value for primitive types and call by sharing for compound types (objects). C# is a little strange (see slides).

NOTE Calling by sharing is typically done when there is a reference model of variables. Everything is a reference already. In languages like Smalltalk, Lisp, Clu, and ML, objects can be altered (just as with call by reference), but the **identity** of the object cannot be changed. References are passed by value, and a called subroutine can still change the caller's object. Unlike with call by reference where the object can be completely overridden, assigning things to a reference decouples it from the caller's variable. It can be manipulated, just not changed to another object.

Read-Only Parameters address the problem of large values being passed by reference for efficiency reasons with a high potential for bugs (this incurs no cost; any attempts to touch it are caught at compile-time). It has the efficiency of call by reference with the safety of call by value. in C or C++, these are `const` parameters. When using call by value, declaring a parameters `const` is pointless.

Some uses of this in C++ include constant definitions, read-only function parameters, immutable references returned by functions, and object methods that cannot change objects (the only type of method that can be invoked on a `const` object; allows for inspecting values in an object — call methods that are ensured to not change the object.).

Parameters and Generics

We have looked at parameter passing modes. What if we want to pass **subroutine closures** as closures? Functions as parameters and function return values require the passing of closures. Languages that support this include

Parameters and Generics continued on next page...

Pascal, Ada 95, and functional programming languages. Note that this *does* incur a cost to maintain and allocate them. Immediately requires or encourages garbage collection.

There is a **restricted passing of functions** in C, C++, and FORTRAN: functions are not allowed to nest (or not significantly in FORTRAN), there is no need for closures, and pointers to subroutines suffice.

Note the use of **default (optional) parameters** as well. These are parameters that need not be specified by the caller and if they are not, they take default values.

The implementation of these are trivial and are found in Ada or C++. They come at no runtime cost and the amount of sophistication required is fairly minimal. Implementation merely requires *counting* the number of arguments — not having enough calls on code created by the compiler to push the default value onto the stack.

Named (keyword) parameters need not appear in a fixed order; this is good for documenting the purpose of parameters in a call and necessary to utilize the full power of default parameters. Implementation is once again trivial here. Rearranging the tokens to the expected input is done by the compiler and compiled as normal. Modern scripting languages like Python allow you to do this.

Variable number of arguments is allowed by languages like C and C++. Java and C# provide similar facilities in a typesafe but more restrictive manner. Think of the `printf` function where an argument is expected for every `$` conversion specification.

Standard subroutines allow the same code to be applied to *many different values*. **Generic subroutines** can be applied to *many different types*. There are trade-offs involved in balancing the runtime efficiency and generality of the framework with type safety.

Lisp, Scheme, Ruby, and Python use **runtime type checks** — they possess **weakly typed** variables. Nothing goes wrong when the parameters are passed in but checks are made when functions expect something it can work with. Compilation does not do anything.

Compile-time type checks upon instantiation are done in languages like C++ with C++ templates. In languages like Java, what is done is **compile-time type checks upon declaration**. This is not quite true; the type checks are split among two different places: the compiler checks if the code uses only methods that are guaranteed to exist upon the premise that a certain class is inherited and whether or not the class being used to instantiate is inheriting the required class. Haskell's concept of type classes is very similar to Java interfaces.

WHAT ARE THE RUNTIME COSTS OF THESE THREE DIFFERENT APPROACHES? For runtime type checks, only one copy of the code is present. The only way this can be done is with a reference model of variables. This comes at the cost that you require another level of abstraction — a runtime cost.

For compile-time type checks upon instantiation, when defining a stack frame, enough room has to be made for allocating variables. The consequence of defining a function like this in a language that imposes a value model of variables *requires* a recopying of the function. This is a compile-time cost that increases the size of the file but is efficient at runtime.

For the last approach, the code is generated only once but it has the added benefit of type safety. It is somewhere in the middle of the two.

Exception Handling

An **exception** is an *unexpected* or an abnormal condition arising during program execution. They may be generated automatically in response to runtime errors or raised explicitly in the program.

TYPICAL SEMANTICS OF EXCEPTION HANDLING (some older languages deviate from this). **Exception handlers** lexically bind to a block of code, an exception raised in the block replaces the remaining code in the block with the code

Exception Handling continued on next page...

of the corresponding exception handler, and if there is no matching handler, the subroutine exits and a handler is looked for in the calling subroutine.

