Runtime Environments
Runtime Environment

- Programs run in the context of a **system**
  - Instructions, registers, memory, I/O ports, etc.
- Compilers must emit code that uses this system
  - Must obey the rules of the hardware and OS
  - Must be interoperable with shared libraries compiled by a different compiler
- Memory conventions:
  - **Stack** (used for subprogram calls)
  - **Heap** (used for dynamic memory allocation)
Subprograms

- **Subprogram general characteristics**
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
  - Caller/callee info stored on stack

- **Procedure vs. function**
  - Functions have return values
Subprograms

- **New-ish terms**
  - **Header**: signaling syntax for defining a subprogram
  - **Parameter profile**: number, types, and order of parameters
  - **Signature/protocol**: parameter types and return type(s)
  - **Prototype**: declaration without a full definition
  - **Referencing environment**: variables visible inside a subprogram
  - **Name space / scope**: set of visible names
  - **Call site**: location of a subprogram invocation
  - **Return address**: destination in caller after call completes
Parameters

- **Formal vs. actual parameters**
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site
- **Semantic models:** *in, out, in-out*
- **Implementations** (key differences are *when* values are copied and exactly *what* is being copied)
  - **Pass-by-value** (*in, value*)
  - **Pass-by-result** (*out, value*)
  - **Pass-by-copy** (*in-out, value*)
  - **Pass-by-reference** (*in-out, reference*)
  - **Pass-by-name** (*in-out, name*)
Parameters

- **Pass-by-value**
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default

- **Pass-by-reference**
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
Subprogram Activation

- **Call semantics:**
  - Save caller status
  - Compute and store parameters
  - Save return address
  - Transfer control to callee

- **Return semantics:**
  - Save return value(s) and out parameters
  - Restore caller status
  - Transfer control back to the caller

- **Activation record:** data for a single subprogram execution
  - Local variables
  - Parameters
  - Return address
  - Dynamic link

*Linkage contract or Calling conventions (caller and callee must agree)*
Standard Linkages

- Caller and callee must agree
- Standard contract:
  - Caller: **precall** sequence
    - Evaluate and push parameters
    - Save return address
    - Transfer control to callee
  - Callee: **prologue** sequence
    - Save & initialize base pointer
    - Allocate space for local variables
  - Callee: **epilogue** sequence
    - De-allocate activation record
    - Transfer control back to caller
  - Caller: **postreturn** sequence
    - Clean up parameters
x86 Stack Layout

- **Address space**
  - Code, static, stack, heap
- **Instruction Pointer (IP)**
  - Current instruction
- **Stack pointer (SP)**
  - Top of stack (lowest address)
- **Base pointer (BP)**
  - Start of current frame (i.e., saved BP)
- "cdecl" calling conventions
  - callee may use AX, CX, DX
  - callee must preserve all other registers
  - parameters pushed in reverse order (RTL)
  - return value saved in AX

```c
void foo()
{
    int a, b;
    bar(a);
    return;
}
```

```c
void bar(x)
{
    int c;
    baz(x,c);
    return;
}
```

```c
void baz(x,y)
{
    int d;
    return;
}
```
**x86 Calling Conventions**

**Prologue:**
- `push %ebp` ; save old base pointer
- `mov %esp, %ebp` ; save top of stack as base pointer
- `sub X, %esp` ; reserve X bytes for local vars

**Within function:**
- `+OFFSET(%ebp)` ; function parameter
- `-OFFSET(%ebp)` ; local variable

**Epilogue:**
- `<optional: save return value in %eax>`
- `leave` ; `mov %ebp, %esp`
- `pop %ebp`
- `ret` ; pop stack and jump to popped address

**Function calling:**
- `<push parameters>` ; precall
- `<push return address>`
- `<jump to fname>` ; call
- `<pop parameters>` ; postreturn
**Decaf Calling Conventions**

- **param** instruction to pass parameters; call in reverse order (RTL)
  - Pushed on system stack (accessible in function using [BP+offset])
- **call** instruction to transfer control
  - *TODO: before call, save live registers (P6)*
  - Save return address on stack and set up stack frame (BP and SP)
  - Reserve space for local variables (accessible in function using [BP-offset])
  - Set IP to function entry point
  - *TODO: after call, clean/pop parameters from stack*
  - *TODO: after call, restore saved registers (P6)*
- **return** instruction to return to caller
  - Tear down stack frame (BP and SP) and pop return address into IP
  - Return value saved in "ret" special register
# Calling Conventions

<table>
<thead>
<tr>
<th></th>
<th>Integral parameters</th>
<th>Base pointer</th>
<th>Caller-saved registers</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl (x86)</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>EAX, ECX, EDX</td>
<td>EAX</td>
</tr>
<tr>
<td>AMD64 (x64)</td>
<td>RDI, RSI, RDX, RCX, R8, R9, then on stack (RTL)</td>
<td>Saved only if necessary</td>
<td>RAX, RCX, RDX, R8-R11</td>
<td>RAX</td>
</tr>
<tr>
<td>Decaf</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>All virtual registers</td>
<td>RET</td>
</tr>
</tbody>
</table>
Other Design Issues

- How are name spaces defined?
  - Lexical vs. dynamic scope

- How are formal/actual parameters associated?
  - Positionally, by name, or both?

- Are parameter default values allowed?
  - For all parameters or just the last one(s)?

- Are method parameters type-checked?
  - Statically or dynamically?
Other Design Issues

- Are local variables statically or dynamically allocated?
- Can subprograms be passed as parameters?
  - How is this implemented?
- Can subprograms be nested?
- Can subprograms be polymorphic?
  - Ad-hoc/manual, subtype, or parametric/generic?
- Are function side effects allowed?
- Can a function return multiple values?
• Macros
  – Call-by-name, “executed” at compile time
• Closures
  – A subprogram and its referencing environment
• Coroutines
  – Co-operating procedures
• Just-in-time (JIT) compilation
  – Defer compilation of each function until it is called
Heap Management

- Desired properties
  - Space efficiency
  - Exploitation of locality (time and space)
  - Low overhead
- Allocation (malloc/new)
  - First-fit vs. best-fit vs. next-fit
  - Coalescing free space (defragmentation)
- Manual deallocation (free/delete)
  - Dangling pointers
  - Memory leaks
Automatic De-allocation

- Criteria: overhead, pause time, space usage, locality impact
- Basic problem: finding reachable structures
  - **Root set**: static and stack pointers
  - Recursively follow pointers through heap structures
- **Reference counting** (incremental)
  - Track the number of active references to each structure
  - Catch the transition to unreachable (count becomes zero)
  - Has trouble with cyclic data structures
- **Mark and sweep** (batch-oriented)
  - Occasionally pause and detect unreachable structures
  - High overhead and undesirable "pause the world" semantics
  - Partial collection: collect only a subset of memory on each run
  - Generational collection: collect newer objects more often
Object-Oriented Languages

- Classes vs. objects
- Inheritance relationships (subclass/superclass)
  - Single vs. multiple inheritance
- Closed vs. open class structure
- Visibility: public vs. private vs. protected
- Static vs. dynamic dispatch
- Object-records and virtual method tables