Using Values

1

First and Second Class Values

- A value for which *all* the kinds of operations are allowed is called a *first-class value*
- In Pascal, functions and procedures are second class values
  - they *can* be passed as parameters
  - but they *cannot* be assigned or used as components
- In most imperative programming languages (such as Fortran, Algol 60, Pascal, and Ada) procedural abstractions (functions and procedures) are second class values in order to avoid some implementation problems
- Some languages, such as ML, Miranda, Lisp (also Algol-68, Icon) avoid most of these class distinctions

The Type Completeness Principle

- First and Second class are just two examples. *E.g.,* Pascal does not allow functions to return composite types, while C allows structures (records) but not arrays.

> **The Type Completeness Principle:**
> No operation should be *arbitrarily* restricted in the types of values involved.

Are Composite Values First Class?

- **Composite values:** *e.g.* arrays, records, choice types
- No principle prevents them from being first class, but considerations of implementation do
- **Pascal:** composite values ...
  - can be passed as parameters, both by value and by reference
  - can be stored in variables
  - *cannot* be named (`const` declarations are only for primitive values)
  - *cannot* serve as function return values
- **Old C:** uses the design principle that no expensive operation could be hidden from the programmer. Composite values
  - cannot be passed as parameters (C has only "by value" parameter passing).
    - If you tried doing that,
      - **Arrays**: a pointer to the first element would be passed.
      - **Struct/Union:** some compilers would barf, others would pass a pointer.
  - can be stored in variables
  - *cannot* be named (there are no `const` declarations)
  - *cannot* serve as function return values

4

Composite Values in ANSI-C

- **ANSI-C:** tried to correct many of the problems of C. Still, only "by value" parameter passing.
- **Struct/union:** can now be passed as parameters by value, and can serve as function return values.
  - Some compilers (Borland-C) allow the programmer to set a warning flag, so that each time this feature is used, the compiler would issue a warning.
  - **Arrays:** still cannot be passed by value, or returned. This feature could not be corrected due to the heavy use of the equivalence between arrays and pointers.
  - **Value naming:** The new **`const`** keyword cannot serve to name values. It is rather used to declare storage which cannot be changed without specifically stating that the **`const`** property is removed.
- **Loophole:** you can still pass arrays by value, by putting them inside a structure or a union.

5

First Class Function Values?

- The structure of many (primarily imperative) languages imposes intricate difficulties on using function values as first class values.
- One problem: ***dangling references*** – access to variables that no longer exist.
  - For example, variables that are defined globally in the "normal" environment of a function, but not when the function is passed to an environment where a variable name it uses is not defined

6

Dangling References

- For instance: assignment of a reference to a local variable into a variable of a longer lifetime (e.g. a global variable), as in the following pseudo-Pascal code fragment:

```
var r: ^ Integer;

procedure P;
   var v: Integer;
      begin
         r := &v
      end;
begin
   P;
    ...
   r^ := 1
end;
```

☹

- Pascal rules out such problems by disallowing assignment of references to local variables into *any* other variable.

7

Storing Function Values in Pascal?

Suppose that Pascal allowed function values to be stored in variables. Dangling references would reappear!

```
var fv: Integer -> Char;

   procedure P;
   var v: ...;
      function f(n: Integer): Char;
      begin
         ...v...
      end;
   begin
      fv := f
   end

begin
   P;
   ... fv(0) ...
end;
```

☹

No explicit reference to the local variable `v`, but the value of `f` uses `v`, and hence assigning `f`'s value to a global variable creates a dangling reference.

Storing Function Values in C

- **C:** function values could be stored in variables
  - No nested functions
- **Gnu-C (***A C like language designed by Richard Stallman***):** allows nested functions. Responsibility lies with programmer.
- **C++:** Keeping the C spirit, allows function values to be stored in variables. However, C++ must have nested functions!
  - Classes contain function members.
  - Classes are just like structs.
  - Structs can be defined anywhere, including inside a function!
  - Function members defined inside functions are nested functions.
  - In fact, you can define classes with function members in them inside any other function members, so arbitrary nesting is possible.

  Solution: nested functions are not allowed to access automatic variables of the wrapping function!
- **Gnu-C++:** Stallman gave in to Stroustrup!
  - No ordinary nested functions (even though Gnu-C++ is supposed to be a super set of Gnu-C.
  - Follows C++ semantics regarding nested member functions

9

Function Values: Implementation as Pointers?

- Suppose you need to define an function for **interrupt handling** that gets a function argument `f` and a parameter `x` of the right type for `f`.
- **In C**: the best way to do that is to define a function `interrupt_handler` with two pointer parameters `f` and `x` of type `void`.
- But then: there is no guarantee that the actual parameters for `f` and `x` agree in their types: a function call `f(x)` within `interrupt_handler` may lead to a type error in runtime!
- **Conclusion**: Pointers for function values, like all pointers, are not type safe. It is safer to use fully **typed function values** as in ML.