Exception handlers means that operations necessary to recover from the exception must be performed. They terminate programs gracefully (with a meaningful error message) and clean up resources allocated in the previous block before re-raising the exception. Control transfer is done in a very meaningful way: somewhere deep in a stack, somewhere stacks higher there is something that knows how to deal with it. These are heavily **non-local control flow structures**.

In programming languages, **exception support** is typically done by representing exceptions as a *built-in type*, an object can be derived from an exception class, or any kind of data can be raised as an exception. The latter seems to be the most flexible and best decision here.

Raising exceptions is done automatically by the run-time system as a result of an abnormal condition (e.g., division by zero) — `throw` or `raise` statements are used to raise them manually.

Exceptions are handled by most languages locally and propagate unhandled exceptions up the dynamic chain. Clu does not allow exceptions to be handled locally (callers can handle exceptions; by wrapping the protected code, local exception handlers could be simulated). Finally, PL/I's exception handling mechanism is similar to dynamic scoping.

> **NOTE** Some languages require exceptions thrown (but not handled inside a subroutine) to be declared as part of the subroutine definition (e.g., Java).

Handling exceptions *without* language support involves inventing a value that can be used instead of a real value normally returned by the subroutine. This returns an explicit "status" value to the caller and the caller has to check the status (e.g., C). Furthermore, this relies on the caller to pass a closure to be called in case of an exception — this is really only half a solution.

Exception propagation involves examining, in order, exception handlers in the current scope. The first one matching the exception is invoked and if no matching handler is found, the subroutine exits and the process is repeated in the caller, potentially ending in abnormal exiting. This is how *real* exceptions work and are handled in a run-time system.

Therefore, stacks must be unwound (restored to previous state) and any necessary clean-up needs to be performed (e.g., deallocation of heap objects, closing of file descriptors). Some languages provide support for this using constructs such as `finally` in Java.

```
try {
   ...
}
catch (SomeException e) { // exception handler
   ...
}
...
finally {
   ...
}
```

A SIMPLE IMPLEMENTATION of exception handling at runtime involves having every subroutine or protected code block pushing its exception handler onto a **handler stack**. Exception handlers with multiple alternatives are implemented using if/then/else or switch statements in the handler. Finally, every subroutine pushes a special exception handler onto the stack that is executed when control escapes and performs all necessary clean-up operations

(restoring frame pointer — unwinds stack). This works, but it is *costly because it requires the manipulation of the handler stack for every subroutine call or return.*

A FASTER IMPLEMENTATION stores a **global table** mapping the memory addresses of code blocks to exception handlers (this can be generated by the compiler). When encountering an exception, perform a **binary search** on this table using the program counter to locate the corresponding handler. Comparing this to the more simple mechanism:

- handling an exception is *more costly* (binary search — logarithmic *v.* constant of the simple implementation) but exceptions are expected to be rare,
- in the absence of exceptions, the cost of the mechanism is *zero*, and
- it cannot be used if the program consists of separately compiled units and the linker is *not* aware of this exception handling mechanism.

A HYBRID APPROACH stores a pointer to the appropriate table in each subroutine's stack frame. The first thing that is done when an exception is encountered is identifying this pointer, and then a fast lookup is done for the corresponding exception handler. The upside for this is that you are no longer maintaining a separate stack, but the stack frames become slightly bigger and the initializations of subroutines need this table address.

NOTE In Java, `throw` throws an exception, ..., only `Throwable` objects can be thrown. In C++, any object can be thrown and exception declaration on functions are not required. The latter is a decision of convenience over crash safety.

Scheme does not support exceptions. However, it has a much more general construct that subsumes subroutines, coroutines, exceptions, ...: **continuations**. A continuation is a "future" of current computation represented as the following, a representation of **machine state**:

- current stack content,
- referencing environment,
- register content,
- program counter, ...

Continuations are first-class objects in Scheme: they can be passed as function arguments, returned as function results, and stored in data structures. Continuations can let you handle just about anything; fully recursive programs, for instance, could be completely created without actually using recursion.

NOTE Scheme is sort of a meta-language rather than a language; experimental platform. Does not have exception support or built-in support for object orientation. However, any good Scheme implementation comes with at least 3 or 4 frameworks for doing object orientation.

