Subprograms

CS463@GMU

What can be used for arguments?

what is allowed as a parameter? Whatever we allow, they are called **first class values**. Examples:

- primitive values (int, float, bool, etc)
- arrays, other structured values
- addresses, references, pointers
- types
- function values

Subprograms

- Any time we have a block of code that we can invoke from elsewhere, we have various names to describe this.
  - subprogram: older name (and very broad). Just a block of code we can enter and return from, nothing explicit about arguments/return values.
  - subroutine: clearer intention to solve part of our overall task, but still just a chunk of code that can be executed, and then we return.
  - function: accepts parameters, returns a value.
  - method: like a function but somehow tied to an object or some more complex structure that is assumed available/present/involved.

We might try to use the specific name some language uses, but can just be more general and <u>call everything a subprogram</u>.

parameter modes

How is data transmitted between the code that performs a function call and the function that is called? Similarly, how is data sent back?

- **in**: send in a value, but the subprogram can't affect the source's verison.
- **out**: let subprogram send a value to the caller (like a named return value)
- **in-out**: both in and out through the same parameter.

passing approaches: in

- **pass by value**: copy the actual value, send to subprogram.
  - recipient won't or can't affect the original.
  - it takes time/space to copy the value over.

example: Java primitives are passed by value.

```
public void noEffect (int x, int y){
    // only local variables (params) modified.
    x++;
    y = x*100;
}
```

passing approaches: in and out

- **pass by reference**: copy the address of the value, send that to subprogram ("pass by sharing")
  - in-out mode (via access paths)
  - recipient can affect the addressed value (but not the original address-copy)
  - constant time to copy the address
  - aliasing (between the caller/callee; also could be between multiple parameters)

Examples

- C language: pointer parameters are effectively p.b.r.
- Java: all non-primitive types (reference types) are p.b.r.

passing approaches: in and out

pass by name: expression-argument is evaluated at each usage in the executing the subprogram.

- thus re-evaluated each time the parameter is used!
- allows for creating your own control structures. (Jensen's device)
- very odd to reason about; introduced in Algol 60, but largely not available to programmers now
- implementation: a closure (or thunk)

Haskell and pass-by-name

Haskell uses a version of pass by name called **pass by need**.

- we have referential transparency (= isn't reassignment, it's true Leibniz equality, and we can logically interchange each side of these equations)
- There's no need to re-calculate the result each time we see the variable, so we can cache the answer and reuse it.
- This "memoized call by name" evaluates each parameter at most once

```
f a b = if a
        then (b,b,b,b,b,b)
        else (0,0,0,0,0,0)

-- never evals fib
example1 = f False (fib 10000)

-- only evals fib once.
example2 = f True (fib 10000)
```

Closures as Haskell functions

Haskell's call by need semantics means that each sub-expression is effectively a **closure/thunk**.

- thunk: an entity that can be run to generate an output; when created, all needed references are figured out, and it will determine what current locals need to be saved for later calculation.
- Haskell: everything's a thunk.
  - every function call
  - every sub-expression
  - nothing is computed until needed, and even then, only as deeply as necessary to get an answer. Laziness in action!

subprograms as parameters

- does it bring its own referencing environment?
  - what would a (non-local) variable named x mean when the subprogram is called in this new location?

  - shallow binding: use the local environment when sub is executed (dynamic scoping)
  - deep binding: use the env. from subprogram's original definition (static scoping)

examples of various bindings

```
#Python-ish code
s = "glob"

def f1(other):
  s = "first"
  other()
def f2():
  print(s)
def main():
  s = "main"
  f1(f2)
```

Notes

- shallow: **f2** prints **"first"**
  - closest definition of **s** when **f2** was called.

- deep: **f2** prints **"glob"**
  - based on **f2**'s original static scope

Other subprogram-as-parameter approaches

- C langs: function pointers
- Haskell, Python: functions are first-class. static scoping.

Haskell

```
ghci> filter even [1..10]
[2,4,6,8,10]

ghci> map (+1) [1..5]
[2,3,4,5,6]
```

Python

```
def inc(x):
  return x+1
def main():
  xs = [1,2,3]
  ys = map(inc,xs)
  print(list(ys))
```

Various kinds of polymorphism

- **subtype polymorphism**:
  - derived/extended types can behave the same as the parent/base type.
  - example: OOP, subclasses
- **parametric polymorphism**:
  - any type may be used for a parameter, because its value is never directly utilized
  - example: Haskell type params, e.g. map :: **forall a b**.(a⟶b)⟶[a]⟶[b]
  - example: C++ Templates
  - example: Java Generics (but must be Class types, no primitives)
- **ad hoc polymorphism**:
  - some method definitions are individually implemented at various types
  - example: implementing Java interfaces
  - example: placing bounds

Implementing Subprogram Calls

First Scenario: old languages with no stack

- in early languages, *all* function variables were static (permanent address)
  - example: early Fortran
  - example: C's static local variables/ (not stored on stack)
- general recursion is not available
  - one permanently afforded frame for the function (no memory for second call)
  - but tail call optimization is still a possibility!
- each method is then just its code segment and its statically-sized frame.
  - linker/loader can just stitch all these blocks together.

