This week’s lab: Expressions

- Inspired by familiar mathematical notation
- Usually have recursive (tree-like) structure
- Can be used to define values in many domains
  - numbers, booleans, strings, lists, sets, etc.
- “Declarative” syntax: tells what to compute rather than how
- Abstracts away from evaluation order* and use of temporaries
  - compare with, e.g., stack machine

* to some extent: depends on language
Imperative Languages

- Most commonly-used languages are imperative

- Consist of sequence of commands that alter the state of the world

- State = values of program variables and external environment (e.g. files, screen, etc.)

Running Imperative Programs

- High-level imperative languages mimic style of the underlying Von Neumann machine architecture
  - Machine programs are sequences of instructions that modify registers and memory locations

- Compiling imperative languages to machine code is relatively straightforward
  - Variables are mapped to machine locations
  - Commands (operations) are mapped to (multiple) machine instructions
Reactive Programs

- Imperative languages are also natural for writing reactive programs that interact with the real world.

Examples:
- Reading mouse clicks and modifying the contents of a display.
- Communicating data on a network link.
- Controlling a set of sensors and relays in an external device.
- Often structured as event-response loops.
Statements are Commands

- Elementary (atomic) statements
  - Assignment
  - I/O operations
  - Function/Procedure calls
    - Atomic from perspective of caller
- Compound statements
  - Built recursively from sub-statements, forming tree-like structure
Assignment

- Most primitive command: store a value into a location

- In simplest form, location is associated with a variable but might be an array or record element, etc.

- In most languages, a variable name means different things on the left-hand side (LHS) and right-hand side (RHS) of an assignment.

  - On LHS, name denotes the location of the variable, into which the value of the RHS expression is to be stored. Here we say name is an l-value.

  - On RHS, name denotes the current value contained in the location, i.e. it indicates an implicit dereference operation. Here we say the name is an r-value.
Assignment Expressions

**b := (a := 42)**  **f(c := 10)**

- In some languages, assignment is an expression
- and expressions can act as atomic statements
- But every expression must define a value! Common choices for the value of an assignment:
  - value of LHS after assignment
  - special “no information” value e.g., in Scala: `():Unit`
- C/C++/Java popularized use of plain `=` for assignment and `==` for relational equality: a truly bad idea, because both form expressions and they are easy to confuse
We’ve noted that order of operations for expressions is usually under-specified.

Parse tree doesn’t completely fix order.

But this causes problems if expressions can be assignments:

```
a = 0;
b = (a = a + 1) - (a = a + 2);
```

What is the result in b?

It can be anything! This C program has “undefined behavior” and the compiler can generate anything it wants (for the entire program!)

...or the compiler could give a warning or error message, but many compilers do not.
Hidden side-effects

- Even without explicit assignment expressions, expression evaluation order can affect behavior:

```c
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2)); // = 0 or 2 ?
```

- Answer depends on evaluation order for function parameters, which is compiler-dependent (though “unspecified” rather than “undefined”).

- This flexibility may let compiler generate more efficient code.

- But most modern languages are moving towards precise specification of order (e.g. left-to-right).
Imperative code is infectious

Root of problem is that imperative code can be hidden within function definitions ("side-effects")

If any part of the code might be imperative, we must worry about order of evaluation in all parts of the code

In a few languages, the type system helps us distinguish functions that have side-effects from "pure" ones that don't

```c
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
int result = h(a,f(2)); // = 0 or 2 ??
```
Structured Control Flow

- All modern higher-level imperative languages are designed to support [structured programming](#).

- Syntactic structure of program text corresponds to [dynamic flow of control](#) during execution.

- Originally proposed as improvement over unreadable "spaghetti code" that is easy to produce using labels and jumps.

Small set of statement kinds

- Use small collection of (recursively defined) **compound statements** to describe control flow

- **Sequential composition**: do a sequence of commands
  
  \[
  \begin{align*}
  \text{(Java)} & \quad \{ \ x = 2; \ y = x + 4; \} \\
  \text{(Pascal)} & \quad \text{begin } x := 2; \ y := x + 4; \text{ end}
  \end{align*}
  \]

- **Selection**: do one of several alternative commands
  
  \[
  \begin{align*}
  \text{(Java)} & \quad \text{if } (x < 0) \ y = x + 1; \ \text{else } z = y + 2; \\
  \end{align*}
  \]

- **Iteration**: do a command repeatedly
  
  \[
  \begin{align*}
  \text{(Java)} & \quad \text{while } (x > 10) \ \text{output}(x--); \\
  \text{(Pascal)} & \quad \text{for } x := 1 \ \text{to } 12 \ \text{do } \text{output}(x*2);
  \end{align*}
  \]
Sequential composition

- Simplest way to combine commands: just write one after another

- Order obviously matters!

- (What about parallel composition?)

