Subprograms

- Each subprogram has a single entry point
- Calling program is suspended
- Only one subprogram in execution at any given point
- Control returns to the caller when subprogram terminates

Definitions

- Subprogram definition: includes interface and the actions
- Subprogram call: explicit request to execute a subprogram
- Active: a subprogram has begun execution but has not yet completed
- Header: specifies name and parameters, used for calling the subprograms
- Procedures vs. functions

Parameters

- For subprograms to gain access to data it has to process:
  - through direct access to non-local variables (can cause side effects)
  - through parameter passing
- Functions communicate back to caller through return values
- Some languages allow computations/subprograms to be passed as parameters
- Parameters in header are called formal parameters
- Parameters in subprogram call are actual parameters

Binding of Actual Parameters to Formal Parameters

- Most languages use positional parameters: first actual parameter is bound to first formal parameters, etc.
- Some languages (e.g. Python) allow keyword parameters:

  ```
  f(x = 5, y = 10)
  ```

- Keyword parameters is more readable, but user must know the name of formal parameters
- Python allows positional and keyword parameters to be mixed
- Many languages allow default values.
- Some languages allow variable number of parameters. Common in scripting languages, but also C/C++. e.g. `va_list`

Procedures vs. Functions

- Procedures are subprograms that have no return values. Their effects may be observed from their parameters (e.g. sort) or through changes to non-local states.
- Functions provide a mapping from input values to output values (similar to a mathematical function). Why "similar"?
- Some languages allow function or operator overloading, usually by number and type of parameters.

Design Issues

- Local variables: statically or dynamically allocated?
- Nested subprograms
- Parameter passing methods
- Type checking of parameters
- Side effects allowed?
- Return values: how many and what type?
- Generic subprograms?

Local Variables

- Stack dynamic variables are common in modern languages:
  - allow space reuse
  - supports recursion
  - requires indirect access, allocation/deallocation at run time

Nested Subprograms

- Some languages allow subprograms to be defined inside another subprogram
- Limit the scope of the subprogram and where it can be called
- If static scoping is used, also provides a structured way to access non-local variables without parameter passing

Parameter Passing

- Typically we think of each parameter as an in-mode parameter, an out-mode parameter or an inout-mode parameter
- These are implemented differently (and sometimes imperfectly) in different languages

Pass-by-Value

- The value of the actual parameter is used to initialize the corresponding formal parameter
- Usually implemented by copy
- Can also implement by a constant reference, assuming the subprogram does not modify the formal parameter
- Copying can be costly for both space and time

Pass-by-Result

- Implement out-mode parameters (e.g. C#)
- No value is transmitted into the subprogram
- Actual variable must be a variable, and the caller receives the computed value when the subprogram terminates
- If results are returned by copying, we have same disadvantages of pass-by-value
- Need to ensure that the initial value of the parameter is not used in the subprogram
- What about `f(x,x)` when both parameters are out-mode parameters?

Pass-by-Value-Result

- Implement inout-mode parameters
- Combine pass-by-value and pass-by-result

Pass-by-Reference

- Implement inout-mode parameters
- Instead of passing values by copying, pass the access path (e.g. address) of the parameter
- Efficient in both time and space
- Accessing parameter in subprogram requires indirect addressing.
- If the parameter is only in or out, we may accidentally use it in the other direction
- What about `f(x,x)`, `f(A[i],A[j])` (if $i = j$)

Pass-by-Name

- Actual parameter is textually substituted for the corresponding formal parameter in all occurrences in subprogram
- Binding of value or address is delayed until the formal parameter is used
- Not widely used in modern languages

Parameter Passing Implementation

- A run-time stack is used to communicate between caller and subprogram.
- Pass-by-value: copy the value into the stack
- Pass-by-result: stack location used for parameter and copy result back from the stack
- Pass-by-reference: copy the address into the stack

Some Common Languages

- C, Java: pass-by-value only
- C++: pass-by-value and pass-by-reference, also constant reference
- Ada: in parameters cannot be assigned to
- C#: pass-by-value, pass-by-reference, out parameters

Parameter Type Checking

- Most modern compiled languages check the types of the parameters when subprograms are called
- The header/prototype is needed
- Coercion may be performed, but usually not for pass-by-reference. Why not?

Multidimensional Array Parameters

- In Java and C#, arrays are one dimensional and maintain their lengths. Can be passed as `A[][]`
- In C/C++, all but the first dimension must be specified. Why?

Subprograms as Parameters

- Some languages allow subprograms be passed as parameters
- e.g. C/C++ allows pointers to functions, as well as "lambda"
- Functional languages allow functions as parameters
- Some issues:
  - how to type check parameters when these subprograms are called?
  - if there are nested subprograms, what is the referencing environment?

Referencing Environment for Subprogram Parameters

- Shallow binding: environment of the call statement
- Deep binding: environment of the definition of the passed subprogram
- Ad hoc binding: environment of the call statement that passed the subprogram as an actual parameter.
- Deep binding is used for static-scoped languages

Indirect Function Calls

- Sometimes we do not know exactly which function we wish to call until runtime.
- C/C++ allows us to specify this using pointers to functions
- C# allows "delegates" to be set up.