`(call-with-current-continuation f)` calls a function `f` and passes the current continuation to `f` as an argument. Most schemers define `call/cc` as a macro. The simplest possible use is an escape procedure. In the example, the function is true or `#t` if all values are positive.

NOTE An improper list in Scheme '(a b . c) defines a concell with two cells a and b, c in the last object. A list in Scheme is merely a linked list with nodes featuring `car` and `cdr` items.

In the second example, a value is not being `returned`... it is a call to a continuation — a jump is immediately done out of the function and the entire code block is being made a value.

Exception Handling continued on next page...

C features `setjmp` and `longjmp` that provides a limited form of continuations. See slide.

Coroutines are *separate threads of execution* that **voluntarily** transfer control to each other (contrast this to threads which are completely independent units and switching between the threads is done by the operating system). These are useful in implementing iterators — see above.

Coroutines are "active" at the same time and therefore cannot use the same stack. Some notion of stack is required in order to allow recursion within coroutines and support lexical scoping. This means non-trivial runtime costs. The solution to this is a **Cactus stack**. See slide.

NOTE Coroutines can be created using continuations.

Data Types and Memory Management

Typing

A **type system** is a mechanism for defining types and *associating* them with operations that can be performed on objects of this type. This means one of two things: built-in types with built-in operations, or custom operations for built-in types and custom types.

A type system includes rules that specify:

- **type equivalence**: do two values have the same type? (structural equivalence *v.* name equivalence)
- **type compatibility**: can a value of a certain type be used in a certain context?
- **type inference**: how is the type of an expression computed from types of its part?

A pessimist would say that type errors are a nuisance — why do we not just get rid of types completely? Types provide security for a programmer and represents *intent*. Note that once a program is compiled in Haskell, there is a chance it will do the right thing if the type system is used to its effect (it is extremely strict but expressive).

Type systems can be **strongly typed**, **statically typed** (strongly typed and type checking is performed at compile time), or **dynamically typed** (types of operands of operations are checked at runtime). Most common high-level programming languages are pretty strongly typed (prohibits application of an operation to any object not supporting this operation).

For a scripting language, there is no reason to be dynamically typed; but why is it so common that interpreted languages are often dynamically typed and compiled languages are usually statically typed? When a compiler is built, the reason you compile is because you care about performance. Interpretation is usually about expressibility and faster development. Also, think about value *v.* reference model of variables. If different values can be stored in a given fixed-size location, almost immediately this forces us into a reference model of variables.

Similar to subroutines in many languages, defining a type has two parts: a type's **declaration** introduces its name into the current scope and and a type's **definition** describes the type (the simpler types it is composed of). Classifications of types include:

- **denotational**: a type is a set of values,
- **constructive**: a type is built-in or composite, and
- **abstraction-based**: a type is defined by an interface, the set of operations it supports.

Typing continued on next page...

Built-in types usually include integers, booleans, characters, real numbers, ... Enumeration and range types are somewhere in-between. Finally, composite types include **records**, arrays, files, lists, sets, pointers, ... Accessing elements of a record is usually done with `.` (dot) notation.

What is the memory layout of **records**? They can be one of three layouts:

- **aligned** (fixed ordering): this has a potential waste of space but allows for one machine operation per element access and a guaranteed layout in memory (good for systems programming).
- **packed**: no waste of space and a guaranteed layout as well, *but* multiple machine operations per memory access.
- **aligned** (optimized ordering): reduced space overhead (good and bad) and one machine operation per memory access, *but* has no guaranteed layout in memory.

Arrays are useful structures. Issues with them include memory allocation, bound checks, and index calculations (higher-dimension arrays). Depending on where when their shape is fixed or determined and the array type, where it is stored is determined: it can be static (stored in a static variable), a local variable of a function (while that function is active), or dynamic (created at some point and destroyed later).

This brings about an interesting problem. Efficient access to **stack-allocated objects** is based on every element having a fixed offset in the stack frame. How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time? The stack is split up into a **dynamic part** and **static part**. All an element of this sort in the static part would hold is a reference to the dynamic part.

But how do we allow index calculation and bound checking for such arrays (and heap-allocated arrays)? We introduce the use of a **dope vector** at the location of this static part: in this vector we have a pointer to the location as well as ranges for every dimension of the array.

