Control

Control:

* what gets executed, when, and in what order

Abstraction of control:

* Expression
* Statement
* Exception handling
* Procedures and functions

Programming is all about 2 things:

* data
* instructions that manipulate data

Expression vs. Statement

In pure (mathematical) form:

* Expression:
  + no side effect
  + return a value
  + Manipulate the data, at the end I will have a value but data is not gonna change. Nothing happens to my state.
* Statement:
  + side effect
  + no return value
  + Do not have a value. Change sth in state.

Functional languages aim at achieving this pure form

No clear-cut in most languages

Expression

Constructed recursively:

* Basic expression (literal, identifiers)
* Operators, functions, special symbols

Number of operands:

* unary, binary, ternary operators

Operator, function: equivalent concepts

* (3+4)\*5 (infix notation)
* mul(add(3,4),5)
* “\*”(“+”(3,4),5) (Ada, prefix notation)
* (\* (+ 3 4) 5) (LISP, prefix notation)

Graphical user interface, text, application

Description automatically generated

Expression and Side Effects

Side Effects:

* Changes to memory, input/output
* Side effects can be undesirable – reliability
* But a program without side effects does nothing!

Expression:

* *No side effect: Order of evaluating sub-expressions does not matter (mathematical forms)*
* *Side effect: Order matters*

Applicative Order Evaluation (Strict Evaluation) *--> If you have side effect, you cannot do*

In order to apply an operator as part of an expression evaluation, I will first evaluate the sub-expressions that is needed by that operator

Evaluate the operands first, then apply operators (bottom-up evaluation)

Sub-expressions evaluated first, no matter whether they are needed

* For example if 3+4 becomes 0, and right subtree has too deep, then result will be 0 no matter what that right subtree is but we have to evaluate it so this brings inefficiency.

Chart, radar chart

Description automatically generated

But is 3+4 or 5-6 evaluated first?

Order Matters

Graphical user interface, application, Word

Description automatically generated

Diagram, text

Description automatically generatedExpected Side Effect

Sequence Operator

(expr1, expr2, …, exprn)

* Left to right (this is indeed specified in C)
* The return value is exprn

Text, letter

Description automatically generated

If there is no side effect, you can do exprn first.

Non-Strict Evaluation

I will only evaluate things when I need them

Evaluating an expression without necessarily evaluating all the subexpressions

short-circuit Boolean expression

if-expression, case-expression

Short-Circuit Evaluation

Text

Description automatically generated

if-expression

Graphical user interface, text, application

Description automatically generated



In C, if it is a statement.

expression

case-expression

A picture containing background pattern

Description automatically generated

Usage of Non-Strict Evaluation

CPU Cycle: I can delay some expressions till I need them bc we need efficiency.

Space For Memory: It may be that evaluation cannot be done in real time. I have an expression and I have an unbounded array (my language allows). I know that I can never generate an unbounded array using my computing capabilities but I can have the concept. I may need to go into array as much as I need.

Normal order evaluation (lazy evaluation)

In strict order evaluation or applicative order evaluation, no matter what the operator is, I am doing the operands first.

In normal order evaluation, I apply the operator and I get the operands.

When there is no side-effect:

* Normal order evaluation (expressions evaluated in mathematical form)
  + Operation evaluated before the operands are evaluated;
  + Operands evaluated only when necessary.
  + e1 \* e2 ---> do multiplication operation before e1 and e2

int double (int x) { return x+x; }

int square (int x) { return x\*x; }

Normal order evaluation: square(double(2)) =

* square(double(2))
* double(2)\*double(2)
* (2+2)\*(2+2) 🡪 (with left to right order) 4\*(2+2) 🡪 4\*4 🡪 16
* ---
* There are 2 function calls so this creates inefficiency. Also if you have some side effect under double(2), then 2 side effects will give you different results.

Applicative order evaluation: square(double(2)) =

* square(double(2))
* square(2+2)
* square(4)
* 4\*4 🡪 16

What is it good for?

A picture containing application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Assignments

Assignment operators have to imply side effects

Central construct in imperative languages

* Functional?

General syntax:

* <target\_var> <assign\_operator> <expression>

Operator:

* “=“, “:=“

Watch it when overloaded

Assignment: Conditional Targets

Graphical user interface, text, application

Description automatically generated

Assignment: Compound Operators

We are putting an expression inside a statement

A shorthand method for specifying commonly needed form of assignment

examples:

a = a + b

can be written as:

a += b

Algol, C/C++, JavaScript, Perl, Python, Ruby

Unary Assignments

• Unary assignment operators in C-based languages combine increment and decrement with assignment • Examples:

sum = ++count; // count inc. added to sum

sum = count++; // count inc. added to sum

Assignment as Expressions

In C-based languages, Perl and JavaScript, assignment statements produce results and can be used as operands

Examples:

while ((ch = getchar()) != EOF) { … }

Multiple/List Arguments

Perl and Ruby support list assignments

Example:

($first, $second, $third) = (20, 30, 40);

($first, $second) = ($second, $third);

Statements

If-statements, case-(switch-)statements, loops

Expression returns a value, you have to use that value somewhere. That means that either as part of another expression or as a return value of a function or as a worst case as a right hand side operand of an assignment operator.

Statement does sth, it will not return any value. So there is no typing information associated with it. It will actually do sth in your state. It will perhaps change a variable’s value, change an output, consume an input…