Review

Last week

- Names and Bindings
- Lifetimes and Allocation
- Garbage Collection
- Scope
Outline

- Expression Evaluation
- Control Structures
- Selection
- Loops

Sources:

PLP, 6.1 - 6.5
Barrett. Lecture notes, Fall 2008.
Gottlieb. Lecture notes, Fall 2009.
Expression Evaluation

Languages may use various notation:

- **prefix**: $(+\ 1\ 2)$ – Scheme
- **postfix**: $0\ 0\ \text{moveto}$ – Postscript
- **infix**: $1\ +\ 2$ – C/C++, Java

Infix notation leads to some ambiguity:

- **associativity**: how operators of the same precedence are grouped
  
  - $x + y - z = (x + y) - z$ or $x + (y - z)$?

- **precedence**: the order in which operators are applied
  
  - $x + y * z = (x + y) * z$ or $x + (y * z)$?
Assignments

- *Assignment* statements are an example of side effects. Side effects change the behavior of subsequent statements and expressions.

- There is a difference between the container for a value ("memory location") and the value itself.
  - *l-value* refers to the locations. (They are on the left hand side.)
  - *r-value* refers to the values.
    - $3 = x + 1$ – Illegal! ”3” Can’t be an l-value
    - $x = x + 1$ – x is both an l-value and an r-value

- Imperative languages rely on side effects
  - Some languages introduced assignment operators.
    - Consider $a[f(i)] += 4$
      - More convenient than $a[f(i)] = a[f(i)] + 4$
      - Ensures that $f(i)$ is evaluated once

- Some languages allow multiway assignment:
  - $a, b, c = getabc()$ – Python, Perl
Initialization

- Assignment statements provide a way to set a value of a variable.
- Language may not provide a way to specify an initial value. This can lead to bugs.
- Some languages provide default initialization.
  - C initializes external variables to zero
- System may check dynamically if a variable is uninitialized
  - IEEE floating point uses special bit pattern (NaN)
  - Requires hardware support and expensive software checking
- Compiler may statically check – Java, C#
  - May be overly conservative
- OO-languages use constructors to initialize dynamically allocated variables
Structured vs. Unstructured Flow

- Early languages relied heavily on *unstructured* flow, especially `goto`’s.
- Common uses of `goto` have been captured by *structured* control statements.
  - Fortran had a `DO` loop, but no way to exit early except `goto`
  - C uses `break` for that purpose
The Infamous goto

- In machine language, there are no if statements or loops.
- We only have branches, which can be either unconditional or conditional (on a very simple condition).
- With this, we can implement loops, if statements, and case statements. In fact, we only need
  1. increment
  2. decrement
  3. branch on zero

to build a universal machine (one that is Turing complete).
- We don’t do this in high-level languages because unstructured use of the goto can lead to confusing programs. See “Go To Statement Considered Harmful” by Edgar Dijkstra.
Control Structures

A control structure is any mechanism that departs from the default of straight-line execution.

- selection
  - if statements
  - case statements

- iteration
  - while loops (unbounded)
  - for loops
  - iteration over collections

- other
  - goto
  - call/return
  - exceptions
  - continuations
Many languages provide a way to group several statements together:

- **PASCAL** introduces the begin-end pair to mark sequences.
- C/C++/JAVA abbreviate keywords to `{ }`.
- **ADA** dispenses with brackets for sequences, because keywords for the enclosing control structure are sufficient.

```
for J in 1..N loop ... end loop
```

- More writing but more readable.

- Another possibility – make indentation significant (e.g., ABC, PYTHON, HASKELL).
Selection

- **if** Condition **then** Statement – PASCAL, ADA
- **if** (Condition) **Statement** – C/C++, JAVA

- To avoid ambiguities, use end marker: **end if**, “}”

- To deal with multiple alternatives, use keyword or bracketing:

  ```
  if Condition **then**
  Statements
  elsif Condition **then**
  Statements
  else
  Statements
  end if;
  ```
Nesting

The infamous “dangling else” problem:

```plaintext
if Condition1 then
    if Condition2 then
        Statements1
    else
        Statements2
end if;
else
    Statements2
end if;
```

The solution is to use end markers. In Ada:

```plaintext
if Condition1 then
    if Condition2 then
        Statements1
    end if;
else
    Statements2
end if;
```
Short-circuit evaluation

```c
if (x/y > 5) { z = ... } // what if y == 0?
if (y == 0 || x/y > 5) { z = ... }
```

But binary operators normally evaluate both arguments. Solutions:

- a lazy evaluation rule for logical operators (LISP, C)

  ```c
  C1 && C2 // don’t evaluate C2 if C1 is false
  C1 || C2 // don’t evaluate C2 if C1 is true
  ```

- a control structure with a different syntax (ADA)

  ```c
  if C1 and then C2 then -- if C1 is false
  if C1 or else C2 then -- if C1 is true
  ```
Multiway selection

Case statement needed when there are many possibilities “at the same logical level” (i.e. depending on the same condition)

```vhdl
case Next_Char is
  when 'I' => Val := 1;
  when 'V' => Val := 5;
  when 'X' => Val := 10;
  when 'C' => Val := 100;
  when 'D' => Val := 500;
  when 'M' => Val := 1000;
  when others => raise Illegal_Roman_Numeral;
end case;
```

Can be simulated by sequence of if-statements, but logic is obscured.
The **Ada** case statement

- no flow-through (unlike C/C++)

- all possible choices are covered
  - mechanism to specify default action for choices not given explicitly

- no inaccessible branches:
  - no duplicate choices (C/C++, **Ada**, **Java**)

- choices must be static (**Ada**, C/C++, **Java**, ML)

- in many languages, type of expression must be discrete (e.g. no floating point)
Implementation of case

A possible implementation for C/C++/JAVA/ADA style case:

(If we have a finite set of possibilities, and the choices are computable at compile-time.)

