Programming Languages

Lecturer: Prof. O. Nierstrasz
Neubrückstr. 10/101
Tel.: 631.4618
Secr.: 631.4692
Assistants: P. Varone, S. Schweizer

Text:

- Wilson & Clark, *Comparative Programming Languages*, Addison Wesley, 1988

Additional material:

- On-line, see: http://iamwww.unibe.ch/~scg/Lectures/pl.html

Overview

1. Introduction
2. Functional programming — *Gofer*
3. Type systems
4. Programming language semantics
5. Object-oriented programming
6. Logic Programming — *Prolog*
7. Structured operational semantics
8. Concurrent programming
9. Programming in the π calculus — *PICT*
10. Objects as processes
11. Text manipulation languages — *Perl*
12. Scripting languages
13. Final exam

What is a Programming Language?

- A formal language for describing *computation*
- A “user interface” to a computer
- “Turing tar pit” — equivalent computational power
- Programming paradigms — different expressive power
- Syntax + semantics
- Compiler, or interpreter, or translator

What Distinguishes Programming Languages?

Generations (increasing abstraction; imperative → declarative):

1. machine codes
2. symbolic assemblers
3. (machine independent) imperative languages (FORTRAN, COBOL, Pascal)
4. domain specific application generators (report generators, database interfaces)

Common Constructs:

- ☞ basic data types (numbers, etc.); variables; expressions; statements; keywords; control constructs; procedures; comments; errors ...

Uncommon Constructs:

- ☞ type declarations; special types (strings, arrays, matrices, ...); sequential execution; concurrency constructs; packages/modules; objects; general functions; generics; modifiable state; ...

Programming Paradigms

A programming language is a *problem-solving tool.*

Imperative style:

- ☞ program = algorithms + data

Functional style:

- ☞ program = functions ∘ functions

Logic programming style:

- ☞ program = facts + rules

Object-oriented style:

- ☞ program = objects + messages

Other styles and paradigms: blackboard, pipes and filters, constraints, lists, ...

A Brief Chronology

Early 1950s “order codes” (primitives assemblers)

1957	FORTRAN
1958	ALGOL
1960	LISP, COBOL
1962	APL, SIMULA
1964	BASIC, PL/I
1966	ISWIM
1970	Prolog
1972	C
1975	Pascal, Scheme
1978	CSP
1978	FP
1980	dBASE II
1983	Smalltalk-80, Ada
1984	Standard ML
1986	C++, Eiffel
1988	CLOS, Mathematica, Oberon
1990	Haskell

Fortran

History:

- John Backus (1953) sought to write programs in conventional mathematical notation, and generate code comparable to good assembly programs
  - No language design effort (made it up as they went along)
  - Most effort spent on code generation and optimization
  - FORTRAN I released April 1957; working by April 1958
  - Current standards are FORTRAN 77 and FORTRAN 90

Innovations:

- comments
- assignments to variables of complex expressions
- **DO** loops
- Symbolic notation for subroutines and functions
- Input/output formats
- machine-independence

Successes:

- Easy to learn; high level
- Promoted by IBM; addressed large user base (scientific computing)

ALGOL 60

History:

- Committee of PL experts formed in 1955 to design universal, machine-independent, algorithmic language
- First version (ALGOL 58) never implemented; criticisms led to ALGOL 60

Innovations:

- BNF (Backus-Naur Form) introduced to define syntax (led to syntax-directed compilers)
- First block-structured language; variables with local scope
- Variable size arrays
- Structured control statements
- Recursive procedures

Successes:

- Never displaced FORTRAN, but highly influenced design of other PLs

COBOL

History:

- designed by committee of US computer manufacturers
- targeted business applications
- intended to be readable by managers

Innovations:

- separate descriptions of environment, data, and processes

Successes:

- Adopted as *de facto* standard by US DOD
- Stable standard for 25 years
- Still the most widely used PL for business applications

4GLs

“Problem-oriented” languages

- PLs for “non-programmers”
- Very High Level (VHL) languages for specific problem domains

Classes of 4GLs (no clear boundaries):

- Report Program Generator (RPG)
- Application generators
- Query languages
- Decision-support languages

Successes:

- highly popular, but generally *ad hoc*

PL/I

History:

- designed by committee of IBM and users (early 1960s)
- intended as (large) general-purpose language for broad classes of applications

Innovations:

- default interpretations for every variable, feature, option etc.
- exception-handling by **on** conditions

Successes:

- achieved both run-time efficiency and flexibility (at expense of complexity)
- first “complete” general purpose language

Interactive Languages

Made possible by advent of time-sharing systems (early 1960s through mid 1970s).

