Control Structures

Based on the slides of Maurizio Gabbrielli

Flow control

- Expressions
  - Notations
  - Evaluation
  - Problems
- Commands
  - Assignment
  - Sequential
  - Conditional
- Iterative commands
- Recursion

Expressions

- An expression is a syntactic entity whose evaluation yields a value or does not end, in which case the expression is undefined.

- Expression syntax: three main notations

  – Infix `A + B`

  – Prefix (Polish) `+ A B`

  – Postfix (Reverse Polish) `A B +`

Expression semantics: infix notation

- Priority among the operators:

```
a + b * c ?
```

- Usually arithmetic operators precedence over those of comparison that have precedence over logical ones
- Exceptions are possible
    - APL, Smalltalk: all operators have equal precedence → you must use parentheses

Priority

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), & (address of), * (contents of), ! (logical not), ~ (bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, -	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)		<, >, <=, >= (inequality tests)	=, /=, <=, >, >= (comparisons)
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Meaning (mod, %, …)

Language	13 mod 3	-13 mod 3	13 mod -3	-13 mod -3
C	1	-1	1	-1
Go	1	-1	1	-1
PHP	1	-1	1	-1
Rust	1	-1	1	-1
Scala	1	-1	1	-1
Java	1	-1	1	-1
Javascript	1	-1	1	-1
Ruby	1	2	-2	-1
Python	1	2	-2	-1

Expression semantics: infix notation

- Associativity

```
15-4-3??        (15-4)-3
```

- Not always obvious: in APL (A Programming Language develop in the 1960), for example,

  ```
  15-4-3
  ```

  is interpreted as

  ```
  15-(4-3)!
  ```

Expression semantics: infix notation

- Recap
  - Precedence rules
  - Rules of associativity
  - However, you need to use parentheses in some cases, for example in

```
(15-4) * 3
```

Parentheses are essential

  - Evaluating an infix expression is not simple...

Expression semantics: postfix notation

- Much simpler than the infix:
  - No precedence rules needed
  - No rules of associativity are needed
  - No parentheses needed
  - Simple evaluation using a stack

Expression semantics: postfix notation

- Evaluating using a stack

1. Read the next exp symbol. And put it on the stack
2. If the symbol is an operator:
   - Apply it immediately to the preceding items on the stack,
   - Store the result in R,
   - Delete operator and operands from stack
   - Stores the value of R on the stack.
3. Repeat

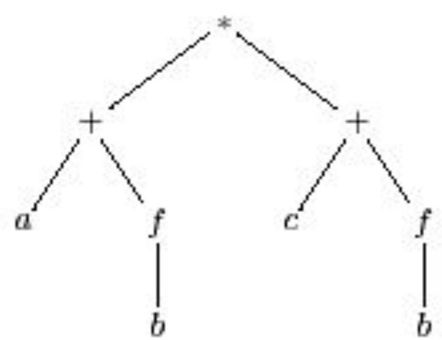
Expression semantics: prefixed notation

- Much simpler than the infix:

  – No precedence rules needed
  – No rules of associativity are needed
  – No parentheses needed
  – Simple evaluation using a stack (but more complicated than the postfix: we have to count the operands that are read)

Evaluating expressions

The expressions internally are represented by trees



Evaluating expressions

- Starting from the tree the compiler produces the object code or the interpreter evaluates the expression
- In both cases the order of evaluation of the subexpressions is important for various reasons:
  - Effects
  - Undefined operands
  - (Optimization)

Effects

- `(a+f(b)) *  (c+f(b))`

  If f modifies b the result from left to right is different from right to left

  - `f(b) = {d = b; b = b-1; return d }`
  - `a=-1,c=5,b=1`
    - `left (-1+1)*(5+0) = 0`
    - `right (-1+0)*(5+1) = -6`
- In some languages, functions with side effects in expressions are not allowed
- In Java the order is clearly specified (from left to right)

Undefined operands

- In C the expression

  $$a == 0 \ ? \ b : b/a$$

  assumes a lazy evaluation. Only the strictly necessary operands are evaluated.
- It is important to know whether the language adopts a lazy assessment or an eager one (all operands are still evaluated)

Short-Circuit Evaluation

- In the case of Boolean expressions often the lazy evaluation is called short-circuit:

```
a == 0 || b/a > 2
```

  - With lazy (short circuit, as in C) → true
  - With eager → possible error

  - With eager (as in Pascal) → error

```
p := list;
while (p <> nil ) and (p^.value <> 3) do
    p := p^.next;
```

Commands

- A command is a syntactic entity whose evaluation does not necessarily return a value, but it can have a side-effect.
  – Side effect: changing the state of computation without returning a value
- The commands
  – are typical of the imperative paradigm
  – are not present in the functional and logical paradigms
  – in some cases they return a value (e.g., = in C)

Variables

- In Mathematics the variable is an unknown that can take the values of a predefined set
  - is not editable
- In imperatives languages: we have changeable variables
  - A variable is a container of values that has a name

    X $\boxed{\mathbf{2}}$

  - The value in the container can be changed by the assignment command.

