Subroutine abstraction

- lexically-scoped subroutines a key part of most programming languages since Algol (~1958)
- need an effective approach to model them within our intermediate representations
- need to account for separate compilation + linking (an assumed part of most large systems, permitting modularity, library use, etc)
- must thus be able to define/declare items in one compilation unit yet refer to them from another: abstraction must account for this

Procedure call abstraction

- when caller invokes callee
  - preserve caller's environment
  - map set of arguments from caller's namespace to callee's
  - set up callee's environment, execute, clean up
- on completion
  - possibly return one or more values from caller to callee
  - restore caller's environment
  - resume execution in caller immediately after point of call

Subroutine namespaces

- generally each subroutine has its own new/protected namespace
- local declarations take precedence over external ones
- parameters generally used to map data from caller's namespace to data in callee's space
- mappings will need to support various addressing modes (e.g. pass-by-value, pass-by-reference)

External interfaces

- need an agreed set of rules for handling references across compilation units
- Need to identify rules on mapping caller/callee namespaces, preserving/restoring caller environment, and setting up/tearing down callee environment
- typically agreed upon by key compiler developers and the language designers very early in language design

Compiler actions

- Compiler must identify storage layout for program components and generate the runtime code that will set up and clean up that storage
- static/global storage layout determined at compile time, as offsets from a base address to be determined when executable loaded into memory
- local storage layouts determined at compile time, but needs to generate the runtime code that will actually set up /tear down the space during execution (e.g. code to set up/tear down a stack frame etc)

Activations

- will refer to each call to a subroutine as an *activation*
- calls made but not yet complete referred to as *active*
- compiler must ensure adequate information is maintained for all active activations
- typical model is stack based, e.g.
  - save current environment on stack
  - push space for return value
  - push parameters
  - push space for local variables, etc

Possible division of responsibility

- Caller sets up:
  - preserve desired registers
  - evaluate actual parameters
  - determine return address
  - ensure pass-by-ref parameters are in memory (not registers)
  - set up parameters
- Callee sets up:
  - set up local variabless
  - rearrange registers as needed
- Callee executes
- Callee cleans up:
  - delete locally allocated space
  - restore registers
  - store return value
- Caller cleans up:
  - return any pass-by-ref params to registers (if appropriate)
  - capture return value
  - deallocate parameter space
  - restore preserved registers
- Caller resumes execution

More complex options

- could maintain information on entire environment for each active subroutine
- encapsulate the environment with the call information
- run the active subroutine in the context of its own environment
- often used in functional languages (e.g. scheme)

Tracking access across scopes

- assuming lexical scoping
  - global scope
  - file scope
  - function scope (possibly nested function declarations)
  - block scope (probably nested blocks supported)
- compiler needs a way to refer to declared items across the various scopes

Scope/offset approach

- number each lexical scope from outermost to innermost, e.g. global (0), file (1), function foo (2), current block (3)
- each local variable/constant's storage location is at some offset from the start of that scope's data storage block
- thus to refer to a local data element we use a pair <scope,offset>
- note that using offsets means we're making some assumptions about storage sizes

Example: <scope,offset>

```
int x = 1;
void f() {
    int x = 2;
    float y = 3;
    print(x * y);
}
int main() {
    float y = 4;
    void g() {
        int x = 5;
        print(x,y);
    }
}
```

scope	x	y
global	<0,0>	n/a
body of f	<1,0>	<1,4>
body of g	<2,0>	<1,0>
body of main	<0,0>	<1,0>

*assuming offsets of 4 for both ints and floats, here not distinguishing between global/file scope, and scope indices are lexically-based

Parameters and return values

- Need to support source/target language addressing modes
- Most common are pass-by-value, pass-by-reference
- Pass-by-value: evaluate argument before copying value to parameter storage space
  - possible concerns for large data types (e.g. arrays) where this is time/memory intensive
  - Might pass a pointer/reference instead, but adds the need to safeguard the passed item against corruption by callee
- Pass-by-reference: need to provide some access mechanism from the caller space to the callee, and ensure callee code uses that mechanism