BASIC:

- developed at Dartmouth College in mid 1960s
- minimal; easy to learn
- incorporated basic O/S commands (NEW, LIST, DELETE, RUN, SAVE)

APL:

- developed by Ken Iverson for *concise* description of numerical algorithms
- large, non-standard alphabet (52 characters in addition to alphanumerics)
- primitive objects are *arrays* (lists, tables or matrices)
- operator-driven (power comes from composing array operators)
- no operator precedence (statements parsed right to left)

Special-Purpose Languages

SNOBOL:

- first successful string manipulation language
- influenced design of text editors more than other PLs
- string operations: pattern-matching and substitution
- arrays and associative arrays (tables)
- variable-length strings

Lisp:

- performs computations on symbolic expressions
- symbolic expressions are represented as lists
- small set of constructor/selector operations to create and manipulate lists
- recursive rather than iterative control
- no distinction between data and programs
- first PL to implement storage management by garbage collection
- affinity with lambda calculus

Functional Languages

ISWIM (If you See What I Mean):

- Peter Landin (1968) — paper proposal

FP:

- John Backus (1978) — Turing award lecture

ML:

- Edinburgh
- initially designed as meta-language for theorem proving
- Hindley-Milner type inference
- “non-pure” functional language (with assignments/side effects)

Miranda, Haskell:

- “pure” functional languages with “lazy evaluation”

Prolog

History:

- originated at U. Marseilles (early 1970s), and compilers developed at Marseilles and Edinburgh (mid to late 1970s)

Innovations:

- theorem proving paradigm
- programs as sets of clauses: facts, rules and questions
- computation by “unification”

Successes:

- prototypical logic programming language
- used in Japanese Fifth Generation Initiative

Object-Oriented Languages

History:

- Simula was developed by Nygaard and Dahl (early 1960s) in Oslo as a language for simulation programming, by adding *classes* and *inheritance* to ALGOL 60
- Smalltalk was developed by Xerox PARC (early 1970s) to drive graphic workstations

Innovations:

- encapsulation of data and operations (contrast ADTs)
- inheritance to share behaviour and interfaces

Successes:

- Smalltalk project pioneered OO user interfaces ...
- Large commercial impact since mid 1980s
- Countless new languages ...

Functional Programming

Overview

- Functional vs. Imperative Programming
- Referential Transparency
- Pattern Matching
- Higher Order Programming
- Lazy Evaluation

References:

- Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, pp 359-411.
- Mark P. Jones, “An Introduction to Gofer,” manual, 1991.

What is a Function?

Extensional view:

A (total) function f: A $\rightarrow$ B is a subset of A $\times$ B (i.e., a *relation*) such that:

1. for each a$\in$ A, there exists some (a,b) $\in$ f (i.e., f(a) is defined), and
2. if (a,b1) $\in$ f and (a,b2) $\in$ f, then b1 = b2 (i.e., f(a) is unique)

Intensional view:

A function f: A $\rightarrow$ B is an *abstraction* λ x . e, where x is a variable name, and e is an expression, such that when a value a$\in$ A is substituted for x in e, then this expression (i.e., f(a)) evaluates to some (unique) value b$\in$ B.

Computation as Functional Composition

What is a Program?

A program (computation) is a transformation from input data to output data.

- Program = Algorithms + Data
- Program = Functions ∘ Functions

Church's Thesis:

Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus.

A Bit of History

- **Lambda Calculus** (Church, 1932-33): formal model of computation
- **Lisp** (McCarthy, 1960): symbolic computations with lists
- **APL** (Iverson, 1962): algebraic programming with arrays
- **ISWIM** (Landin, 1966): *let* and *where* clauses; equational reasoning ...
- **ML** (Edinburgh, 1979): originally meta language for theorem proving
- **SASL, KRC, Miranda** (Turner, 1976-85): lazy evaluation
- **Haskell** (Hudak, Wadler, et al., 1988):

Stateless Programming

Imperative style:

```
n := x;
a := 1;
while n>0 do
begin a:= a*n;
        n := n-1;
end;
```

Declarative (functional) style:

```
fac n =
        if n == 0 then 1
        else n * fac (n-1)
```

Declarative languages, and in particular, functional languages, have no implicit state. Programs are constructed entirely by composing expressions.

In functional languages, the underlying model of computation is functional composition.

Referential Transparency

Referential transparency means that “equals can be replaced by equals”.

Evaluation proceeds by replacing expression by their values:

fac 4 ➪ if 4 == 0 then 1 else 4 * fac (4-1)
- ➪ 4 * fac (4-1)
- ➪ 4 * fac 3
- ➪ 4 * (if 3 == 0 then 1 else 3 * fac (3-1))
- ➪ 4 * 3 * fac (3-1)
- ➪ 12 * fac (3-1)
- ➪ 12 * fac 2
- ➪ 12 * (if 2 == 0 then 1 else 2 * fac (2-1))
- ➪ 12 * 2 * fac (2-1) ➪ 24 * fac (2-1) ➪ ... ➪ 24 * 1 ➪ 24

The Church-Rosser property

"If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative order evaluation), then all of these evaluation orders yield the same result".

