COMS W4115
Programming Languages and Translators
Lecture 17: Procedures
April 1, 2013

Lecture Outline

1. Names
2. Procedures
3. Parameter-passing mechanisms
4. Evaluation strategies
5. Storage-allocation strategies
6. Activation trees and records

1. Names

- A name is a character string used to represent something.
- Names in most languages are identifiers.
- In some programming languages, certain identifiers fall into distinct namespaces that do not interfere with one another. For example, in the C structure declaration

```
struct id {  /* here id is a structure tag */
 int id;    /* here id is a structure member */
} id;        /* here id is a structure variable */
```

the structure tag `id`, the structure member `id`, and the structure variable `id` are in different namespaces and hence distinct identifiers that can be distinguished by context.

- A binding is an association between two things such as between a name and its type or between a symbol and the operation it represents. The time at which this association is determined is called the binding time. Bindings can take place from language design time to runtime.
- The textual region of a program in which a binding is active is called its scope. A scope is often with respect to a namespace.
- In a language with static scoping, the bindings between names and objects are determined at compile time by examining the text of the program. Static scoping is sometimes called lexical scoping. In static scoping, a name refers to its lexically closest declaration.
- In a language with dynamic scoping, the bindings between names and objects depend on the order in which procedures are called at runtime.
- Lifetimes
- The lifetime of a name-object binding is the time between the creation and destruction of that binding.
- The lifetime of an object is the time between the creation and destruction of the object. Depending on the language, the lifetime of an object may be different than that of lifetime of the name-object binding.
  - In C, an object is a location in storage. The storage class determines the lifetime of the storage associated with an object.
  - In C, there are two storage classes: automatic and static. Automatic objects are local to a block and are discarded on exit from the block. Static objects retain their values across entry from and reentry to functions and blocks.
- Object lifetimes are usually determined by the storage-allocation strategy used to manage the storage for that object.
  - Static objects are allocated memory in the code space and have an address that is retained throughout the execution of a program.
  - Stack objects are allocated in a last-in, first-out order on the runtime stack usually in conjunction with procedure calls and returns.
  - Heap objects are dynamically allocated and deallocated at arbitrary times on the runtime heap. Some languages such as Java and C# use a garbage collection mechanism to identify and reclaim heap objects that become unreachable during program execution.

2. Procedures

- A procedure *P* in a programming language is a collection of statements that defines a parameterized computation. An invocation of *P* is called an activation of *P*.
- We use the term *actual parameters* to denote the parameters used in the call of a procedure.
- We use the term *formal parameters* to denote the parameters used in the definition of a procedure.
- We will often call a procedure that returns a value a *function*. (C uses the term function for procedure as well.)
- The type of the function `return_type f(arg1_type a, arg2_type b)` can be denoted by the type expression

  `arg1_type x arg2_type â†’ return_type`
- Some design issues for implementing procedures
  - choice of parameter-passing mechanism
  - storage allocation for local variables: static or dynamic
  - can procedure declarations nest
  - can procedures be passed as parameters, returned as values
  - can procedure names be overloaded

- generic procedures, ones whose computations can be done on different types
- does language have closures (encapsulations of procedures with their runtime context)

3. Parameter-Passing Mechanisms

- Programming languages differ in how the values of parameters are passed to called procedures.
- Call by value
  - The actual parameter is evaluated if it is an expression or copied if it is a variable. The r-value is placed in the location belonging to the corresponding formal parameter of the called procedure.
  - C and Java use call by value. C leaves the order in which the parameters are evaluated unspecified; Java evaluates the parameters left to right.
  - "swap" example from C
    - Consider the following C program fragment

```
a = 1;
b = 2;
swap(a, b);
printf("a = %d, b = %d\n", a, b);
```

where the function `swap` is defined as

```
void swap(int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

- Now consider the same program fragment with `swap(&a, &b)` in place of `swap(a, b)` and with `swap` defined as

```
void swap(int *px, int *py) {
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

- Call by reference
- The address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- If the parameter is an expression, the expression is evaluated and its value is stored in a new location before the call. The address of that location is passed.
- Useful for passing large parameters to procedures.
- Used for reference parameters in C++. In C++, `swap` can be written with reference parameters as

```
void swap(int &x, int &y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

In the body, `x` and `y` are `int`'s, not pointers to `int`'s. The caller passses as parameters the variables whose values are to be swapped, not their addresses.

- Call by name
- A call-by-name parameter is re-evaluated in the caller's referencing environment each time it is used. The effect is as though the called procedure is textually expanded at the point of the call with each actual parameter substituted for the corresponding formal parameter at every occurrence in the body of the procedure. Local names in the called procedure may need to be renamed to keep them distinct.
- Used in Algol 60.
- Also used at compile time by macros in the C preprocessor.
- Example: Consider the macro definition in C

  ```
  #define max(a, b) ((a) > (b) ? (a) : (b))
  ```

  The C statement

```
x = max(p+q, r*s);
```

will be replaced by the statement

```
x = ((p+q) > (r*s) ? (p+q) : (r*s);
```

4. Evaluation Strategies for the Arguments of a Procedure

- An evaluation strategy defines when and in what order the parameters to a procedure are evaluated.
- In applicative-order evaluation, all parameters are evaluated before applying the procedure. C functions and Java methods use applicative-order evaluation.
- In normal-order evaluation, parameters are evaluated after applying the procedure, and then only if the result is needed to complete the evaluation of the procedure. Normal-order evaluation is used with macros and call-by-name parameters. Haskell uses a memoized version of call by name called call by need.

5. Storage-Allocation Strategies

- Static allocation
  - Storage for all data objects is laid out at compile time.
  - Names are bound to storage as program is compiled.
  - Static allocation was used in early versions of Fortran.
  - Recursion is restricted.
  - Size of all data objects must be known at compile time.
  - No dynamic data structures can be supported.
- Stack allocation
  - Run-time storage is organized as a stack.
  - Activation records (ARs) are pushed and popped as activations of procedures begin and end.
  - Typical kinds of data appearing in an activation record:

```
Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries
```

- Storage for the locals in each call is contained in the AR for that call.
- Used by C and Java.
- Heap allocation
- Storage is allocated and deallocated as needed at run time from a data area called a heap.
- Necessary when data outlives the call to the procedure that created it.
- Also needed when the values of local names must be retained after an activation ends.

6. Activation Trees and Records

- Consider the following C program for Euclid's algorithm.

```
#include
int x, y;
int gcd(int u, int v) {
  if (v == 0)
     return u;
  else
     return gcd(v, u%v);
}

main() {
  scanf("%d%d", &x, &y);
  printf("%d\n", gcd(x, y));
  return 0;
}
```

- A tree, called an activation tree, can be used to represent the procedure calls made during an execution of gcd because the lifetimes of the procedure activations are nested.
- Note that:

- The sequence of procedure calls corresponds to a preorder traversal of the tree.
- The sequence of returns corresponds to a postorder travseral.
- The path from the root to a node *N* shows the activations that are live at the time *N* is executing.
- Procedure calls and returns are managed by a control stack.
- On each procedure call, an activation record for that procedure is pushed on the stack. The activation record for each procedure call contains the information needed to manage the execution of that procedure call. When the call returns, that activation record is popped from the stack.

7. Reading

- ALSU, Sections 6.9, 7.1, 7.2

---

aho@cs.columbia.edu