First Scenario: activation records

- return address: stores a *code address.*
  - this is the instruction to run after this function call is done.
  - a helper function may be called from multiple places; need to know which caller/which instruction to run after we return.
- parameters/local variables: all known at compile time
  - pre-reserve their space in the frame.

- all frame sizes are known at compile time.

- no dynamically sized items possible in the frame.

frame:
return address
parameters
local variables

Second Scenario: X86_64 style languages

Language Assumptions/choices:

- has a stack. (thus multiple frames of one function are allowed)
  - general recursion is possible. (each frame has its own locals)
- stack-dynamic locals (store locals in frame)
- we'll **disallow nested subprogram** definitions (only have locals or globals)

Stack maintenance:

- for calling/returning, need to maintain dynamic links
  - a pointer to the start of the previous frame, saved for later
  - at return, we need to give back all used stack space
    - dying frame can reset the frame pointer (%rbp) to beginning of previous frame
  - sequence of these dynamic links is the **dynamic chain**.

Second Scenario:

all frame content sizes known at compile time.

- return address: pointer to code
- **dynamic link**: pointer to stack frame
- params/locals: data.

frame:
return address
dynamic link
parameters
local variables

Scenario Two: Scoping Issues

- with **static scoping**: can only see locals or globals; all permanent addresses. known at compile time.
  - frame offsets for locals
  - actual address for globals/statics

- with **dynamic scoping**: must be able to trace through the dynamic chain until we find the correctly-named variable
  - must keep track of names during runtime…
  - keep stepping down dynamic chain, searching name/value pairs, until we find a match.

Third Scenario: nested subprograms

```
def add3(x,y,z):
  temp = x+y
  def add_more(n):
    nonlocal temp;
    temp +=n
  add_more(z)
  return temp
```

example: Python

Language assumptions/choices:

- use a stack
- stack-dynamic locals
- subprogram definitions may be nested
  - thus globals, nonlocals, and locals are possible

Stack Maintenance:

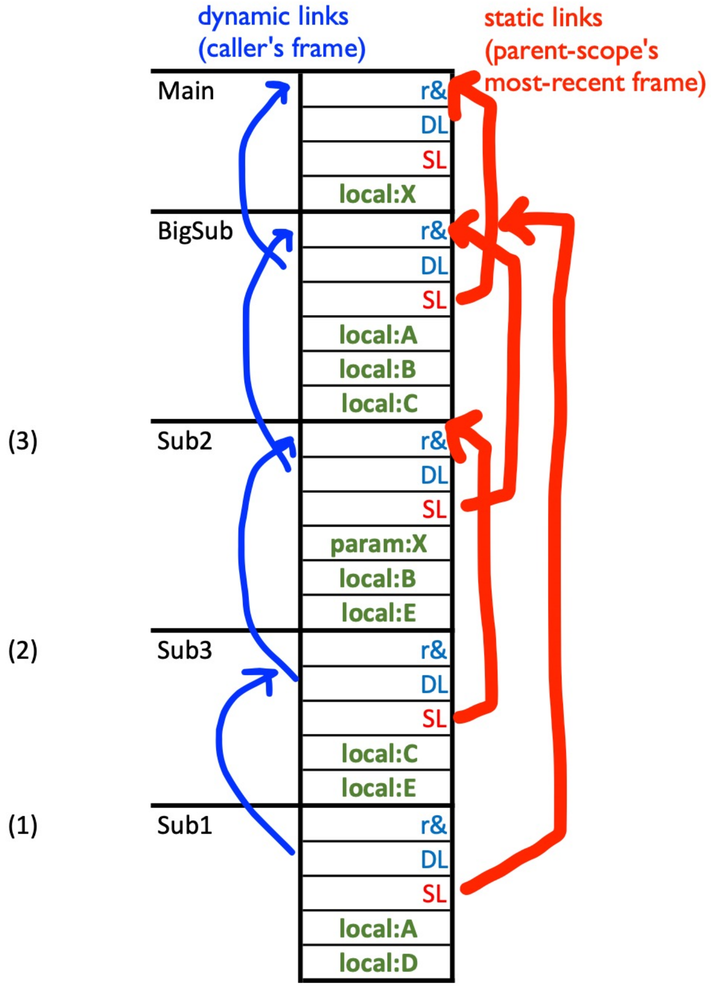
- same as before: use the **dynamic chain** (series of old frame pointers)
- new: **static links**. pointer to parent scope's most-recent frame.

