Lecture 22

- Louden Chapters 7,8
- Control

Expressions and Statements

- Expression = returns a value and produces no side effect
  - (1+2)
- Statement = doesn't return a value but is executed for it's sideeffects
  - print(5)
- In many languages the distinction is blurry
- Similar to functions vs procedures

Evaluation of expressions

- For "pure" expressions order of evaluation of subexpressions doesn't matter
- For side-effects of course it matters
- Sequence operator in C
  - x = 1; y = 2;
  - x = (y+=1, x+=y, x+1)   /* value of rightmost expression returned */
  - After x = 5, y = 3

Control

- GOTO
- If-statements
  - Dangling else
    - if (e1) if (e2) S1 else S2
    - most closely nested rule
  - Solution: bracketing keyword
    - Ada – if "brackets" with end if
    - Algol68 – if "brackets" with fi

Switch statement in C

```
switch (x-1)
{
    case 0: y=0;
            z=2;
            break;
    case 2:
    case 3:
    case 4:
            y=3;
            z =1;
            break;
    default:
        break;
}
```

Swith statement "falls through" be careful about forgeting break statements

Loops

```
while (e) S;
```

Syntactic sugar:

```
do S while (e);
=
S;
while (e) S;
```

break : exit loop immediately
continue: skip remainder of loop and start again

All computer games:

```
while(1)
{
    if (....) break;
}
```


Exception Handling

- Exception raised or thrown
- Exception handler "handles" or "catches" an exception
- Initial motivation: handling graceful hardware interupts/problems
- Motivation today: libraries can detect errors but in many cases the handling needs to be done by the user

Exceptions

- Languages with exceptions
  - C++, Java, Ada, ML, CLISP
- Languages without exceptions
  - C, Scheme, Smalltalk

Propagating the exception

- Call unwinding (try to find handler in closest block (subexpression) propagate upwards
- Where to continue execution ?
  - resumption model
  - termination model (easier to implement and has better semantics, resumption can be simulated)
- Significant run-time overhead
  - typically no cost when no exception

Procedures and environments

- functions vs procedures similar to expressions vs statements
- Initially as a way to split compilation
- Fortran: static entity without recursion
- Algol60, C: recursion
- LISP and functional languages: functions are first class citizens

Procedure definition and activation

```
// C++ code
void intswap(int&, int&);    // specification


void intswap(int& x, int& y)  // specification
{
   int t = x;    // body
   x = y;        // body
   y = t;        // body
}

intswap(a,b);                    // activation
```

callee
caller
formal parameters
actual parameters

Procedure Semantics

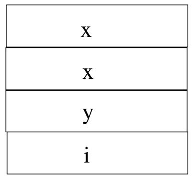
- Activation record = memory allocated for one activation of a procedure
- Communication with the outside world through arguments and non-local references

An example in C

```
int x;
void B(void)
{ int i;
  i = x / 2;
  ... }

void A(void)
{ int x,y;
  ...
  x = y * 10;
  B();
}

main() { A(); }
```



Global environment (defining environment of B)

Activation record of A (calling environment of B)

A possibility ?

- Do everything with argument passing
  - works ok for variables
  - what about constants and functions ?
- Closed form
  - strive for it
- Closure = code of a function together with a representation of the defining environment (used to resolve all outstanding nonlocal references)

Parameter Passing Mechanisms

- Pass-by-value
- Pass by reference
- Pass by value-result
- Pass by Name (delayed evaluation)

Pass-by-value

Replace all parameters with their values
(however values can be changed using pointers)

```
void init_p (int *p)   // the value is the pointer
{ *p = 0; }
int init_ptr(int *p)
{ p = (int *) malloc(sizeof(int)); }      // will be ignored
```

Java:

```
void append_1(Vector v)    // works
{ v.addElement(new Integer(1)); }

void make_new (Vector v)
{ v = new Vector(); }             // will be ignored
```

Pass by reference

- Parameter becomes an alias for the argument – changes are reflected outside (default in Fortran)

```
void inc(int& x) // C++
{ x++; }

procedure inc(var x: integer); // Pascal
begin
    x := x +1;
end;
```

```
// C simulation
void inc(int *x)
{ (*x)++;}
```


Pass by value result

- Similar to pass-by-reference but no aliasing (copy-in, copy-out)

```
void p(int x, int y)
{
   x++;
   y++;
}

main()
{ int a = 1;
  p(a,a);
}
```

pseudo-C notation real C has only pass-by-value

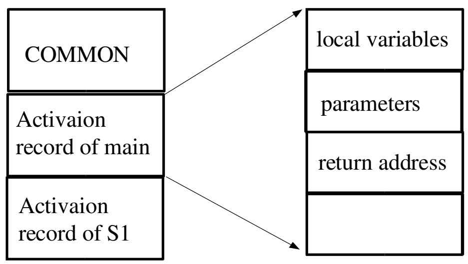
pass-by-reference result = 3
pass-by-value-result = 2

Pass-by name and Delayed evaluation

Argument is not evaluated until it's actual use in the called procedure
- the "name" textual representation at the point of call, replace the name of the parameter.

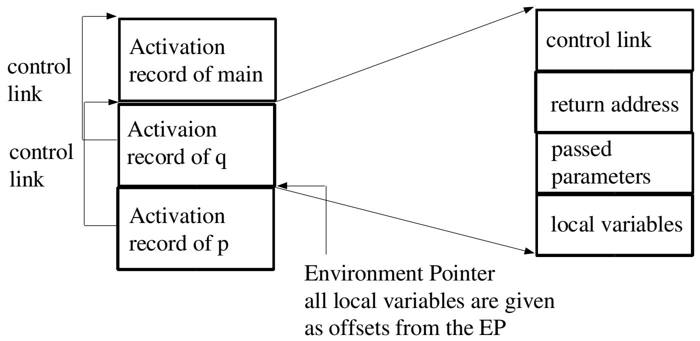
Weird semantics in the presence of side effects
Works nicely in Haskell which is a pure functional language

Procedure Environments, Activations and Allocation



Fortran
static
evnironment
no recursion or
nested procedures

Stack-Based Runtime



Nested procedures

- C and Fortran no nested procedures
  - all non-local references global (easy to find)
- Nested procedures (Pascal, Ada, Modula-2)
- Following the control links results in dynamic scoping rather than lexical
- Additional field called access link : link to lexical or defining environment
- closure <ep, ip>

Maximum flexibility

- Procedures are first class values: they can be created at will
- Stack-based environment impossible
- Basically have to store full environment and code (closure) for everything (ML, Scheme, LISP)
- Automatic reclamation of storage
  - Reference counting
  - Garbage collection

Dynamic Memory Management

- Never deallocate
  - Large memory requirement
- Maintaing free space
  - list of free blocks – coalescing, fragmentation (done with disk drives too)
- Reference counting (eager)
- Mark and sweep
- Generational garbage collection

Mark and sweep

- Lazy: Allocator runs out of space
- First pass: Follow all pointers recursively and mark everything reachable (extra bit)
- Second pass: Move all unreferenced cells back to free list
- Problem: processing delays

Generational garbage collection

- Spread cost more evenly
- Allocated objects that survive long enough are simply copied to permanent space and never get reallocated
- Only newer storage allocations need to be searched