Miscellaneous

- Side effects: most imperative languages do not prevent side-effects, but pure functional languages cannot have side effects in functions
- Return types: some languages allow subprograms to be returned. Most do not.
- Number of return values: some languages allow only a single return value, others allow multiple.

Overloading

- An overloaded subprogram is one that has the same name as another in the same referencing environment
- Distinguished by number and types of parameters, and possibly return type
- Type coercion and default values may make distinction more difficult
- Some languages have predefined overloaded subprograms
- Some languages allow operators to be overloaded

Generic Subprograms

- Subprograms that takes parameters of different types on different activation's
- This is a form of parametric polymorphism
- C++: templates
- Java: Generic methods

Closures

- The closure is a subprogram and the referencing environment where it was defined
- Static-scoped languages without nested subprograms: no need for closures
- If a subprogram can be passed to and called at another location than the defining location, closure is needed to access the appropriate variables (which may no longer exist)
- Supported by almost all functional languages, scripting languages, and also C++ and C#.

Semantics of Calls and Returns

- Subprogram linkage: call and return operations together
- Includes:
  - saving and restoring caller environment
  - parameter and return value passing
  - local variable allocation and deallocation
  - transfer control
- If nested subprograms are allowed, need way to access non-local variables

Simple Subprograms

- Assume no nested subprograms.
- Calling:
  1. Save execution status of current program unit
  2. Compute and pass parameters
  3. Pass the return address to the called subprogram
  4. Transfer control

Simple Subprograms

- Returning:
  1. Copy/move pass-by-value-result and out-mode parameters
  2. Move return value to a place accessible to caller (if needed)
  3. Restore execution status of current program unit
  4. Transfer control back to caller

Simple Subprograms

- Some tasks are done by caller, some done by called
- Prologue and epilogue of subprogram linkage: linkage actions of the called subprogram that occur at the beginning or the end of its execution
- An activation record is used to store data relevant:
  - caller status
  - parameters
  - return address
  - return value
  - local variables
- The activation records for all active subprograms are stored in a runtime stack

Simple Subprograms

- Each invocation of a subprogram pushes an activation record to the stack
- Termination pops the activation record
- Top of the stack corresponds to currently running subprogram (identified by environment pointer EP)
- Support recursion
- Size and format of activation record is usually known at compile time (e.g. C/C++)
- Dynamic link: points to the activation record of the caller
- Static scoping: allows traceback for debugging
- Dynamic scoping: allows access to non-local variables

Simple Subprograms

- Caller actions:
  1. Create activation record for called subprogram
  2. Save status of caller
  3. Compute and pass the parameters
  4. Pass return address
  5. Transfer control

Simple Subprograms

- Called Prologue:
  1. Save old EP as dynamic link, create new EP value
  2. Allocate local variables
- Called Epilogue:
  1. Copy/move pass-by-value-result or out-mode parameters
  2. Move return value to a place accessible to caller (if needed)
  3. Restore EP to point to old dynamic link, set stack pointer to pop the activation record
  4. Restore execution status of current program unit
  5. Transfer control back to caller

Nested Subprogram: Static Scope

- All accessible non-static variables are in existing activation records on run-time stack
- To find the correct location:
  - find the correct activation record in the enclosing scope
  - find the offset within that activation record

Static Chain

- Each activation record includes also a static link
- Pointer to the activation record of the static parent
- A static chain is the chain of the static links connecting certain activation records on the stack
- Static depth: an integer associated with the static scope. 0 is the main program, and each nested subprogram has a static depth 1 higher than its surrounding scope
- Nesting depth/chain offset: the difference of static depth between subprogram referencing a name and subprogram declaring the name.
- Each reference is specified by two integers (chain offset, local offset)
- See example in text

Static Chain

- When a subprogram completes execution, its AR is popped off the run-time stack. There is nothing else to do.
- When a subprogram is called, the static link in its AR needs to be set to the AR of the parent scope
- This can be done with a search following the dynamic links
- But the difference is static depth can also be determined at compile time, so it can be found the same way as any other variables

Static Chain

- As long as there are no closures (e.g. subprograms as parameters), static chain works in all cases
- Access to non-local variables requires links to be followed, possibly slow
- In practice the number of levels is small, but access time is difficult to predict

Dynamic Scoping

- There are two ways (and others) to implement dynamic scoping:
  - Deep access
  - Shallow access

Deep Access

- Search the ARs for the correct variable by following the dynamic links
- "deep": access may require searching deep into the stack
- there is no way to determine the depth of the search at compile time
- AR must store the names of the variables as well

Shallow Access

- Maintain separately a stack for each variable name
- Instances of local variables are pushed onto the stack when the subprogram declaring them are called
- Popped when the subprogram declaring them terminates
- Deep access: faster subprogram linkage, slower variable access
- Shallow access: slower subprogram linkage, faster variable access