Scenario Three: stack maintenance

- **static links**:
  - each frame keeps a reference to the parent scope's most recent frame
  - statically known how many scope levels outward any variable is (and its frame offset)
  - using a non-local: follow the static links outwards enough, then lookup variable within that frame. (see chaining examples)
- alternative implementations:
  - for every unique variable name in the entire program, keep a stack of values.
    - each declaration pushes a definition onto the name's stack; popped when going out of scope
    - always use top of name's stack as the access/store location
    - space-intensive, a bit faster
  - use a table with one entry per name
    - use caller-save style backups whenever a shadowing variable is introduced/dies

frame:
return address
dynamic link
static link
parameters
local variables

Chaining 2 Example



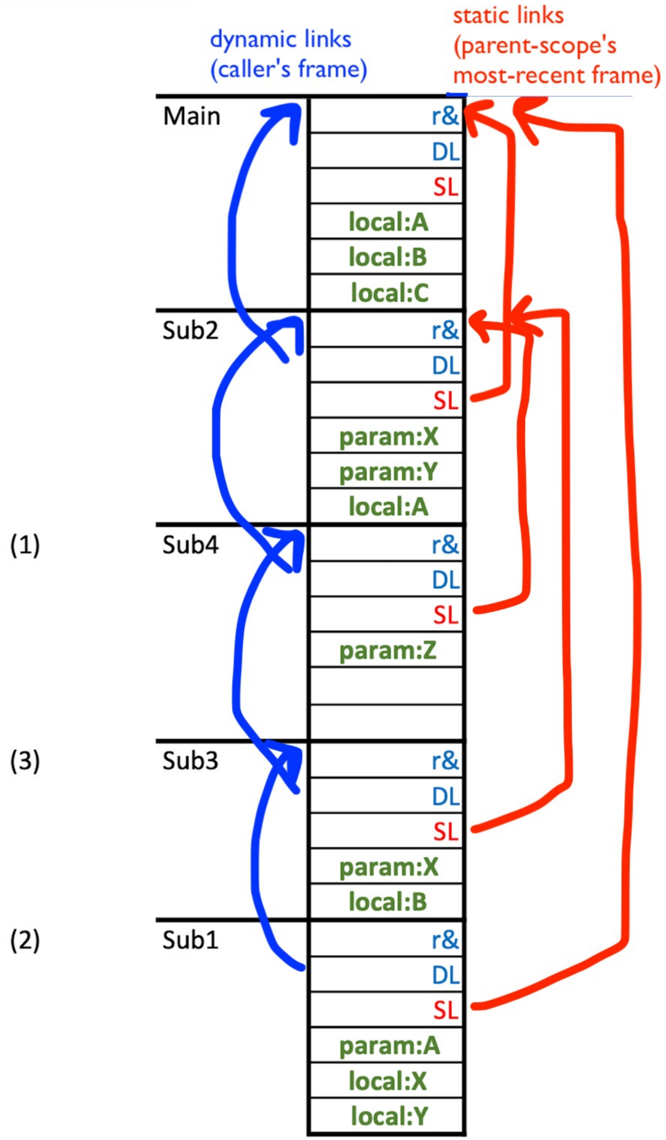

```
# see page 456 of the text.


def Main ():
  int X
  def Bigsub():
    int A, B, C
    def Sub1():
      int A, D
      A := B + C + X      (1) <--------
    def Sub2(int X):
      int B, E
      def Sub3():
        int C, E
        Sub1()
        E := B + A        (2) <--------
      Sub3()
      A := D + E          (3) <--------
    Sub2(7)
  BigSub()
end
Main()
```