- build table of addresses, one for each choice
- compute value
- transform into table index
- get table element at index and branch to that address
- execute
- branch to end of case statement

This is not the typical implementation for a ML/HASKELL style case.
Complications

case (x+1) is
  when integer’first..0 ⇒ Put_Line ("negative");
  when 1 ⇒ Put_Line ("unit");
  when 3 | 5 | 7 | 11 ⇒ Put_Line ("small_prime");
  when 2 | 4 | 6 | 8 | 10 ⇒ Put_Line ("small_even");
  when 21 ⇒ Put_Line ("house_wins");
  when 12..20 | 22..99 ⇒ Put_Line ("manageable");
  when others ⇒ Put_Line ("irrelevant");
end case;

Implementation would be a combination of tables and if statements.
void send (int *to, int *from, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
        case 0: do {
            *to++ = *from++;
        } while (--n > 0);
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    }
}
Indefinite loops

- All loops can be expressed as while-loops
  - good for invariant/assertion reasoning
- condition evaluated at each iteration
- if condition initially false, loop is never executed

```plaintext
while condition loop ... end loop;
```

is equivalent to

```plaintext
if condition then
    while condition loop ... end loop;
end if;
```

if `condition` has no side-effects
Executing while at least once

Sometimes we want to check condition at end instead of at beginning; this will guarantee loop is executed at least once.

- **repeat ... until** condition; (PASCAL)
- **do { ... } while** (condition); (C)

while form is most common can be simulated by while + a boolean variable:

```plaintext
first := True;
while (first or else condition) loop
  ...
  first := False;
end loop;
```
Breaking out

A more common need is to be able to break out of the loop in the middle of an iteration.

- **break** (C/C++, JAVA)
- **last** (PERL)
- **exit** (ADA)

```plaintext
loop
    ... part A ...
    exit when condition;
    ... part B ...
end loop;
```
Breaking way out

Sometimes, we want to break out of several levels of a nested loop

- give names to loops (ADA, PERL)
- use a goto (C/C++)
- use a break + lable (JAVA)

Outer: while C1 loop ...
  Inner: while C2 loop ...
    Innermost: while C3 loop ...
      exit Outer when Major_Failure;
      exit Inner when Small_Annoyance;
      ...
      end loop Innermost;
    end loop Inner;
  end loop Outer;
Definite Loops

Counting loops are iterators over discrete domains:

- `for J in 1..10 loop ... end loop;`
- `for (int i = 0; i < n; i++) { ... }`

Design issues:

- evaluation of bounds (only once, since ALGOL 60)
- scope of loop variable
- empty loops
- increments other than 1
- backwards iteration
- non-numeric domains
Evaluation of bounds

for J in 1..N loop
  ...
  N := N + 1;
end loop;  -- terminates?

Yes – in ADA, bounds are evaluated once before iteration starts. Note: the above loop uses abominable style. C/C++/JAVA loop has hybrid semantics:

for (int j = 0; j < last; j++) {
  ...
  last++;  -- terminates?
}

No – the condition “j < last” is evaluated at the end of each iteration.
The loop variable

- is it mutable?
- what is its scope? (i.e. local to loop?)

Constant and local is a better choice:

- **constant**: disallows changes to the variable, which can affect the loop execution and be confusing
- **local**: don’t need to worry about value of variable after loop exits

```plaintext
Count: integer := 17;
...
for Count in 1..10 loop
  ...
end loop;
...
-- Count is still 17
```
Different increments

**ALGOL 60:**

```plaintext
  for j from exp1 to exp2 by exp3 do ... 
```

- too rich for most cases; typically, `exp3` is +1 or -1.
- what are semantics if `exp1 > exp2` and `exp3 < 0`?

**C/C++:**

```plaintext
  for (int j = exp1; j <= exp2; j += exp3) ... 
```

**ADA:**

```plaintext
  for J in 1..N loop ... 
  for J in reverse 1..N loop ... 
```

Everything else can be programmed with a while loop
Non-numeric domains

ADA form generalizes to discrete types:

    for M in months loop ... end loop;

Basic pattern on other data types:

- define primitive operations: first, next, more_elements
- implement for loop as:

    iterator = Collection.Iterate();
    element thing = iterator.first;
    for (element thing = iterator.first;
         iterator.more_elements();
         thing = iterator.next()) {
        ...
    }
List Comprehensions

- **PYTHON** calls them “generator expressions”
- Concise syntax for generating lists
- Example:

```python
l = [1,2,3,4]
t = 'a', 'b'
c1 = [x for x in l if x % 2 == 0]
c2 = [(x,y) for x in l if x < 3 for y in t]
print str(c1)  # [2,4]
print str(c2)  # [(1, 'a'),(1, 'b'),(2, 'a'),(2, 'b')]
```

- Shorthand for:

```python
c2 = []
for x in l:
    if x < 3:
        for y in t:
            c2.append((x,y))
```