- Can also have sequential composition of expressions

  - $e_1; e_2$ means: evaluate $e_1$; throw away the result; then evaluate $e_2$

  - Obviously only interesting if $e_1$ has side-effects
Selection: if

Basic selection statement based on booleans

\[
\text{if } e \text{ then } s_1 \text{ else } s_2
\]

compiles to

\[
\begin{align*}
\text{evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq } l_1 \\
\text{s}_1 \\
\text{br } l_2 \\
\text{l}_1: \quad s_2 \\
\text{l}_2:
\end{align*}
\]

Structured statements have simple equivalents in terms of labels + jumps

pseudo assembly code
Selection: case

- Generalizes boolean conditionals to types with larger domains

```plaintext
case e of
  c_1 : s_1
  c_2 : s_2
  ...
  c_n : s_n
  default : s_d
```

- Note that the $c_i$ are constants

- Choice of most efficient compilation method depends on density of the $c_i$ within the domain of possible values for $e$ and on whether $e$’s type is ordered
Sparse case compilation

\[
\text{case } e \text{ of } \\
\begin{align*}
    c_1 & : s_1 \\
    c_2 & : s_2 \\
    \cdots & \\
    c_n & : s_n \\
    \text{default} & : s_d
\end{align*}
\]

is equivalent to

\[
\begin{align*}
t & := e; \\
    \text{if } t = c_1 \text{ then } \\
    & \quad s_1 \\
    \text{else if } t = c_2 \text{ then } \\
    & \quad s_2 \\
    \text{else} \\
    & \quad \cdots \\
    \text{else if } t = c_n \text{ then } \\
    & \quad s_n \\
    \text{else} \\
    & \quad s_d
\end{align*}
\]

- This is just a linear search (O(n) time)
- If e’s type is ordered, we can do better with a binary search (O(log n) time)
Dense case compilation

If labels are dense in the range \([c_1, c_n]\), it’s better to use a jump table (O(1) time):

```haskell
case e of
  c_1 : s_1
  c_2 : s_2
  ...
  c_n : s_n
  default : s_d
```

compiles to

```assembly
evaluate e into t
  cmp t, c_1
  brlt l_d
  cmp t, c_n
  brgt l_d
  sub t, c_1, t
  add table, t, t
  br *t

table: l_1
  l_2
  ...
  l_n

done:
  l_1: s_1
  l_2: s_2
  ...
  l_n: s_n
  l_d: s_d
```

The most efficient translation of `case` statements depends on the density of the value \(c_1, c_2, \ldots, c_n\) within the range of possible values for \(e\).
Iteration: while and repeat

while \( e \) do \( s \) compiles to

\[
\begin{align*}
top: & \quad \text{evaluate } e \text{ into } t \\
& \quad \text{cmp } t, \text{true} \\
& \quad \text{brneq done} \\
& \quad \text{br top} \\
\text{done:}
\end{align*}
\]

repeat \( s \) until \( e \) is equivalent to

\[
\begin{align*}
s; & \quad \text{while not } e \text{ do } s
\end{align*}
\]
Counted loops

Since iterating through a range of numbers is very common, many languages offer a dedicated statement, e.g. 

\[
\text{for } i := e_1 \text{ to } e_2 \text{ do } s
\]

The detailed semantics vary, and can be tricky (e.g. can \( s \) change \( i \)?)

C/C++/Java offer a more general-purpose statement

\[
\text{for } (e_1; e_2; e_3) s;
\]

is equivalent to

\[
e_1; \text{ while } (e_2) \{ s; e_3 \}
\]
Data-driven Iteration

- Many modern languages support generalized for loops that can iterate through any collection

```
val s = List(1,3,42,7)
for (v <- s)
  print(v+1) // prints 2 4 43 8
```

- In some languages this is implemented using iterators -- data objects that keep a pointer ("cursor") into the collection that can be advanced one element at a time

- Code above is shorthand for this:

```
val iter = s.iterator
while (iter.hasNext())
  print(iter.next()+1)
```

almost Scala
Loop exits

It can be useful to break out of the middle of a loop

```plaintext
loop
  s1;
  exitif e;
  s2
end
```

compiles to

```plaintext
top: s1
  evaluate e into t
  cmp t,true
  breq done
  s2
  br top
done:
```

C/C++/Java `break` is unconditional form of `exit`

These languages also have a `continue` statement that jumps back to the top of the loop
Uses for goto?

An efficient program using goto

```c
int i;
for (i = 0; i < n; i++)
    if (a[i] == k)
        goto found;

n++;
a[i] = k;
b[i] = 0;

found:
b[i]++;
```

In most languages, there is no equivalently efficient program without goto: must add a flag variable
Multi-level break

But we *can* do as well in Java (or JavaScript, Go, ...), using a named, multi-level break statement:

```java
int i;
search:
{ for (i = 0; i < n; i++)
    if (a[i] == k)
        break search;
    n++;
    a[i] = k;
    b[i] = 0;
}
b[i]++;
```

This construct was invented by Don Knuth in the 1960’s but not adopted into a mainstream language for 30 years!
The COME FROM statement

10 J = 1
11 COME FROM 20
12 PRINT J
   STOP
13 COME FROM 10
20 J = J + 2


- A notorious joke!

- But with a serious point: even with an ordinary GOTO, we must examine the whole label/branch structure of the program to understand its behavior.