code line (1):	#SL hops	frame offset
A	0	3
B	1	4
C	1	5
X	2	3

code line (2):	#SL hops	frame offset
E	0	4
B	1	4
A	2	3

code line (3):	#SL hops	frame offset
A	1	3
D	compilation error	
E	0	5

Chaining 3 Example



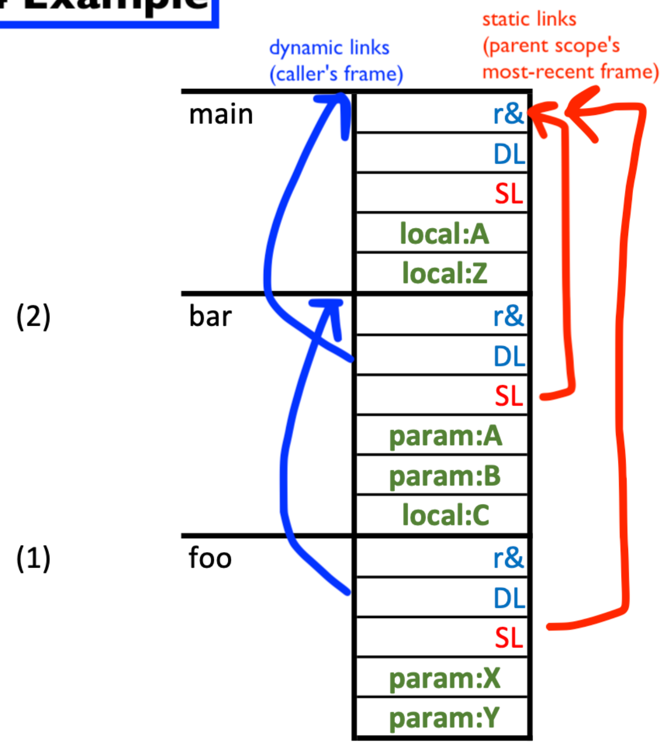
code line (1):	#SL hops	frame offset
Z	0	3
B	2	4
X	1	3

code line (2):	#SL hops	frame offset
A	0	3
B	1	4
X	0	4

code line (3):	#SL hops	frame offset
A	1	5
C	2	5
X	0	3

```
def Main ()
    A, B, C;
    def Sub1 (A):
        X,Y;
        X ≔  A + B;         #2: A, B, X
    def Sub2 (X, Y):
        A;
        def Sub3 (X):
            B;
            Sub1(X+B);
            A ≔ A + X + C;  #3: A, C, X
        def Sub4 (Z):
            Z ≔ Z + B + X;  #1: Z, B, X
            Sub3(Z);
        Sub4(B);
    Sub2(A,B);
Main()
```

Chaining 4 Example

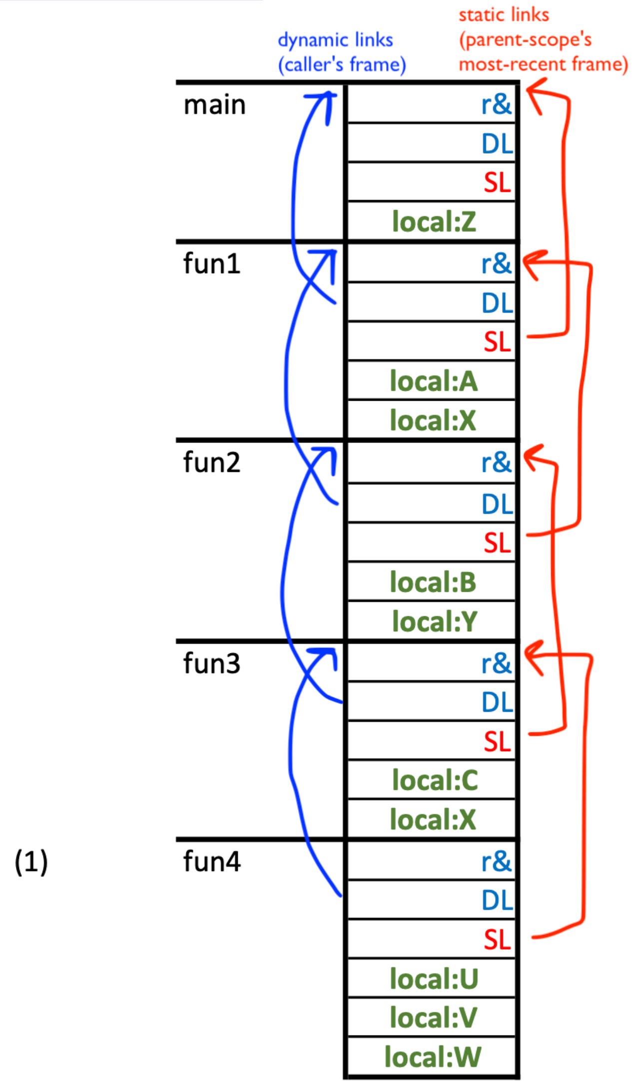


code line (1):	#SL hops	frame offset
A	1	3
Y	0	4

code line (2):	#SL hops	frame offset
A	0	3
X	compilation error	
Z	1	4

```
def main():
    A,Z;
    def foo(X,Y):
        nonlocal A;
        A = 3+Y;      <--- Loc #1
    def bar(A,B):
        nonlocal Z;
        C = 4;
        foo(B,C)
        Z = A+X;      <--- Loc #2
    bar(1,1)
main()
```

Chaining 5 Example



code line (1):	#SL hops	frame offset
A	3	3
B	2	3
C	1	3
W	0	5
X	1	4
Y	2	4
Z	4	3

```
def main():
    Z;
    def fun1():
        A,X;
        def fun2():
           B,Y;
            def fun3():
                C,X;
                def fun4():
                    U,V,W;
                    Z = A+B+C+W+X+Y   ⟵ Loc#(1)
                fun4()
            fun3()
        fun2()
    fun1()
main()
```