COSC 6352 Notes for 9/18/08 lecture
Nathan Leach (proxy999@hotmail.com)

Procedural Abstraction

Basic components:

- formal parameter part -
  - some languages allow this to be empty
  - contains binding occurrences of the formal parameters
- body of the procedure
  - a construct whose interpretation is deferred until the procedure is invoked

Form:

```
proc I (...;p;...)
        C;
```

or **procedure abstract** form

```
I = proc (...;p;...)
        C;
```

Semantics of Procedures

Mathematical function mapping a sequence of locations (explicit arguments) and a store (implicit arguments) to a new store.

(note: function keyword in some languages will be referred to as expression procedure. If no value is returned, it will be called a command procedure.

The store is the store at invocation. What about the environment? Environment is important because of **free identifiers**.

Example:
(binding occurences in bold)

```
proc (var i:t)
var temp : t;
begin
        temp := i + k;
        i := temp;
end;
```

k & t are free identifiers, environment is going to show the location of k & t

If the environment is the procedure definition environment, then this is static binding. It does not change with execution. Location bound to free identifiers is fixed. (Pascal/C use static binding.)

If procedure invocation environment, then this is dynamic binding. Locations change as program executes. (Lisp/APL use dynamic binding.)

Interpreted languages tend to use dynamic binding; it is easier to implement than static binding.

Advantages/Disadvantages of static and dynamic binding:

dynamic binding makes it more difficult for program readers to see where the identifiers are bound.

PL processors (compilers) have difficulties doing type checking and identifier binding before execution.

There are vulnerabilities of identifiers in the procedure invocation environment (some programmers might choose identifier names that match names of free identifiers.) This allows the procedures to change the free identifier value because of the name.

Expression Procedures

form

```
proc I (...;p;...)
        E;
```

similar form:

```
I = proc {...;p;...)
        E;
```

The body is conceptually E, so the invocation is also E. Conceptually the body is an expression, but it is a composite command in practice.

Command Procedures

body is C so invocation is also C

notice that semantics of command and expression procedures are similar.

Principle of Abstraction
(A VERY IMPORTANT PRINCIPLE TO NOTE)

Any semantically meaningful syntactic class S that can in principle be used as the body of a form of abstraction and the resulting S-procedure may be invoked by an invocation that is also in class S.

Example:

S = l-expressions [expressions that have l-values]

l-expression procecdures & invocation will be an l-expression

```
selector I {...;p;...) : I;
        L: returns l-value of L

type stack =
        record
                a : array [1..n] of t;
                p : 0..n;
        end

selector top (var s : stack) : t;
        returns s.a [s.p];
```

invoke like:

top (d) := ...;

where d is of type *stack*.

Parameter Passing Mechanisms

1. Call-by-value – The actual parameter must have an r-value and the formal parameter is bound to a new location that is initialized by r-value of actual parameter.

Example:

proc p (...;f : int; ...) [r-value of *a* below has a new location]

p(*a*)

2. Call-by-reference – Actual parameter must have an l-value and the formal parameter is bound to the l-value of the actual parameter.

3. Call-by-name – Introduced in Algol 60 [similar to beta reduction in lambda calculus]

Imperative languages don't tend to use this.

Algol 60 report on call-by-name semantics (obtained from http://www.masswerk.at/algol60/report.htm#Description)

4.7.3. Semantics. A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. procedure declarations). Where the procedure body is a statement written in Algol the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement.

4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and types) of the corresponding actual parameters, these assignments being considers as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. if the procedure is called from a place outside the scope of any non-local quantity of the procedure bode the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence. The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions. For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct Algol statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter to be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an Algol 60 statement (as opposed to non-Algol code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further

procedure calls. Ultimately it can only be used by a procedure body expressed in non-Algol code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters. All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number is the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code. The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-Algol code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

Example:

```
proc square (var I : int) [assume var is the call-by-name keyword]
        var t : int;
        begin
                t := sqr (i);
                i := t;
        end
```

invocation: square (a[j])

replaced with:

```
var t : int;
```

```
begin
        t := sqr (a[j]);
        a[j] := t;
end
```

2 kinds of conflicts:

1. free identifiers of actual parameter expression can be captured by identifiers in the proc
2. free identifiers of procedure can be captured by identifiers in the invocation environment. They made renaming rules for handling conflicts.

NOTE: Any textual substitution in any PL can have these problems.