Review: Expressions

- Usually tree-structured

- Abstract away from evaluation order* and use of temporaries
  - compare with e.g., stack machine

- Can be defined over many value domains
  - numbers, booleans, strings, lists, sets, etc.

- May be undefined on some dynamic values
  - consider division by zero
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
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*.

- Examples:
  
  ```plaintext
  a := 42
  a[\text{x+2}] := 42
  a := a + 5
  ```
Initialization Values

Many languages require variables to be declared before they are used: this gives them a scope, perhaps a type, and (maybe) an initial value given by an expression.

It is surely a bug to use any variable as an r-value unless it has been previously assigned a value.

But many languages let us write such code, resulting in runtime errors—either checked (e.g. as in Python) or unchecked (e.g. as in C).

Simplest fix is to require an initial value to be given for every declared variable (e.g. as in Scala).
Assignment Expressions

In some languages, assignment is an expression

But every expression must produce a value! Common choices:

- value of RHS

- 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 are expressions and are easy to confuse
Order of Operations

- 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:

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

  ANSI C99

- What is the result in `b`?

- It can be anything! This 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 actual parameters, which is language-dependent (and possibly unspecified).
- 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.

May explore this more later on in the course.

```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 ??
```
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.


blogbv2.altervista.org
Small set of statement kinds

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

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

- **Selection**: do one of several alternative commands
  
  (Java) if (x < 0) y = x + 1; else z = y + 2;

- **Iteration**: do a command repeatedly
  
  (Java) while (x > 10) output(x--);
  
  (Pascal) for x := 1 to 12 do output(x*2);
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
Basic selection statement based on booleans

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

compiles to

\[
\begin{align*}
\text{evaluate } e & \text{ into } t \\
\text{cmp } t, \text{true} & \\
\text{brneq } l_1 & \\
s_1 & \\
\text{br } l_2 & \\
l_1: & s_2 \\
l_2: & \\
\end{align*}
\]
Selection: case

- Generalizes boolean conditionals to types with larger domains

```
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

\[
\text{case } e \text{ of }  \\
\quad c_1 : s_1  \\
\quad c_2 : s_2  \\
\quad \ldots  \\
\quad c_n : s_n  \\
\quad \text{default : } s_d \\
\]

is equivalent to

\[
t := e; \\
\quad \text{if } t = c_1 \text{ then } s_1  \\
\quad \text{else if } t = c_2 \text{ then } s_2  \\
\quad \text{else}  \\
\quad \quad \ldots  \\
\quad \text{else if } t = c_n \text{ then } s_n  \\
\quad \text{else}  \\
\quad \quad s_d \\
\]

- This is just a linear search \((O(n) \text{ time})\)
- If \(e\)’s type is ordered, we can do better with a binary search \((O(\log n) \text{ time})\)
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

```plaintext
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 \ast t
table: l_1
      l_2
      ...
      l_n
done:
```

The most efficient translation of case statements depends on 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

\[
\text{top: } \text{evaluate } e \text{ into } t \\
\text{cmp } t,\text{true} \\
br\text{neq}\text{ done} \\
s \\
br\text{top} \\
done:
\]

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

\[
s; \\
\text{while not } e \text{ do } s
\]
Loop exits

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

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

compiles to

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

done:
```

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

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

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

n++;  // transfer control to point well past end of the loop
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, 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!
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 \( i \) ?)

- Many modern languages support generalized iterators through sets (More on these later in the course)

- C/C++/Java offer a more general-purpose statement
  \[
  \text{for } (e_1; e_2; e_3) s;
  \]
  is equivalent to
  \[
  e_1; \text{ while } (e_2) \{ s; e_3 \}
  \]
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.