Consider:

sqr n = n * n

Applicative-order evaluation:

sqr (2+5) ➪ sqr 7 ➪ 7*7 ➪ 49

Normal-order evaluation:

sqr (2+5) ➪ (2+5) * (2+5) ➪ 7 * (2+5) ➪ 7 * 7 ➪ 49

Modelling State

State can be modelled explicitly as a function parameter:

```
sfac s n =
        if n == 0 then s
        else sfac (s*n) (n-1)
```

sfac 1 4

- ➪ sfac (1*4) (4-1)
- ➪ sfac 4 3
- ➪ sfac (4*3) (3-1)
- ➪ sfac 12 2
- ➪ sfac (12*2) (2-1)
- ➪ sfac 24 1
- ➪ ... ➪ 24

Equational Reasoning

Theorem:

For all n >= 0, fac n = sfac 1 n

Proof of theorem:

n = 0: fac 0 = sfac 1 0 = 1

n > 0: Suppose fac (n-1) = sfac 1 (n-1)

```
fac n   = n * fac (n-1)
        = n * sfac 1 (n-1)
        = sfac n (n-1)          -- by lemma
        = sfac 1 n
```

Lemma:

For all n >= 0, sfac s n = s * sfac 1 n

Proof of lemma:

n = 0: sfac s 0 = s = s * sfac 1 0

n > 0: Suppose sfac s (n-1) = s * sfac 1 (n-1)

```
sfac s n  = sfac (s*n) (n-1)
          = s * n * sfac 1 (n-1)
          = s * sfac n (n-1)
          = s * sfac 1 n
```

Pattern Matching

Patterns:

```
fac' 0      = 1
fac' n      = n * fac' (n-1)
```

Guards:

```
fac'' n     | n == 0          = 1
            | n >= 1          = n * fac'' (n-1)
```

Lists

Lists are pairs of elements and lists of elements:

- [ ] stands for the empty list
- x : xs stands for the list with x as the head and xs as the rest of the list
- [1,2,3] is syntactic sugar for 1:2:3:[ ]
- [1..n] stands for [1,2,3, ... n]

Lists can be deconstructed using patterns:

```
head (x:_) = x

len [ ]      = 0
len (x:xs)   = 1 + len xs

prod [ ]     = 1
prod (x:xs)  = x * prod xs

fac''' n     = prod [1..n]
```

Higher Order Functions

Higher-order functions are *first-class values* that can be composed to produce new functions.

```
map f [ ] = [ ]
map f (x:xs)= f x : map f xs
```

```
map fac [1..5]
```

➪ [1, 2, 6, 24, 120]

Anonymous functions can be written as lambda abstractions:

```
map (\x->x * x) [1..10]
```

➪ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Currying

A *curried function* takes its arguments one at a time, allowing it to be treated as a higher-order function.

```
fac = sfac 1
        where sfac s n
                | n == 0        = s
                | n >= 1        = sfac (s*n) (n-1)
```

The following higher-order function takes a binary function as an argument and turns it into a curried function:

```
curry f a b = f (a,b)

sfac (s, n) =   if n == 0 then s
                else sfac (s*n, n-1)
fac = (curry sfac) 1
```

Remembering State

Naive recursion may result in unnecessary recalculations:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

Efficiency can be regained by explicitly passing calculated values:

```
fib' 0 = 0
fib' n = a                  where (a,_) = fib'' n

fib'' 1 = (1,0)
fib'' (n+2) = (a+b,a)       where (a,b) = fib'' (n+1)
```

Lazy Evaluation

"Lazy", or normal-order evaluation only evaluates expressions when they are actually needed. Clever implementation techniques (Wadsworth, 1971) allow replicated expressions to be shared, and thus avoid needless recalculations.

So:

sqr (2+5) ➪ (2+5) * (2+5) ➪ 7 * 7 ➪ 49

Lazy evaluation allows some functions to be evaluated even if they are passed incorrect or non-terminating arguments:

```
ifTrue True x y = x
ifTrue False x y = y

ifTrue True 1 (5/0)
    ➪  1
```

Lazy Lists

Lazy lists are infinite data structures whose values are generated by need:

```
from n = n : from (n+1)

take 0 _  = [ ]
take _ [ ] = [ ]
take (n+1) (x:xs)= x : take n xs

take 5 (from 10)
```

➪ [10, 11, 12, 13, 14]

NB: The lazy list (from n) has the special syntax: [n..]

```
fibs = fibgen 0 1
        where fibgen a b = a : fibgen b (a+b)

take 10 fibs
```

➪ [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Functional Programming Style

```
primes = 2 : primesFrom 3              -- or just: primes = primesFrom 2

primesFrom n = p : primesFrom (p+1)
        where p = nextPrime n

nextPrime n
        | isPrime n                    = n
        | otherwise                    = nextPrime (n+1)

isPrime 2 = True
isPrime n = notdiv primes n

notdiv (k:ps) n
        | (k*k) > n                    = True
        | (mod n k) == 0               = False
        | otherwise                    = notdiv ps n
```

take 100 primes ➪ [2, 3, 5, 7, 11, 13, ... 523, 541]

Type Systems

Overview

- What is a Type?
- Static vs. Dynamic Typing
- Kinds of Types
- Polymorphic Types
- Overloading
- User Data Types

Sources:

- Mark P. Jones, “An Introduction to Gofer,” manual, 1991.
- Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, pp 359-411.
- L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,’“ACM Computing Surveys, vol. 17, no. 4, Dec. 1985, pp. 471-522.
- D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

What is a Type?

Type errors:

```
5 + [ ]
ERROR: Type error in application
*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]
```

A type is a set of values:

- int = { ... -2, -1, 0, 1, 2, 3, ... }
- bool = { True, False }
- Point = { [x=0, y=0], [x=1, y=0], [x=0, y=1] ... }

Are all sets of values types?