Assignment

- command that changes the value of a variable
  - X := 2
  - X = X + 1

  Note the different role of X And X

  - X is a **L-Value**, a value that denotes a location (and may appear to the left of an assignment)
  - X is a **R-Value**, a value that can be contained in a location (and can appear to the right of an assignment)

- In General

  Exp1 Assignment Exp2

Assignment

- Normally evaluating an assignment does not return a value but produces a side effect
  - In some languages the assignment also returns a value. In C

    X = 2 returns 2 so we can write

    Y = X = 2

- In imperative languages computation is done by side effects

Assignment operators

- x := x + 1
  - x +:= 1 (Pascal)
  - x += 1 (C)

- In C 10 different assignment operators, increment/decrement and prefix and postfixed
  - ++e (--e): Increment (decrements) before supplying the value to the context
  - e++ (e--): Increment (decrements) after supplying the value to the context
- Incrementing a pointer takes into account the size of the bulleted objects
  - p += 3   Increments the p pointers of 3n bytes, where n is the object size pointed

Expressions and commands (Imperative Languages)

- ALGOL 68: Expression oriented
  - There is no separate notion of command
  - Every procedure returns a value

```
begin
  a:= if b< c then d else e;
  a:= begin f(b); g(c) end;
  g(d);
  2+3
end
```

- Pascal: Commands separated by expressions
  - A command cannot appear where an expression is required (vice versa)
- C: commands mixed with expressions
  - Expressions may appear where you expect a command
  - Assignment (=) allowed in expressions

```
if (a == b){ …                if (a = b) { ….
/* if a = b do ...            /* assign b to a and if result is not 0 do ...
```

Commands for sequence control

- Commands for the explicit sequence control
  - ;
  - goto
- Conditional commands
  - if
  - case
- Iterative commands
  - Bounded iteration (for)
  - Unbounded iteration (while)

Sequential command

- C1; C2
  - is the basic construct of imperative languages
  - It only makes sense if there are side-effects
  - In some languages the ";" more than a sequential command is a terminator
- ALGOL 68, C: The value of a composite command is that of the last command

Goto

- Access debate in the years 60/70 on the usefulness of the Goto

```
if a < b goto 10
...
10:...
```

- Considered useful essentially for
    - Exiting the center of a loop, return from subprogram, handle exceptions
- At the end considered malicious
- Modern Languages
    - They use other constructs to manage the control of loops and subprograms (while, for, if then else, procedures... see ALGOL 60)
    - They use a structured exception handling mechanism (CLU, Ada, C++, Lisp, Haskell, Java, Modula 3)
    - Goto is not present in Java

E. Dijkstra. Go To Statements Considered Harmful. Communications of the ACM, 11 (3): 147-148. 1968.

Structured programming

- Goto "defeated" because considered against the principles of structured programming
- Structured programming (~ 70s), precursor of object oriented programming
  - Modular Code
  - Meaningful identifiers names
  - Extensive use of comments
  - Structured data types (arrays, records..)
  - Structured flow controls
  - ...

Structured control commands

- Only one entry point and one exit point
  - parsing in a linear way the text matches the flow of execution
  - this is a key for understanding the code
- Structured commands
  - `for, if, while, case` ...
  - not the case of `goto`
- Allows structured code and not "spaghetti code"

Conditional command

```
if B then C_1 else C_2
```

- Introduced in ALGOL 60
- Various rules to avoid ambiguity in the presence of nested if
  - Pascal, Java: else associates with the closest then
  - ALGOL 68, Fortran 77: keyword at the end of the command

```
if B then C_1 else C_2 endif
```

  - Explicit multiple branches

```
if Bexp1 then C1
  elseif Bexp2 then C2
  ...
  elseif BexpN then Cn
  else Cn + 1
endif
```

Case

```
case exp of
|   Label_1 : C_1
|   Label_2 : C_2
|   Label_n : C_n
else C_n + 1
```

Descendant of the Fortran goto and switch of ALGOL 60

Many versions in different languages

- Modula: Possible multiple values in the same branch;
- Pascal, C: No range in the label list;
- Pascal: Each branch contains a single command, no branch default (unless `else` used);
- Modula, Ada, Fortran: Default branch;
- Ada: Labels cover all possible values in the EXP type domain;

If or case?

