Chapter 6 Control Flow

October 11, 2017

Expression evaluation

- It is common in programming languages to use the idea of an *expression*, which might be
  - a simple object
  - function invocation over some number of arguments (which usually can be expressions)
  - an operator over some number of operands (which are generally expressions) (also note that some languages treat operators as *syntactic sugar* for functions)

{in,pre,post}fix

- Infix just means that the function name appears among the arguments.
- Prefix means that the function name appears first followed by its arguments.
- Postfix means that arguments appear first and the function name last.

Expressions and typical languages

- Smalltalk is all infix
- Forth, PostScript, and some intermediate languages (PCode type, for example) use postfix.
- The Lisp family of languages are prefix
- Most other languages use a mix of prefix and infix notation.

Precedence and associativity

- Infix is inherently ambiguous with regard to precedence and associativity, and languages that allow infix notation have to have rules governing their use; rules can be simplistic (Pascal and Smalltalk, for instance), or can be quite elaborate (C is an excellent example.)

Assignment

- In imperative languages, assignment allows a given, named area of memory to have its value changed.
- Side effects come into play when an assignment makes some other state change than just changing the value of the variable being assigned to.
- Many programming languages differentiate between "expressions", which also produce a value and may or may not have side effects, and "statements", which exist to produce side effects and any return value is cordially ignored.
- Purely functional languages and side effects: Haskell really tries to get ride of side-effects, even with respect to IO (think about that for a moment: how do you structure IO as *not* a side-effect? The answer lies in (modern) monads — not the same as Leibniz's monads.)

References and Values

- l-values and r-values ($2 := a$; "2 is not an l-value!")
- Legal C (indeed, not uncommon)

```
(f(a)+3)->b[c] = 2;
```

- Subtleties of assigments to and from pointers.

Orthogonality and Algol 68

- Features can be used in any combination
- All combinations make sense
- The meaning of a given feature stays consistent no matter the context it is used.

Algol 68: it is just an expression

- See Expression-oriented programming languages

```
begin
    a := if b < c then d else e;
    a := begin f(b); g(c) end;
    g(d);
    2 + 3;
end
```

- evaluates to 5

C: close to the same

- Ternary '? :' construction
- The problematic "if (a = b)" issue

Combination assignment operators

- C: “+=”, “-=”, prefix “++”, prefix “–” (not postfix forms, though)
- Perl: “=~”

Multiway assignment

- Swap and multi-valued returns

```
a,b = b,a
a,b,c = func(d,e,f)
```

Initialization

- Static variables, local to a routine, need some sort of value
- Static variables can be preallocated by compiler (though you have to be careful to make sure that the actual memory section is writable at runtime)
- Initialization can help prevent using uninitialized variables, a common problem in some languages

Uninitialization

- Languages like Perl support the idea of being able to identify an uninitialized variable (undefined() function).

Constructors

- Common in object-oriented languages to allow constructors to initialize new objects

Ordering within expressions

- Common not to specify this since it lets the compiler choose the most optimal version
- So what do you about side effects?

Application of Mathematical Identities

- Consider

```
a = b + c
d = c + e + b
```

- Applying a bit of math, this can be reduced to

```
a = b + c
d = a + e
```

- Dangerous many times in programming languages because of precision issues.

Short-circuit evaluation

- Can be very efficient

```
if(x or y)
```

- if x is true, don't bother evaluating y

Short-circuit evaluation

```
if(x and y)
```

- if is false, don't bother evaluating y

C

```
if(p && p->value != something)
{
   p = p->next;
}
```

- Very common construction!

Pascal doesn't short-circuit

- Ends up writing more auxiliary variable code by hand

Short-circuits and side-effects

- What if you wanted a side-effect to happen from an evaluation that was short-circuited?

6.2 Structured and Unstructured Flow

- Assembly: jmp, branch, call all change the program counter PC; usual semantics are

```
jmp X    =>  PC := X
branch X =>  PC := PC + X
call X   =>  PUSH PC, PC := X
```

GOTO

- A Goto statement is pretty similar; name some line of code in some way, and then Goto that name.
- Dijkstra and the Goto statement
- How about “longjmp()”?

Perl's structured alternatives: next, last, redo

- next: start the next iteration of a loop
- last: immediate exit a loop
- redo: restart the loop block without evaluating any conditionals; to quote the man page "This command is normally used by programs that want to lie to themselves about what was just input."

C's continue

- skip the rest of this loop (like Perl's next)

Multi-level returns

- Can be done (Ruby actually likes these and supports quite sophisticated versions, even used in its own libraries), but unwinding (at the least) is always an issue when leaving multiple levels of invoked subroutines.

Errors and other exceptions

- For an interesting discussion on Haskell's philosphy, see Error versus Exception
- Structured exceptions are similar to multi-level returns and present similar implementation issues. (Try looking at the approachable (if dated) material here.)

