Aslan Oztreves

CS-360

Krzysztof Nowak

Summary 6.1 – 6.7

**6.1 – Expression Evaluation**

An expression generally consists of either a simple object (e.g., a literal constant, or a named variable or constant) or an *operator* or function applied to a collection of operands or arguments, each of which in turn is an expression. It is conventional to use the term *operator* for built-in functions that use special, simple syntax, and to use the term *operand* for an argument of an operator. In most imperative languages, function calls consist of a function name followed by a parenthesized, comma-separated list of arguments, as in;

my\_func(A, B, C)

* 6.1.1 **Precedence and Associativity**
* 6.1.2 **Assignments**
* 6.1.3 **Initialization**
* 6.1.4 **Ordering within Expressions**
* 6.1.5 **Short-Circuit Evaluation**

**6.2 - Structured and Unstructured Flow**

Control flow in assembly languages is achieved by means of conditional and un- conditional jumps (branches). Early versions of Fortran mimicked the low-level approach by relying heavily on goto statements for most nonprocedural control flow:

if (A .lt. B) goto 10 ! ".lt." means "<"

The 10 on the bottom line is a *statement label*. Goto statements also featured prominently in other early imperative languages. ‘

* 6.2.1 Structured Alternatives to goto
* 6.2.2 Continuations

**6.3 - Sequencing**

Sequencing is central to imperative programming. It is the principal means of controlling the order in which side effects (e.g., assignments) occur: when one statement follows another in the program text, the first statement executes before the second.

**6.4 - Selection**

Selection statements in most imperative languages employ some variant of the if. . . then . . . else notation introduced in Algol 60:

if *condition* then *statement*

else if *condition* then *statement*

else if *condition* then *statement*

...

else *statement*

As we saw in Section 2.3.2, languages differ in the details of the syntax. In Algol 60 and Pascal both the then clause and the else clause were defined to contain a single statement (this could of course be a begin. . . end compound statement). To avoid grammatical ambiguity, Algol 60 required that the statement after the then begin with something other than if (begin is fine). Pascal eliminated this restriction in favor of a “disambiguating rule” that associated an else with the closest unmatched then. Algol 68, Fortran 77, and more modern languages avoid the ambiguity by allowing a statement *list* to follow either then or else, with a terminating keyword at the end of the construct.

* 6.4.1 Short-Circuited Conditions
* 6.4.2 Case/Switch Statements

**6.5 - Iteration**

Iteration and recursion are the two mechanisms that allow a computer to per- form similar operations repeatedly. Without at least one of these mechanisms, the running time of a program (and hence the amount of work it can do and the amount of space it can use) would be a linear function of the size of the program text. In a very real sense, it is iteration and recursion that make computers useful for more than fixed-size tasks. In this section, we focus on iteration. Recursion is the subject of Section 6.6.

* 6.5.1 Enumeration-Controlled Loops
* 6.5.2 Combination Loops
* 6.5.3 Iterators
* 6.5.4 Generators in Icon
* 6.5.5 Logically Controlled Loops

**6.6 - Recursion**

Unlike the control-flow mechanisms discussed so far, recursion requires no special syntax. In any language that provides subroutines (particularly functions), all that is required is to permit functions to call themselves, or to call other functions that then call them back in turn. Most programmers learn in a data structures class that recursion and (logically controlled) iteration provide equally powerful means of computing functions: any iterative algorithm can be rewritten, automatically, as a recursive algorithm, and vice versa. We will compare iteration and recursion in more detail in the first subsection below. In the following subsection, we will consider the possibility of passing *unevaluated* expressions into a function. While usually inadvisable, due to implementation cost, this technique will sometimes allow us to write elegant code for functions that are only defined on a subset of the possible inputs, or that explore logically infinite data structures.

* 6.6.1 **Iteration and Recursion**
* 6.6.2 **Applicative- and Normal-Order Evaluation**

**6.7 - Nondeterminacy**

A nondeterministic construct is one in which the choice between alternatives is deliberately unspecified.