- In comparison with **`If … Then … Else`** the **`Case exp....`** offers
  - More readable code
  - Higher efficiency of code (with a smart compiler)
    - instead of sequential tests as in the evaluation of

      `if... then... else`
    - address calculation given by `exp` and direct jump to the corresponding branch

Iteration

- Iteration and recursion are the two mechanisms that make it possible to obtain complete Turing powerful formalisms.
- Iteration
  - Unbounded: Logically controlled cycles
    (`while, repeat`, ...)
  - Bounded: Numerically controlled cycles
    (`do, for` ...) with number of cycle repetitions determined at the time of the cycle start

Unbounded iteration

```
while condition do command
```

- Introduced in Algol-W, remained in Pascal and in many other languages
- In Pascal also post-test version:

```
repeat command until  condition
```

  - Equivalent to

```
Command;
while not condition do command
```

Unbounded iteration

- Unbounded because the number of iterations is not known a priori
- The unbounded iteration allows the expressive power of the Turing Machines
- It is easy to implement using the physical machine exploiting the conditional jump instruction

Bounded iteration

```
FOR Index := Start TO End BY Step Do
 ….
End
```

- You cannot change `Index, Start, End, Step` inside the loop
- At the beginning of the cycle execution the number of repetitions of the cycle is bounded
- The expressive power is less than the indeterminate iteration: you cannot express computations that do not end
- In many languages (e.g., C, Java) the `for` is not a bounded iteration construct!

Foreach

- Variant of for that iterates over all elements of a data structure

  `foreach { FormalParameter : Expression } Command`
- Increases code readability

Recursion

- Alternative way to iteration to get the expressive power of Turing Machines
- Intuition: a function (procedure) is recursive if defined in terms itself.
- Example: the factorial

```
int fact (int n){
    if (n <= 1)  return 1;
    else
        return n * fact(n-1);
}
```

Recursion and iteration

- Recursion is possible in any language that allows
  - Functions (or procedures) that can call themselves
  - Dynamic memory management (stack)
- Alternative ways to achieve the same expressive power:
  - Each recursive (iterative) program can be translated into an iterative (recursive) equivalent
  - Recursion is more natural with functional and logical languages
  - Iteration is more natural with imperative languages
- In the case of naive implementations, recursion less efficient than iteration
  - optimizing compiler can produce efficient code
  - tail-recursion...

Tail recursion

- A call of g in f it is tail if f returns the return value from g without further computation.
- f is tail recursive if it contains only tail calls

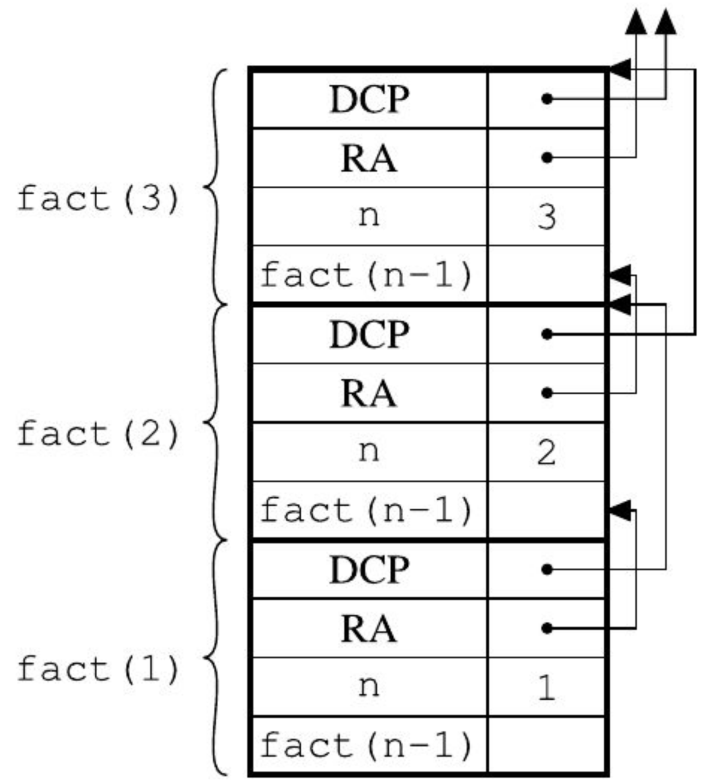
```
function tail_rec (n: integer): integer
begin … ; x:= tail_rec(n-1) end

function non_tail rec (n: integer): integer
begin … ; x:= non_tail_rec(n-1); y:= g(x) end
```

- No need for dynamic memory allocation with stack: just a single Activation Record
- More efficient (with a smart compiler)

Example: the case of factorial (not tail rec!)

```
int fact (int n){
    if (n <= 1)  return 1;
    else
        return n * fact(n-1);
}
```

Situation of the AR after the call of fact(3) and the successive recursive calls



A tail-recursive version of the factorial

```
int factrc (int n, int res){
   if (n <= 1)
      return res;
   else
      return factrc(n-1, n * res)
}
```

- We have added a parameter to store "the rest of the computation"
- Just a single AR
  - After each call the AR can be deleted

Suggested Exercises

- Chapter 6 exercises 1,5,6