A type is a partial specification of behaviour:

- **n, m : int** ⇒ **n + m** is valid, but **not(n)** is an error
- **n : int** ⇒ **n := 1**is valid, but **n := "hello world"** is an error

What kinds of specifications are interesting? Useful?

Static and Dynamic Typing

Values have *static types* defined by the programming language.

Variables and *expressions* have *dynamic types* determined by the values they assume at run-time.

A language is *statically typed* if it is always possible to determine the type of an expression based on the program text alone.

A language is *strongly typed* if it is possible to ensure that every expression is *type consistent* based on the program text alone.

A language is *dynamically typed* if only *values* have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

Type consistency may be assured by (i) compile-time type-checking, (ii) type inference, or (iii) dynamic type-checking.

Kinds of Types

All programming languages provide some set of built-in types.

Most strongly-typed modern languages provide for additional user-defined types.

- **Primitive types:** booleans, integers, floats, chars ...
- **Composite types:** functions, lists, tuples ...
- **User-defined types:** enumerations, recursive types, generic types ...

The Type Completeness Principle:

No operation should be arbitrarily restricted in the types of values involved.

First-class values can be evaluated, passed as arguments and used as components of composite values. Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.

Function Types

Function types allow one to deduce the types of expressions without the need to evaluate them:

fact :: Int -> Int

42 :: Int

⇒ fact 42 :: Int

Curried types:

t1 -> t2 -> ... -> tn	stands for:	t1 -> (t2 -> (... -> tn) ...)

so

f x1 x2 ... xn	stands for:	(... ((f x1) x2) ... xn).
(+) :: Int -> Int -> Int	⇒	(+) 5 :: Int -> Int

List and Tuple Types

List Types

If a is a type then [a] is the type whose elements are lists of values of type a.

[1] :: [Int]

Note that all of the elements in a list must be of the same type, so that an expression such as ['a', 2, False] is not permitted.

Tuple Types

If t1, t2, ..., tn are types and n>=2, then there is a type of n-tuples written (t1, t2, ..., tn) whose elements are also written in the form (x1, x2, ..., xn) where the expressions x1, x2, ..., xn have types t1, t2, ..., tn respectively.

(1, [2], 3) :: (Int, [Int], Int)

('a', False) :: (Char, Bool)

((1,2),(3,4)) :: ((Int, Int), (Int, Int))

The unit type is written () and has a single element which is also written as ().

Polymorphism

Languages like Pascal have *monomorphic type systems:* every constant, variable, parameter and function result has a unique type.

- ☞ good for type-checking
- ☞ bad for writing generic code

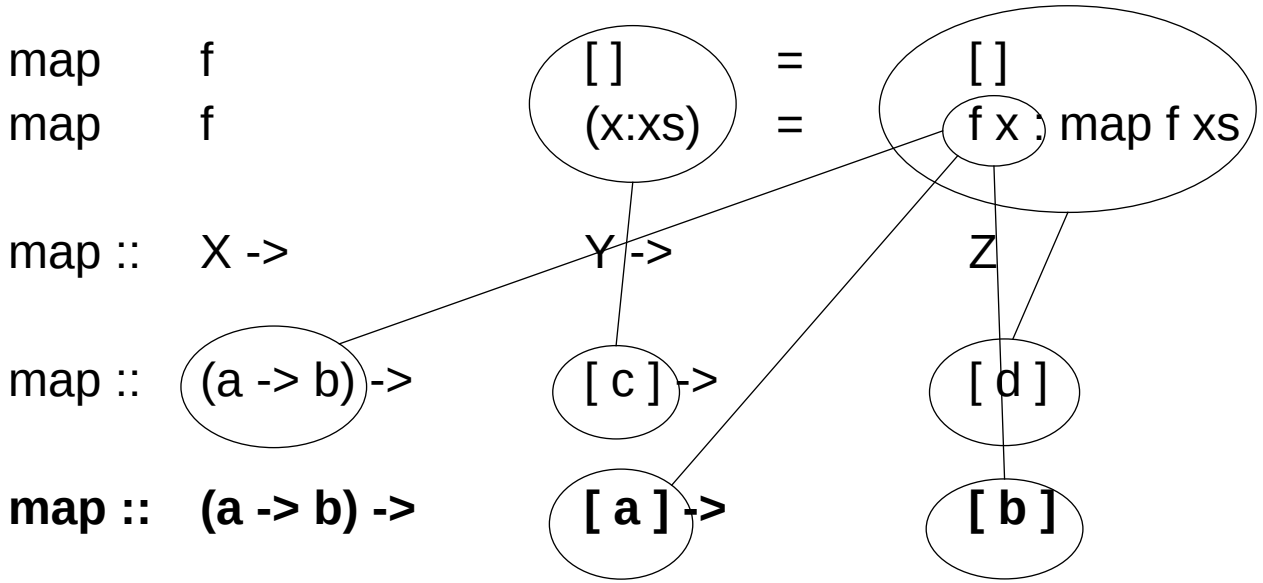
A polymorphic function accepts arguments of different types:

```
length :: [a] -> Int
length [ ] = 0
length (x:xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (x:xs) = f x : map f xs

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Polymorphic Type Inference



Hindley-Milner Type Inference provides an effective algorithm for automatically determining the types of polymorphic functions. The corresponding type system is used in many modern functional languages, including ML and Haskell.

Type Specialization

A polymorphic function may be explicitly assigned a more specific type:

```
idInt :: Int -> Int
idInt x = x
```

Note that the :t command can be used to find the type of a particular expression that is inferred by Gofer:

```
? :t \x -> [x]
\x -> [x] :: a -> [a]

? :t (\x -> [x]) :: Char -> String
\x -> [x] :: Char -> String
```

The (Untyped) Lambda Calculus

Syntax:

$e ::= x \mid e_1\ e_2 \mid \lambda\, x.e$

(Operational) Semantics:

α conversion (renaming):	$\lambda\, x\, .\, e \Leftrightarrow \lambda\, y\, .\, [y/x]\, e$	where y is not free in e
β reduction:	$(\lambda\, x\, .\, e_1)\ e_2 \Rightarrow [\, e_2/x]\, e_1$	
η reduction:	$\lambda\, x\, .\, (e\ x) \Rightarrow e$	if x is not free in e

Example:

True	$\equiv \lambda\, x\, .\, \lambda\, y\, .\, x$
False	$\equiv \lambda\, x\, .\, \lambda\, y\, .\, y$
if b then x else y	$\equiv \lambda\, b\, .\, \lambda\, x\, .\, \lambda\, y\, .\, b\ x\ y$
if True then x else y	$= (\lambda\, b\, .\, \lambda\, x\, .\, \lambda\, y\, .\, b\ x\ y)\ (\lambda\, x\, .\, \lambda\, y\, .\, x)\ x\ y$
	$\Rightarrow^* (\lambda\, x\, .\, \lambda\, y\, .\, x)\ x\ y$
	$\Rightarrow^* x$

The Typed Lambda Calculus

Syntax:

$$e ::= x^{\tau} \mid e_1^{\tau 2 \to \tau 1}\ e_2^{\tau 2} \mid (\lambda\, x^{\tau 2}.e^{\tau 1})^{\tau 2 \to \tau 1}$$

(Operational) Semantics:

α conversion (renaming): $\lambda\, x^{\tau 2}\ .\ e^{\tau 1} \Leftrightarrow \lambda\, y^{\tau 2}\ .\ [\ y^{\tau 2}/x^{\tau 2}\ ]\ e^{\tau 1}$ where $y^{\tau 2}$ is not free in $e^{\tau 1}$

β reduction: $(\lambda\, x^{\tau 2}\ .\ e_1^{\tau 1})\ e_2^{\tau 2} \Rightarrow [\ e_2^{\tau 2}/x^{\tau 2}\ ]\ e_1^{\tau 1}$

η reduction: $\lambda\, x^{\tau 2}.\ (e^{\tau 1}\ x^{\tau 2}) \Rightarrow e^{\tau 1}$ if $x^{\tau 2}$ is not free in $e^{\tau 1}$

Polymorphic functions like “map” cannot be typed in this calculus!
Need *type variables* to capture polymorphism:

β reduction (ii): $(\lambda\, x^{\nu}\ .\ e_1^{\tau 1})\ e_2^{\tau 2} \Rightarrow [\ \tau 2\ /\ \nu\ ]\ [\ e_2^{\tau 2}/x^{\nu}\ ]\ e_1^{\tau 1}$

Kinds of Polymorphism

Polymorphism:

- Universal:
  - Parametric: *polymorphic map function in Gofer; nil pointer type in Pascal*
  - Inclusion: *subtyping — graphic objects*
- Ad Hoc:
  - Overloading: *+ applies to both integers and reals*
  - Coercion: *integer values can be used where reals are expected and v.v.*

Coercion or overloading — how does one distinguish?

3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0

Overloading

Overloaded operators are introduced by means of *type classes:*

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x /= y     = not (x == y)
```

For each overloaded instance a separate definition must be given:

```
instance Eq Int where (==)          = primEqInt
instance Eq Bool where
        True == True                = True
        False == False              = True
        _ == _                      = False
instance Eq Char where c == d       = ord c == ord d
instance (Eq a, Eq b) => Eq (a,b) where
        (x,y) == (u,v)              = x==u && y==v
instance Eq a => Eq [a] where
        [ ] == [ ]                  = True
        [ ] == (y:ys)               = False
        (x:xs) == [ ]               = False
        (x:xs) == (y:ys)            = x==y && xs==ys
```

User Data Types

New data types can be introduced by specifying a datatype name, a set of parameter types, and a set of constructors for elements of the type:

data DatatypeName a1 ... an = constr1 | ... | constrm

The constructors may be of the form:

1. **Name type1 ... typek**
   which introduces Name as a new constructor of type:
   type1 -> ...-> typek -> DatatypeName a1 ... an

2. **type1 CONOP type2**
   which introduces (CONOP) as a new constructor of type:
   type1 -> type2 -> DatatypeName a1 ... an

Examples of User Data Types

Enumeration types:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

what_shall_I_do Sun = “relax”
what_shall_I_do Sat = “go shopping”
what_shall_I_do _ = “looks like I'll have to go to work”
```

Union types:

```
data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```

Recursive Data Types:

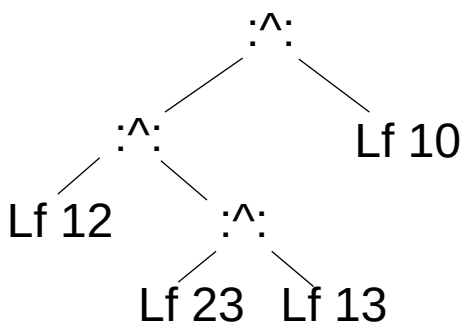
```
data Tree a = Lf a | Tree a :^: Tree a

(Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10 :: Tree Int

leaves, leaves' :: Tree a -> [a]

leaves (Lf l) = [l]
leaves (l :^: r) = leaves l ++ leaves r

leaves' t = leavesAcc t [ ]
        where   leavesAcc (Lf l) = (l:)
                leavesAcc (l :^: r) = leavesAcc l . leavesAcc r
```



Equality for Data Types and Functions

Why not automatically provide equality for all types of values?

Syntactic equality does not necessarily entail semantic equality!

User data types:

```
data Set a = Set [a]


instance Eq a => Eq (Set a) where
        Set xs == Set ys = xs `subset` ys && ys `subset` xs
                where xs `subset` ys = all (`elem` ys) xs
```

Functions:

```
? (1==) == (\x->1==x)
ERROR: Cannot derive instance in expression
*** Expression       : (==) d148 ((==) {dict} 1) (\x->(==) {dict} 1 x)
*** Required instance : Eq (Int -> Bool)
```

Introduction to Denotational Semantics

Overview:

- Syntax and Semantics
- Approaches to Specifying Semantics
- Semantics of Expressions
- Semantics of Assignment
- Other Issues

Texts:

- D. A. Schmidt, *Denotational Semantics*, Wm. C. Brown Publ., 1986
- D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

Defining Programming Languages

Three main characteristics of programming languages:

1. **Syntax:** What is the appearance and structure of its programs?
2. **Semantics:** What is the *meaning* of programs?

   The *static semantics* tells us which (syntactically valid) programs are semantically valid (i.e., which are *type correct*) and the *dynamic semantics* tells us how to interpret the meaning of valid programs.
3. **Pragmatics:** What is the usability of the language?

   How easy is it to implement? What kinds of applications does it suit?

Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate to the implementors as well as to programmers:

1. *A precise standard for a computer implementation:* How should the language be implemented on different machines?
2. *User documentation:* What is the meaning of a program, given a particular combination of language features?
3. *A tool for design and analysis:* How can the language definition be tuned so that it can be implemented efficiently?
4. *Input to a compiler generator:* How can a reference implementation be obtained from the specification?

Methods for Specifying Semantics

Operational Semantics:

- ☞ ⟦ program ⟧ = abstract machine program
- ☞ can be simple to implement
- ☞ hard to reason about

Denotational Semantics:

- ☞ ⟦ program ⟧ = mathematical denotation (typically, a function)
- ☞ facilitates reasoning
- ☞ not always easy to find suitable semantic domains

Axiomatic Semantics:

- ☞ ⟦ program ⟧ = set of properties
- ☞ good for proving theorems about programs
- ☞ somewhat distant from implementation

Structural Operational Semantics:

- ☞ ⟦ program ⟧ = transition system (defined using inference rules)
- ☞ good for concurrency and non-determinism
- ☞ hard to reason about equivalence

Concrete and Abstract Syntax

How to parse “4 * 2 + 1”?

Abstract Syntax is compact but ambiguous:

Expr	::=	Num
	|	Expr Op Expr
Op	::=	+| - | * | /

Concrete Syntax is unambiguous but verbose:

Expr	::=	Expr LowOp Term
	|	Term
Term	::=	Term HighOp Factor
	|	Factor
Factor	::=	Num
	|	(Expr)
LowOp	::=	+ | -
HighOp	::=	* | /

Semantic Domains

In order to define semantic mappings of programs and their features to their mathematical denotations, the semantic domains must be precisely defined:

```
data Bool = True | False

(&&), (||) :: Bool -> Bool -> Bool
False    &&    x    = False
True     &&    x    = x
False    ||    x    = x
True     ||    x    = True

not :: Bool -> Bool
not      True          = False
not      False         = True
```

A Calculator Language

Abstract Syntax:

```
P   ::=  'ON' S
S   ::=  E 'TOTAL' S      |  E 'TOTAL' 'OFF'
E   ::=  E1 '+' E2        |  E1 '*' E2   |  'IF' E1 ',' E2 ',' E3
     |   'LASTANSWER'     |  '(' E ')'   |  N
```

Test Program = “ ON 4 * (3 + 2) TOTAL OFF ”

Data Structures for Syntax Tree:

```
data Program     = On ExprSequence
data ExprSequence =  Total Expression ExprSequence
                 |   Off Expression
data Expression  =   Plus Expression Expression
                 |   Times Expression Expression
                 |   If Expression Expression Expression
                 |   LastAnswer
                 |   Braced Expression
                 |   N Int

test =       On   (Off  (Times    (N 4)
                                  (Braced   (Plus (N 3)
                                                  (N 2) ) ) ) )
```