2D arrays, in contiguous memory layout, feature **row-major layout** or **column-major layout**. Modern machines have caches (large blocks of consecutive memory cells or words) and in row-major layout, this is exactly the manner of which it is stored in caches. Therefore, column-major layout would be at least 1 magnitude of order slower unless a programmer goes column-by-column. Note that there are more sophisticated block-recursive layouts which, combined with the right algorithms, achieve much better cache efficiency than the above.

In old-style C, **row-pointer layout** was necessary to utilize multi-dimensional arrays. In terms of space-usage, the latter appears to be better... or is it? It depends on the memory layout. For efficiency, the arithmetic is easier for the latter but there may be two cache misses. This depends on the processor architecture.

Associative Arrays allow arbitrary elements as indices. These are directly supported in Perl and many other scripting languages. C++ and Java call them **maps** (they are not really arrays) and Scheme calls them **association lists** (A-lists) — they are implemented as a list of pairs. Efficiency of these structures differ depending on how they are implemented (they could be iterative or constant-lookup hash maps).

> **NOTE** Different lookup operations are present in Scheme for different notions of equality in association lists; the differences rely in difference operands for equality tests; `eq` is the most efficient but is limited (true if the exact same object). `equal` looks at the contents of the object (recursively, if necessary). `eqv` is somewhere in-between.

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists. Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays. There is a good reason for this.

Arrays are a natural way to store sequences when manipulating individual elements *in place* (imperatively change the content of a memory location). Functional arrays that allow updates without copying the entire array are non-trivial to implement (remember; no side effects are allowed). **Lists** are *naturally* recursive and fit into the recursive

Typing continued on next page...

approach taken to most problems in functional programming. Note that **strings** are typically arrays of *characters* in imperative languages, and lists of characters in functional languages.

References

Pointers point to memory locations that store data (often of a specified type, e.g. `int *`). They are generally not necessary in languages with reference model of variables (Lisp, ML, CLU, Java) but are required for recursive types in languages with value models of variables (C, Pascal, Ada).

Storage reclamation can be explicit (manual) or automatic (by garbage collection). The advantage of explicit reclamation is that garbage collection can incur serious run-time overhead (a challenge to do right); disadvantages are that there is a potential for memory leaks or for dangling pointers (cannot happen in languages that do not require explicit deallocation) and segmentation faults.

Pointer allocation and deallocation mechanisms differ between languages. C is not type-safe and has explicit deallocation. Pascal and C++ are type-safe and has explicit deallocation.

> **NOTE** The semantics for `p = new element()` differ between Java and C++; in Java, `p` is a type of `element`. In C++, `p` is a pointer to a something of type `element`.

A **dangling reference** is a *pointer* to an already reclaimed object. This can only happen when reclamation of objects that are no longer needed is the responsibility of the programmer. They are notoriously hard to debug and a major source of program misbehavior and security holes. The techniques utilized to catch them involve **tombstones** or **keys and locks**. The idea is to crash to expose the bug rather than do the wrong thing.

Tombstones have a fairly simple idea. Whenever a pointer is created to an object, you create a reference to a tombstone record before pointing to the actual object. Pointer assignment would happen at the tombstone level, therefore. Tombstones would store a special marker to "kill" itself and therefore if tried to access from a "dangling pointer", a crash will occur. This has issues:

- space overhead,
- runtime overhead (two cache misses instead of one),
- check for invalid tombstones is a hardware interrupt (cheap); RIP = null pointer, and
- how to allocate/deallocate the tombstones?
  - from separate heap (no fragmentation; each tombstone has the same size),
  - need reference count or other garbage collection strategy to determine when I can delete a tombstone,
  - need to track pointers to objects on the stack in order to invalidate their tombstones.

Why not implement a garbage collector for all objects? Very often, reference counts are actually good enough to implement garbage collectors (fairly limited; does not identify all sorts of garbage but good enough for tombstones). An interesting side effect is that there is an extra level of indirection that allows for **memory compaction**.

Locks and keys are similar. When allocating a new pointer or object, allocate slightly more memory to store a certain hash value which is stored somewhere in the object. Every pointer will store this same value to that object (think of this as a lock on the object and a pointer has a key to that lock). Deleting a pointer will clear that lock on the object so any other access will force an error.

References continued on next page...

Note that these can only be used for pointers to heap-allocated objects, provides only probabilistic protection, is unclear which one has higher run-time overhead (*v.* tombstones), and is unclear which one has a higher space overhead. Languages that provide for such checks often allow them to be turned on/off using compile-time flags.