10

Function Values – Conclusions

- Allowing both nested functions and first-class function values may lead to dangling references problems detected only during run time (as in **Gnu-C**)
- Solutions using syntactic restrictions:
  - Function values are second class (**Pascal**)
  - No nested functions (**C**)
  - Nested functions have no access to variables of a wrapping function (**Algol-68**, **C++**, **Gnu-C++**)
- A more dramatic solution: only global variables (as in functional languages)

11

Expressions

12

Expressions

- An expression is a program phrase that can be evaluated to yield a value
  - **Building Blocks:**
    - Literals
    - Aggregates
    - Constant and Variable Access
  - **Building Tools:**
    - Conditional expressions
    - Function calls
- The interest is not in the syntactic details, but rather with the underlying concepts:
  - A language is impoverished if it omits one of the above
  - It may carry redundancies if it has more than the above

Literals

- The simplest kind of expression
- A fixed and manifest value of some type
- Pascal:
  - **Integer:** `365`
  - **Real:** `3.1415926`
  - **Character:** `'?'`
  - **String:** `'this is the time for all good men to come'`
- In Smalltalk `7/9` is a literal (an object of the ***fraction*** class).

Constant and Variable Access

- An expression may access the content of a variable or a named constant.
  - Pascal:
    ```
    const Pi = 3.1416; var r: real;...
    ```
    The expression:
    ```
    2 * Pi * r
    ```
    Involves both constant **and** variable **access.**
- A programming languages should provide notation for accessing the components of values of composite types
- **Example**: In Pascal `V[I]` and `V.I` are used for accessing variables, but there is no similar notation for constant access.
  const classic = 'War and Peace ';
  var title: **packed array** [1..16] **of Char;**
  title[1] is legal but classic[1] is not.
- In Ada and Turbo Pascal the problem is fixed: same notation for access of constant and variable**.**

Aggregates

- An expression that constructs a composite value from its components
  - The components values are determined by evaluating sub-expressions
- **ML:**
  - Tuple: `(a*2.0, b/2.0)`
  - Record: `{y = thisyear + 1, m = "Jan", d = 1}`
  - List:
    ```
    [31, if leap(thisyear) then 29 else 28,
     31, 30, 31, 30, 31,31, 30, 31,30, 31]
    ```
- **C** and **Ada**:
  - Arrays (only in intializers):
    ```
    monthsize: array(Month) of Natural :=
     (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
    ...
    if leap(thisyear) then monthsize(feb) := 29 end if;
    ```
  - Records:
    ```
    newyearsday := (y => thisyear+1, m=> jan, d=> 1);
    ```
- **Pascal**: None
- **C++**: Only if a constructor was explicitly defined for the structure/class.

Conditional Expressions

- Several sub-expressions, *exactly* one of which is chosen to be evaluated.
- Many languages do not provide conditional expressions:
  - Pascal.
  - Fortran.
  - Ada: `if x > y then max := x else max := y end if;`
- Languages with conditional expressions:
  - C: (limited ?: operator)
  - Icon
  - ML:

```
case thismonth of
  "Feb" => if leap(thisyear) then 29 else 28
|"Apr" => 30 | "Jun"  => 30 | "Sep" => 30 | "Nov" => 30
|  _    => 31
```

  - Algol-68:
    (**if** u>v **then** v **else** u) := u+v;

Function Calls

- Compute a result by applying a function abstraction to an argument
  - **F**(**Actual Parameter Expression(s)**)
  - Most languages: F must be an identifier.
  - If function abstractions are first class values (e.g. ML):
    `(if ... then sin else cos) (x)`
- An operator is nothing but a function called with (usually) an infix notation:
  - ♦E is essentially ♦(E)
  - $E_1 + E_2$ is essentially +(E_1, E_2)
  - Lisp is unique in not recognizing the infix notation at all.
- **Pascal** and **C:** only a rough analogy.
  - Overloading is peculiar to operators and some built-in functions.
- **Icon, C++, Ada, ML:** operator call is the exact equivalent of the function call form.
  - Easier to learn.
  - Allow program to redefine operators: notationally convenient.

More on Strings: Operations

- Simple common operations:
  - copying (:=), equality test (=), lexical order (<), concatenation (+)
- Simple but less common (ABC):
  - repetition ("ho"*3 = "hohoho")
  - Length
  - Finding the minimum character in a string
- Conversions from other data types to strings
  - Important because output is often a string, so every other type needs to be converted to strings.
- More sophisticated operations:
  - Substring extraction, character search in string, string search in string
  - Extraction from strings into another data type (C's **sscanf**)
  - Split operations: from a string to an array (ABC, Perl, SAL):

```
split("Veni, vidi, vici", ",", Result)
```

`Result` is a string array that contains now `"Veni"`, `" vidi"`, and `" vici"` in its cells.

19

Methods for string conversion

- Different conversion function for each type
- One all-purpose function like C's **sprintf**:

```
sprintf(ResultString,
    "Give me %d number%s", IntVar,
    (IntVar == 1) ? "" : "s");
```

- SAL's edited strings:

```
ResultString :=
    'Give me {IntVar}
     number{if IntVar = 1 then ""
     else "s" end}'
```

20

Pattern Matching

- SAL, AWK and Perl offer the match operator ~
- Gets a *target string* and a *regular expression string* and returns a Boolean value according to whether the two match.
- Important regular expression constructors:
  - R1 | R2 - matches both R1 and R2
  - R* - matches zero or more Rs
  - (R) - matches R and calls it a group
  - \n - matches the nth group that was matched
- Example: "(l|b)(i|o)b\2" matches "libi", "lobo", "bibi" and "bobo"
- **Substitute**(MyString, (i(s))", "wa\2")
  - Replaces any "is " in MyString by "was "

21