Calculator Semantics

Programs:

P *: Program → Int **

P ⟦ **ON** S ⟧ = **S** ⟦ S ⟧ (0)

Sequences:

S *:: ExprSequence → Int → Int **

S ⟦ E **TOTAL** S ⟧ (n) = let n' = **E** ⟦ E ⟧ (n) in n' cons **S** ⟦ S ⟧ (n')

S ⟦ E **TOTAL OFF** ⟧ (n) = **E** ⟦ E ⟧ (n) cons nil

Expressions:

E *: Expression → Int → Int*

E ⟦ E1 **+** E2 ⟧ (n) = **E** ⟦ E1 ⟧ (n) + **E** ⟦ E2 ⟧ (n)

E ⟦ E1 ***** E2 ⟧ (n) = **E** ⟦ E1 ⟧ (n) * **E** ⟦ E2 ⟧ (n)

E ⟦ **IF** E1 , E2 , E3 ⟧ (n) = **E** ⟦ E1 ⟧ (n) == 0 → **E** ⟦ E2 ⟧ (n) # **E** ⟦ E3 ⟧ (n)

E ⟦ **LASTANSWER** ⟧ (n) = n

E ⟦ (E) ⟧ (n) = **E** ⟦ E ⟧ (n)

E ⟦ N ⟧ (n) = N

Implementing the Calculator

Programs:

```
pp :: Program -> [Int]
pp (On s)                   =       ss s 0
```

Sequences:

```
ss :: ExprSequence -> Int -> [Int]
ss (Total e s) n            =       let n' = (ee e n) in n' : (ss s n')
ss (Off e) n                =       (ee e n) : [ ]
```

Expressions:

```
ee :: Expression -> Int -> Int
ee (Plus e1 e2) n                           =       (ee e1 n) + (ee e2 n)
ee (Times e1 e2) n                          =       (ee e1 n) * (ee e2 n)
ee (If e1 e2 e3) n
        |       (ee e1 n) == 0      =       (ee e2 n)
        |       otherwise           =       (ee e3 n)
ee (LastAnswer) n                           =       n
ee (Braced e) n                             =       (ee e n)
ee (N num) n                                =       num
```

A Language with Assignment

Abstract Syntax:

P	::=	C '.'
C	::=	C1 ';' C2
	|	'if' B 'then' C1 'else' C2
	|	I ':=' E
E	::=	E1 '+' E2
	|	I
	|	N
B	::=	E1 '=' E2
	|	'not' B

Example:

“ z := 1 ; if a = 0 then z := 3 else z := z + a . ”

Abstract Syntax Trees

Data Structures:

```
data Program      =  Dot Command
data Command      =  CSeq Command Command
                  |  Assign Identifier Expression
                  |  If BooleanExpr Command Command
data Expression   =  Plus Expression Expression
                  |  Id Identifier
                  |  Num Int
data BooleanExpr  =  Equal Expression Expression
                  |  Not BooleanExpr
type Identifier   =  Char
```

Example:

```
Dot    (CSeq   (Assign 'z' (Num 1))
               (If (Equal (Id 'a') (Num 0))
                        (Assign 'z' (Num 3))
                        (Assign 'z' (Plus (Id 'z') (Id 'a')))
               )
        )
```

Modelling Environments

A store is a mapping from identifiers to values:

```
type Store = Identifier -> Int

newstore :: Store
newstore id                 =       0

access :: Identifier -> Store -> Int
access id store             =       store id

update :: Identifier -> Int -> Store -> Store
update id val store         =       store'
                                    where store' id'
                                    | id' == id         = val
                                    | otherwise         = store id'
```

Semantics of Assignments

```
pp :: Program -> Int -> Int
pp (Dot c) n                = access 'z' (cc c (update 'a' n newstore))

cc :: Command -> Store -> Store
cc (CSeq c1 c2) s           = cc c2 (cc c1 s)
cc (Assign id e) s          = update id (ee e s) s
cc (If b c1 c2) s           = ifelse (bb b s) (cc c1 s) (cc c2 s)

ee :: Expression -> Store -> Int
ee (Plus e1 e2) s           = (ee e2 s) + (ee e1 s)
ee (Id id) s                = access id s
ee (Num n) s                = n

bb :: BooleanExpr -> Store -> Bool
bb (Equal e1 e2) s          = (ee e1 s) == (ee e2 s)
bb (Not b) s                = not (bb b s)

ifelse :: Bool -> a -> a -> a
ifelse True x y = x
ifelse False x y = y
```

Practical Issues

Modelling:

- Errors and non-termination:
  - ☞ need a special “error” value in semantic domains
- Branching:
  - ☞ semantic domains in which “continuations” model “the rest of the program” make it easy to transfer control
- Interactive input
- Dynamic typing
- ...

Theoretical Issues

What are the denotations of lambda abstractions?

- need Scott's theory of semantic domains

What is the semantics of recursive functions?

- need least fixed point theory

How to model concurrency and non-determinism?

- abandon standard semantic domains
- use "interleaving semantics"
- "true concurrency" requires other models ...