NOTE In functional languages, order of evaluations can be done in whatever order is wanted. The last time a reference is used is only known when nothing points to an object anymore, in which case a garbage collector is needed to clean it up.

Garbage collection is an automatic reclamation of space. They are *essential* for functional languages, popular in imperative languages, but difficult to implement. It is also slower than manual reclamation. Methods include **reference counts**, **mark and sweep**, and **mark and sweep variants** (incl. stop and copy, generational technique).

Reference counts are fairly simple to implement; it has very little overhead (low cost). Issues include *subroutine returns*: a large number of variables are destroyed and therefore code must be generated to decrease the reference counts of all objects referenced by variables in the stack frame. This requires us to keep track of which entries in a stack frame are pointers. Not a very major issue. Note, furthermore, that these do not work when there are **circular references** (e.g., doubly linked lists) — there would be a failure of *not* reclaiming structures. This is not a problem in *purely functional languages* (an ideal): there is no way to generate a circular reference structure — every object, once created, is immutable and therefore a new copy has to be made to close a circle.

NOTE In Objective-C, when a reference is passed to someone else, either the object is temporarily used and the count is not increased, or it is held for a longer period of time and therefore increase its count. It is a "manual" way of keeping track of reference counts if explicit ownership is taken for an object. In circular structures, it is said the responsibility it is on the programmer to manage destroying those structures.

Mark and Sweep involves an algorithm that marks every allocated memory block as *useless*. For every pointer in the static address space, and on the stack, mark the block it points to as *useful*. For every block whose status changes from useless to useful, mark the blocks referenced by pointers in this block as useful — apply this rule recursively. In the end, all blocks marked as useless are reclaimed. This is effectively a **depth-first graph search**.

Note that this is *extremely expensive*, more complicated to implement, requires inspection of all allocated blocks in a sweep (in the heap), has high space usage if recursion is deep (a garbage collector is called usually when space is becoming limited so this is a problem — needs to be fixed), and requires *type descriptors* at the beginning of each block to know its size and to find pointers in the block (there may be a potential way to fix this). The only advantage is that this works with circular data structures.

MARK AND SWEEP IN CONSTANT WORKING SPACE. Whenever a recursive call is made, note, that a pointer is followed. Therefore, this pointer space can be used to store temporary data (e.g., it gives one pointer worth of working space to utilize at every pointer). So, at every pointer we can consider the previous and the current object by replacing the latter with the former at each location. When done, therefore, you return along the path to the origin and restore all pointers. This *takes care of the space issue* but which pointer you followed in the previous object does have to be remembered (store twice as many pointers).

How do we deal with looking at the entire heap and the issue of type descriptors? A variant, **stop and copy**, divides the heap into two halves. Allocation will happen in the first half, and once the first half is full, garbage collection is started along with **compaction**:

- find useful objects by following pointers as in standard mark-and-sweep but without first marking objects as useless,
- every useful object found is copied to the second half of the heap and replaced with a tag pointing to a new location,

References continued on next page...

- every subsequent pointer to this object finds the tag and gets replaced with a pointer to the new copy, and
- at the end, the roles of the two halves are swapped.

The advantages of this includes a time proportional to number of useful objects (not total number), it eliminates external fragmentation, and both an advantage and disadvantage is that only half of the heap is available for allocation (not really an issue with virtual memory).

The **generational technique** variant also divides the heap (but into two or more regions — often two). Allocation happens in the first region and garbage collection involves applying mark-and-sweep to the first region, promoting all objects that survive a small number (often one) rounds of garbage collection in a region to the next region in a manner reminiscent of stop-and-copy. Finally, subsequent regions are inspected only if collection in the regions inspected so far did not free up enough space.

The idea of this is the fact that *most objects are short-lived*. Thus, collection inspects *only* the first region most of the time and thus is cheaper. Every short-lived object surviving at least one sweep is considered long-lived. There are two main issues with this, however. How does one discover useful objects in a region without searching other regions? And how does one update pointers from older regions to younger regions when promoting objects? Techniques include:

- disallow pointers from old regions to young regions by moving objects around appropriately, and
- keep list of old-to-new pointers (requires instrumentation; this is runtime overhead).

