CS342: Organization of Prog. Languages

Topic 10: Control Flow

- Statements vs Expressions
- Flow of execution
- Contitional execution
- Loops
- Iterator objects and Generators

Order of Computation

- Some programming langs require **explicit control** of the order in which the parts of a program are evaluated/performed.
- Some programming langs evaluate the arguments of functions before calling the function (**eager evaluation**).
  Others defer the evaluation of arguments until their values are ultimately required (**lazy evaluation**).
- Some programming langs are inherently **serial**, others have support for **concurrent** execution.

Statements vs Expressions

- Some imperative languages, like C, are *statement*-based.

  A program consists of a series of commands, and the computation is viewed as the *execution* of statements for their side effects.

  The the execution order of the statements is called the *control flow*.

- Some imperative languages, like Lisp, are A program consists of an expression. Computation consists of the *evaluation* of the expression for its resulting value (and possibly side effects).

  Evaluation of the expression consists of evaluating sub-expressions and using their values to determine the value of the entire expression.

  The the evaluation order of subexpressions is the *control flow* in this case.

- Statement-based languages often have an expression-based sub-language to compute values.

  We will talk about statements and expressions interchangeably since the ideas of control flow apply equally to both.

Examples

- In C, some expressions/statements have defined order of evaluation:

  ```
  (i < n) && a[i] != 0

  (a < 0) ? f(a) : g(a)

  for (i = 0; i < n; i++) f(i);
  ```

- For others, the order of evaluation is undefined:

  ```
  h(i++, i++)

  (i < n) & a[i] != 0
  ```

Sequential Evaluation

- C:

```
{ s1; s2; s3; }
```

- Scheme:

```
(lambda varlist e1 e2 ...)
```

  and derived forms

```
(begin e1 e2 e3)
```

- Pascal:

```
begin s1; s2; s3 end
```

Boolean expression evaluation

- Many imperative languages have boolean operations which do not evaluate their 2nd (etc) arguments unless they have to.
- This uses the properties: $\forall x.\ \mathsf{F}\ \textbf{and}\ x = \mathsf{F}$ $\forall x.\ \mathsf{T}\ \textbf{or}\ x = \mathsf{T}$
- C: `a && b` `a || b`
- Scheme: `(and e1 e2 ...)` `(or e1 e2 ...)`
- Ada: `a and then b` `a or else b`
- This saves computation, and simplifies program text when evaluation of both arguments could generate an error.

```
if (i < n && a[i] != 0) { STUFF }

if (i < n) {
        if (a[i] != 0) { STUFF }
}
```

Conditional evaluation – If

- **If-then** in various syntactic forms:

  `if` (*expr*) *stat*

  `if` *expr* `then` *stat*

  `if` *expr* `then` *stat* `endif`

  `if` *expr* `then` *stat* `fi`

  (`if` *expr* *expr*)

  ...

- **If-then-else** in its various syntactic forms:

  `if` (*expr*) *stat* `else` *stat*

  `if` *expr* `then` *stat* `else` *stat*

  `if` *expr* `then` *stat* `else` *stat* `endif`

  (`if` *expr* *expr* *expr*)

  expr `?` *expr* `:` *expr*

  ...

The syntactic form with a closing keyword avoids the “dangling else” problem:

```
if expr then
   if expr then stat
else
   stat
```

Conditional evaluation (cont'd)

- **Multi-way if-then-else.**

  This syntactic sugar helps to write programs without deeply nested if-then-else-s.

  if *expr* **then** *stat*
  elseif *expr* **then** *stat*
  ...
  elseif *expr* **then** *stat*
  else *stat*
  endif

  c1 ? *e1* : *c2* ? *e2* : *c3* ? *e3* : *e4*

  (**cond** (*c1 e11* ...) (*c2 e21* ...) ...)

  { *c1* => *e1*; *c2* => *e2*; ... ; *eN* }

Conditional evaluation (cont'd)

- **Arithmetic if** [Fortran]

  Goto one of three labels according as an expression is -ve, zero, +ve.

  `if` (*expr*) *L1*, *L2*, *L3*

  Now deprecated.

  This allowed programmers to do decide among three cases with one test.

Conditional evaluation (cont'd)

- **Conditional sequence exit** [Axiom]

  This generalizes the multi-way if-then-else to allow computations before the tests:

  ```
  r := { e1; c2 => e2; e3; c4 => e4; e5; e6}
  ```

  This is a syntactic sugar for

  ```
  r := {
          e1;
          if c2 then e2 else {
                  e3;
                  if c4 then e4 else { e5; e6 }
          }
  }
  ```

  This can be done in Scheme as

  ```
  (set! r (if (begin e1 c2) e2 (if (begin e3 c4) e4 (begin e5 e6))))
  ```

Conditional evaluation (cont'd)

- **Nondeterministic if**

  Each of the alternatives is a "guarded statement."

  Of all the branches for which the condition is true, choose one at random and execute the statement.

  if $c_1 \longrightarrow s_1$
  [] $c_2 \longrightarrow s_2$
  ...
  [] $c_n \longrightarrow s_n$
  fi

Iteration – Loops

- **Loop with boolean test at beginning**

  Each iteration test condition.
  Depending on value, either execute body or end loop.

  while *expr* **do** *body* **od**
  while (*expr*) *body*

  Evaluates body zero or more times.

- **Loop with boolean test at end**

  Each iteration execute body, then test condition.
  Depending ofn value, either rexecute the body or end the loop.

  do *body* **while** (*expr*)
  repeat *body* **until** *expr*

  Evaluates the body one or more times.

  Useful when the action of the body sets up values to test in the condition.

- Depending on the language, either of these kinds of loop may use a positive or negative end test. Often this is indicated by a keyword `while` or a keyword `until`.