Continuations

- C allows setjump()/longjmp() (and now also setcontext()/getcontext(), though I am not familiar with any code that actually uses the latter), which are like continuations with some limitations
- Scheme and Ruby allow first-class continuations; indeed, part of Scheme's reputation was built around continuations

6.3 Sequencing

- Compound statements : begin/end, { }; as mentioned, some languages where everything is an expression use that concept to give a value to statement blocks also, usually by adopting the evaluation of the final statement in the compound block
- Freedom from side-effects and idempotence: there are some advantages to being able to call a function twice with the same given set of arguments, and get the same result. Not only can we reason more clearly about the program, we can may also be to do optimizations that cannot happen with a function that has side-effects.
- Of course, sometimes side-effects are desirable – for instance, it's also nice to call a read_some_data_function() and get different data, or call a random_number_generator() and get a different answer.

6.4 Selection

- Most imperative languages support at least *if/then/else* type constructions (many in the C family also support ideas like the ternary operator).
- Declarative and functional languages also all support alternation facilities, but they can be expressed in a large number of ways, some of which are somewhat subtle.

6.4.1 Short-circuits

- Since we generally don't use the value of a boolean expression evaluated as part of an alternation or looping construction for anything other than governing the construction's execution, we can often "short-circuit" the evaluation. The text gives a particularly nice example in examples 6.46 and 6.47, where the expression

```
if ((A>B) and (C>D) or (E != F) then
  then-clause
else
  else-clause
```

6.4.1 Short-circuits

- Is "short-circuited" to

```
r1 := A
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1
```

(continued)

```
L4:
  r1:=E
  r2:=F
  if r1 == r2 goto L2
L1:
  then-clause
  goto L3
L2:
  else-clause
L3:
```

6.4.1 Short-circuits

- If you do need the value of the expression, then you can still use short-circuit code to generate it.
- One of the triumphs of programming language design, as noted in the footnote on page 250.

6.4.2 Case/Switch statements

- What if you have a large number of alternative cases? Using an extended if/then/else/elsif often is visually less than pleasing.
- How about a construction that expresses each case with a simplified syntax? You could call it “case”.
- That works out to be not only more elegant and expressive, but it also allows one to use fast alternatives such as a jump table.

6.4.2

- Alternatives include sequential testing (i.e., same as extended if/then/else), hashing, or binary search
- When to use an alternative: when your jump table is not dense. Example

```
case (X)
  1: do_something();
  200000: do_something_else();
  20000000000: do_other_stuff();
endcase
```

6.4.2

- C's "switch" statement (a type of case statement) has the ability to "fall through", which can be quite useful; look at the re2c code from the example at the end of the notes for chapter 4 – it uses fall through extensively.
- Even more dramatic use of "fall through": Duff's device.

6.5 Iteration

- A fundamental mechanism; you need iteration to make a computer actually useful (as to the alternative idea of recursion, you can regard actual recursion on von Neumann hardware to be an augmented instance of iteration.)

6.5.1 Enumeration-controlled loops

- Fortran's "do" loops; many other languages have corresponding "for" constructions.
- Some languages do not allow access to the value of the enumeration variable after the loop is finished (usually in such languages the header is considered to declare the index.)

6.5.2 Combination loops

- The C family (and Perl, for that matter) has the rather nice combination loop:

```
for(INITIALIZATION;
    BOOLEANEXPRESSION;
    ITERATIONSTEP)
```

6.5.5 Logically controlled loops

- while / until / do while
- Mid-loop testing: as mentioned previously, in Perl, you can have last, next, or even “redo”.

6.6.1 Iteration and recursion

- At a machine level, these are equivalent. Both are accomplished with an instruction that changes the program counter arbitrarily rather than just incrementing it; the recursive version also implicitly pushes the current program counter to a stack so that it can return to the next instruction simply by popping that value from the stack, something that can be done explicitly.

6.6.1 Iteration and recursion

- Tail recursion: can be optimized back to a straight jump instruction since there are no following instructions, and thus you don't need to push any address since you don't need to come back.

6.6.2 Applicative- and normal-order evaluation

- Applicative-order evaluation means that arguments are fully evaluated before being passed to a function.
- Normal-order evaluation means that arguments are only evaluated when their value is needed (you can regard it as lazy evaluation without the memoization). Lazy evaluation in Haskell is actually quite nice.

Chapter Summary

- Control flow can take many forms, but for a language to be Turing complete, it needs to be able to express alternation and iteration.
- We can have other ideas, such as exception handling and concurrency, and even nondeterminancy (also, you might look at Non-deterministic parallelism considered useful.)
- "Goto" is largely gone, and largely unlamented. However, in the end, what is actually running is built over actual "goto" (unlimited changes to the program counter.)