What usually is being done is, like the second technique, keeping a **bit vector**: keeping track of columns (pages) in older regions that have objects pointing to younger regions. This is typically *implementation-specific*: garbage collectors can be implemented however one wants as long as byte codes are correctly interpreted.

NOTE The above is what the Java Virtual Machine does.

There still remains the question we asked earlier. How do we deal with type descriptors? Normally, garbage collectors need to know which positions in an object are pointers. A more *conservative* approach that does not need type descriptors involve **treating each word as pointers**. This may fail to reclaim useless objects because some integer or other object happens to equal the address of an object, but statistically *this is rare* (same idea as locks and keys).

As many objects are object-oriented, virtual methods become more common: they depend on the objects being called. Type descriptors, therefore, are usually present and so no additional cost is being forced. Therefore, the above is only really for purists.

Object-Orientation

Object-Oriented Programming

Elements of object-oriented programming include:

- data items to be manipulated as **objects**,
- objects are members of **classes**, that is, classes are types,
- objects store data in **fields** and behaviour in **methods** specified by their classes.

Main characteristics of most object-oriented programming systems involve:

- **encapsulation** by hiding internals,

- customization of behaviour through **inheritance**, and
- **polymorphism** through **dynamic method binding**.

The advantages of object-oriented programming include the fact that it **reduces conceptual load** (the amount of detail the programmer must think about at the same time), it provides **fault and change containment** (it limits the portion of a program to be looked at when debugging and the portion that needs to be changed when changing the behaviour of an object without changing its interface), and it provides **independence of program components** and thus **facilitates code reuse**. Most of these are ultimately consequences of encapsulation and thus also apply to programming using modules.

Object-oriented languages include: SIMULA, Smalltalk, C++, Modula-3, CLOS, Eiffel, Oberon, Java, and Ada. Most of them are similar to how C++ and Java do their object-orientation. Smalltalk considers all methods as messages. CLOS does not bundle information with data but polymorphic functions can be defined to do different things depending on objects called on (multiple function definitions). The advantage of this over Java-style object-orientation is that it allows greater versatility in having users define new methods for classes without modifying classes.

Class definitions typically involve private fields, public fields, constructors, destructors, public methods, and private methods. Definitions in C++ can be defined inside classes or outside of them (in which case the method definitions outside the class needs to be qualified using certain syntax) — allows the ability of providing an interface separate from implementation.

Using **inheritance** we can define a new **derived** or **child class** based on an existing **parent class** or **superclass**. The derived class inherits all fields and methods of the superclass, can define additional fields and methods, and can override existing fields and methods. The purpose of this is to extend or specialize the behaviour of the superclass. This allows us to define a **class hierarchy** where if only single inheritance is allowed, the hierarchy is a tree; if multiple inheritance is allowed, the hierarchy is a lattice.

NOTE Java allows only one parent class, but C++ allows multiple parents. Single inheritance tends to be cleaner and simpler. In serious applications of object-orientation, multiple inheritance is still extremely valuable.

NOTE A bad example for a syntax of inheritance in the textbook is `class queue : public list { ... }`. There is something fundamentally wrong about this conceptually. A queue is not conceptually a type of list. The right way of implementing this is that a queue does not have a parent class but internally has a list as a data member.

Overriding methods of a base class is relatively simple; to replace a method of a base class, it is redefined in the derived class. Methods of the base class are still accessible in the derived class by use of a keyword such as `super` or through some other technique such as scope resolution (any ancestor class can be accessed in this manner).

Visibility

Encapsulation in programming can be achieved by using **modules**:

- defining an **opaque** module type, a type whose definition is not exported by the module, and
- exporting subroutines to manipulate objects of the type — the implementation of these subroutines is not visible to the user.

Or using **classes**:

- **public** methods are accessible to the class's user and **private** methods are not,

Visibility continued on next page...

- private methods are accessible to other objects of the same class, and
- effective use of inheritance requires more fine-grained control over visibility of methods than sufficient when using modules.

In C++ there are three visibility levels: **private**, **protected**, and **public**. Private methods or fields are visible to members of the object of the same class and to friends (classes can explicitly declare other elements as friends). Protected methods or fields are visible to members of objects of the same class or derived classes and to friends. Finally, public methods or fields are visible to the whole world.

Furthermore, derived classes can restrict (but not increase) the visibility of its base class's members in objects of the derived class. See slide. Visibility can also be altered for individual members in so long as visibility is not increased.