  NB An `until` keyword does not guarantee the test is at the end. Check the definition of the language.

Iteration (cont'd)

- **Nondeterministic do**

  do $c_1 \rightarrow s_1$
  [] $c_2 \rightarrow s_2$
  ...
  [] $c_n \rightarrow s_n$
  od

  At each iteration, randomly choose one of the s_i for which the c_i is true and evaluate it. If none of the c_i are true, then the loop terminates.

Iteration (cont'd)

- **Discrete steps** integral or enumerated values.

```
do i = 1,100
   ...
end do

do i = 1, 100, 2
   ...
end do

do i = 100, 1, -2
   ...
end do
```

If the step direction is not determinable at compile time, then an extra the end test has to be decided to be "`i < limit`" or "`i > limit`" at run time.

For this reason some languages have two forms:

`for` *var* `:=` *start* `to` *limit* `by` *step* `do ...`

`for` *var* `:=` *start* `downto` *limit* `by` *step* `do ...`

This simplifies the code executed, and it can be somewhat more efficient.

Iteration (cont'd)

- **Elements of a structure**

  Some programming languages have primitive iteration over the elements of a homogeneous aggregate.

```
numlist := [1, 2, 99, 88, 23];

for n in numlist do ... od
```

Iteration (cont'd)

- **Programmer-defined stepping**

  E.g. in C,

  `for (` *init* `;` *test* `;` *step* `) ...`

  E.g. in Scheme,

```
(do ((var1 init1 step1) ...)
    (test fini1 ...)
     body1 ...  )
```

Programmer-defined **iterator objects** can provide an interface used generically in these kinds of loops:

```
//
// A class to specify generic iteration.
//
template<class T> class Iter {
public:
    virtual int    isEmpty() = 0;
    virtual void   step ()   = 0;
    virtual T      value()   = 0;
};
```

```
//
// Iteration over arrays
//
template<class T>
class ArrayIter : public Iter<T> {
    T    *_a;
    int  _i, _end;
public:
    ArrayIter(T *a, int start, int end)
        : _a(a), _i(start), _end(end) { }

    int    isEmpty() { return _i > _end; }
    void   step()    { _i++; }
    T      value()   { return _a[_i]; }
};
```

```
//
// Linked lists and their iterator
//
template<class T> struct Cons {
    T       _car;
    Cons *  _cdr;
    Cons(T a, Cons *d) : _car(a), _cdr(d) { };
};


template<class T>
class ConsIter : public Iter<T> {
    Cons<T> *  _cons;
public:
    ConsIter(Cons<T> *pcons) : _cons(pcons) {};

    int     isEmpty() { return _cons == 0; }
    void    step()    { _cons = _cons->_cdr; }
    T       value()   { return _cons->_car; }
};
```

```
//
// A function which uses generic iteration.
//

#include <iostream.h>

template<class T>
void looper(Iter<T>& iter)
{
    int i = 1;

    for ( ; !iter.isEmpty(); iter.step())
        cout << i++
             << " => "
             << iter.value()
             << "\n";
}
```

```
//
// Examples using the generic iteration.
//
#define N 10

int main(int argc, char **argv) {

    // Set up two data structures
    Cons<int>  *list = 0;
    int        i, p, nums[N];

    for (p = i = 1; i < N; p *= i++) {
        list = new Cons<int>(p, list);
        nums[i] = p;
    }

    // Create iterator objects
    ArrayIter<int>  niter(nums, 1, N-1);
    ConsIter <int>  citer(list);

    // Iterate over them
    looper(niter);
    looper(citer);
    return 0;
}
```

Iteration (cont'd)

- **Programmer-defined generators**

  The previous (iterator) approach required the programmer to keep track of the current state of a structure's traversal in an auxilliary object.

  Instead, some languages allow the programmer to keep track of this in the state of a traversal program.

  They define the notion of a generator, a program which can yield values to the calling program, and then be resumed on the next iteration.

```
values(a: Array DoubleFloat) == generate {
    -- Any pgm can go here, with yields anywhere.
    i: SingleInteger := 0;
    while i < #a repeat {
        yield a.i;
        i := i + 1;
    }
}
a0: Array DoubleFloat := [1.414, 2.718, 3.1416];

// The "for x in ..."  iterates over a generator.
for x in values a0 repeat { ... }
```

Iteration (cont'd)

- With user-defined generators, a useful notion is that of parallel iteration:

```
dot(u, v) == {
    sum := 0;
    for eu in u  for ev in v repeat
        sum := sum + eu*ev;
    sum
}
```

  Each iteration both variables receive new values.

  This is very different than the two nested loops

```
dot(u, v) == {
    sum := 0;
    for eu in u repeat
        for ev in v repeat
            sum := sum + eu*ev;
    sum
}
```

  Here ev takes on all possible values for each value of eu.

Iteration (cont'd)

- **Early exit**. Many languages have a notion of a statement which causes the enclosing loop to terminate (`break` in C, `exit` in Ada).

  Some languages allow this statement to take a numeric argument, which is the number of levels of loop to break out of.

  Others allow the statement to refer to a loop label to indicate which loop shall be terminated.

- **Skip to next iteration**. Many languages have a statement which causes the enclosing loop to quit executing the current loop body iteration and skip to the of the next iteration (`continue` in C, `iterate` in Aldor).

Iteration (cont'd)

- **Collection vs Looping**.

  In an expression based language, there is the question of the value of the loop.

  There are three possible answers:

  Nothing.

  The *last* body value.

  The values of *all* executions of the body.

- A **collect** form gathers the values of all the bodies. E.g.

```
lsq := [i^2 for i in 1..10];
```

```
tot := sum(3*k + 1 for k in 1..100);
```