This is not the same way all languages do this. Eiffel allows increasing and decreasing of visibility. Java is similar to C++ but there is no notion of friends, base classes are always public, and protected members are visible in derived classes and in the same package. Python has all class members being public. Finally, Smalltalk and Objective-C have all public methods and all private fields.

Classes

A **constructor** does not allocate the space for an object; it initializes or constructs the object in the allocated space. The execution order of constructors in C++ involves: executing the constructor(s) of base class(es), of class members, and then the class itself.

NOTE See that `y.f(4.5)` in the *Constructor and Method Overloading (1)* example calls method 3 because it is the only `const` method — the float is truncated to an integer. C's silent type-casting comes in and "tricks you". In example *(2)*, if a variable is being passed in, it could be used as a reference or as its value — an error will occur.

What goes on when object-oriented constructs occur and the associated costs — what happens in the sequence of constructing and assigning objects to other objects (see *Copy Constructors and Assignment*). In C++, a lot of transparency occurs; the introduction of constructors and operator overloading leads to lots of things happening *implicitly* — the way the semantics of the language are defined incurs one of two behaviours depending on the assignment.

A similar analysis applies to the next two definitions of class `A`: if the first notation is used (construct `A` then `x`), different behaviour will result.

NOTE Visibility rules come at 0-runtime cost. They can be checked at compile-time and are not referred to during runtime.

In languages with a reference model of variables or when using pointers in C++, we can use an object of a derived class where an object of a base class is expected. One thing that differentiates languages are whether methods are bound **statically** or **dynamically**. Assuming the derived class overrides a method of the base class, when accessing an object of the derived class through a variable whose type is the base class, which method should we call?

Static method binding involves a method that is invoked being determined by the type of the variable through the object by which it is accessed; languages that use this include Simula, C++, Ada 95. **Dynamic method binding** invokes methods by determination of the type of the accessed object. Languages using this include Smalltalk, Modula 3, Java, Eiffel.

Classes continued on next page...

Which is more efficient: static or dynamic method binding? It is more efficient to implement static method binding as that it is possible to be done at compile-time. In C++, dynamic method binding is available by declaring methods to be **virtual**. *Which is more natural*? Dynamic method binding is arguably more natural — the whole point of overriding in derived classes is to specialize the behaviour of those types of object.

An **abstract method** is a method that is *required* to be defined only in derived classes. An **abstract class** has at least one abstract method and thus *cannot* be instantiated. If all methods are abstract, then all the class does is define an **interface**.

NOTE In GTK, using C, function addresses are stored in the object in order to simulate virtual methods.

IMPLEMENTATION OF VIRTUAL METHODS. The **virtual method table** or **vtable** is an *array of addresses* of the virtual methods of the object, constructed at compile time. The overhead involved here are *two extra memory accesses*. The record of a derived class possesses an appending of extra data members to the record of the base class and provides trivial access to those members through pointers whose type is the base class. The vtable of a derived class would copy the vtable of the base class and replace entries of overridden virtual methods; entries are appended for those virtual methods declared in the derived class. There is a consequence to using virtual methods. Once something is virtual, it must remain virtual forever.

Inheritance

With single inheritance, some finer points of implementing type checking in regards to inheritance will involve checking to see if the above can be properly managed at compile time. See slide.

The second error is resolved by considering **type casting**. The implementation of **dynamic casts** involves including in each vtable the address of a run-time type descriptor.

In general, C++ has a slew of differing casting abilities: dynamic casts, static/reinterpret casts (conversions between unrelated types; neither is a derived or base type), and const casts (no type conversion other than removing the const-ness of a pointer).

Multiple inheritance allows a derived class to have multiple base classes.

```
class A : public B, public C { ...  }
```

Implementation issues include: how to access objects of A through a base class pointer (a pointer of type A will only see fields of A in a derived object), how to allow overriding of methods of different base classes. Semantic issues include: if a method `m` is defined in more than one base class, which method is invoked by `a.m()`? If B and C are derived classes of common base class D, does A have two or only one copy of each data member D? Rules have to be fixed on convention; this is arbitrary.

NOTE Textbook goes into more detail here.

NOTE Normally, an object will contain *one* vtable. In the case of multiple inheritance, multiple vtables are necessary (one per base class) and distributing parts amongst the object. This is typically extremely messy and why many languages do not support multiple inheritance.