Object-Oriented Programming

Overview

- What is Object-Oriented Programming?
- Objects, Classes and Inheritance
- The Principle of Substitutability
- Inheritance and Subtyping
- Dimensions of Object-Oriented Languages

Suggested texts:

- B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.
- R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990
- P. Wegner, “Concepts and Paradigms of Object-Oriented Programming,” ACM OOPS Messenger, Vol. 1, No. 1, Aug. 1990

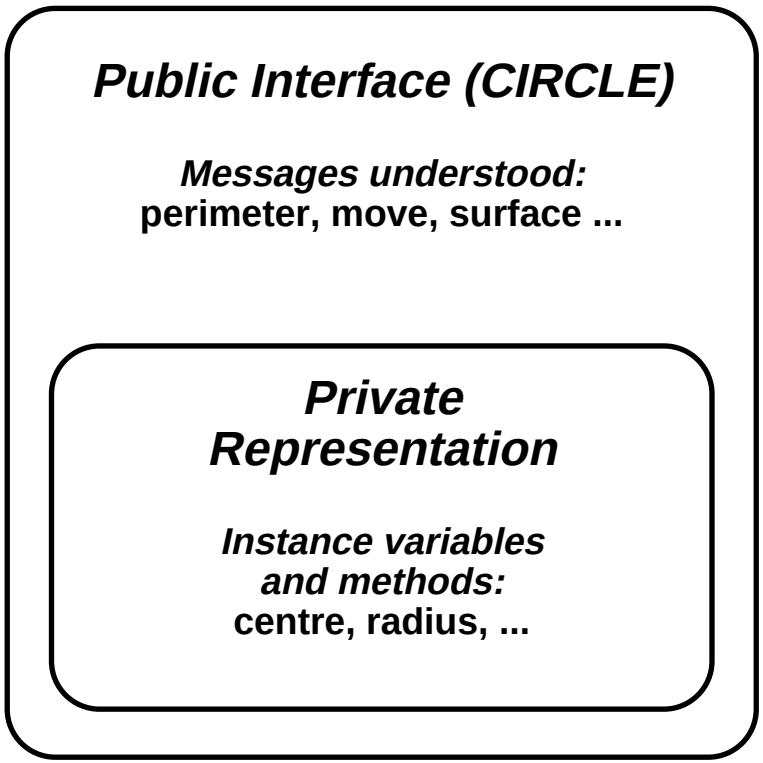
What is Object-Oriented Programming?

Object-oriented programs model applications as collections of communicating objects:

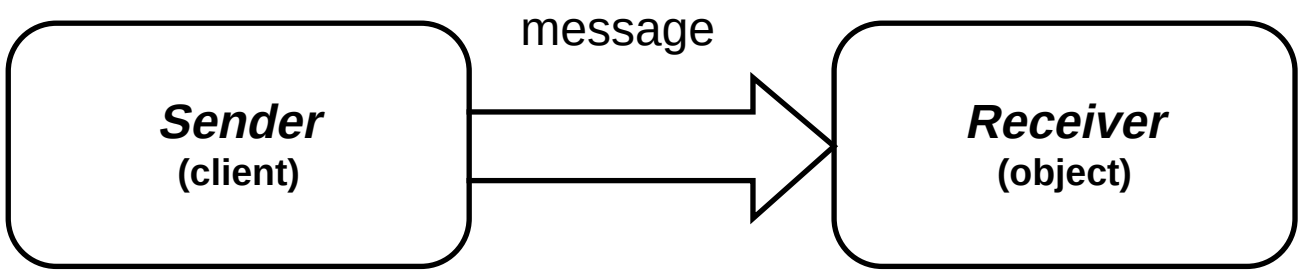
- *Objects* encapsulate data and operations
- Objects implement a client/server *contract*
- Clients may only access an object's services by sending it a *message*
- Objects may have different *methods* to respond to the same set of messages
- *Classes* define templates for instantiating objects
- Classes may *inherit* features from parent classes and extend or modify them
- *Abstract classes* may specify generic interfaces, representation and behaviour, while deferring implementation of features to be defined by concrete subclasses
- *Frameworks* define generic software architectures as hierarchies of related abstract classes

Objects

Objects both *encapsulate* data and the operations that may be performed with them, and they *hide* their internal representation, thus promoting understandability, maintainability and consistency.



Message-Passing Paradigm



a_circle.move(5,10)

Objects can *only* be accessed through their public interface.

A client requests a service of an object by sending it a *"message"* consisting of a service name and some arguments.

The object selects the appropriate *method* to handle the message. Two objects may understand the same messages, but use different methods to respond to them.

An object implements a client/server *contract*.

Classes and Instances

A class describes the implementation of a set of objects.

```
class POINT
    export
        x, y, distance, translate,
    feature
        x: REAL;
        y: REAl;
        Create (a, b: REAL) is
            do
                x := a; y := b;
            end;
        translate(a, b: REAL) is
            do
                x := x + a; y := y + b;
            end;
        distance (other: POINT): REAL is
            do
                Result := sqrt((x - other.x) ^ 2 + (y - other.y) ^ 2)
            end;
end -- class POINT
```

Interface

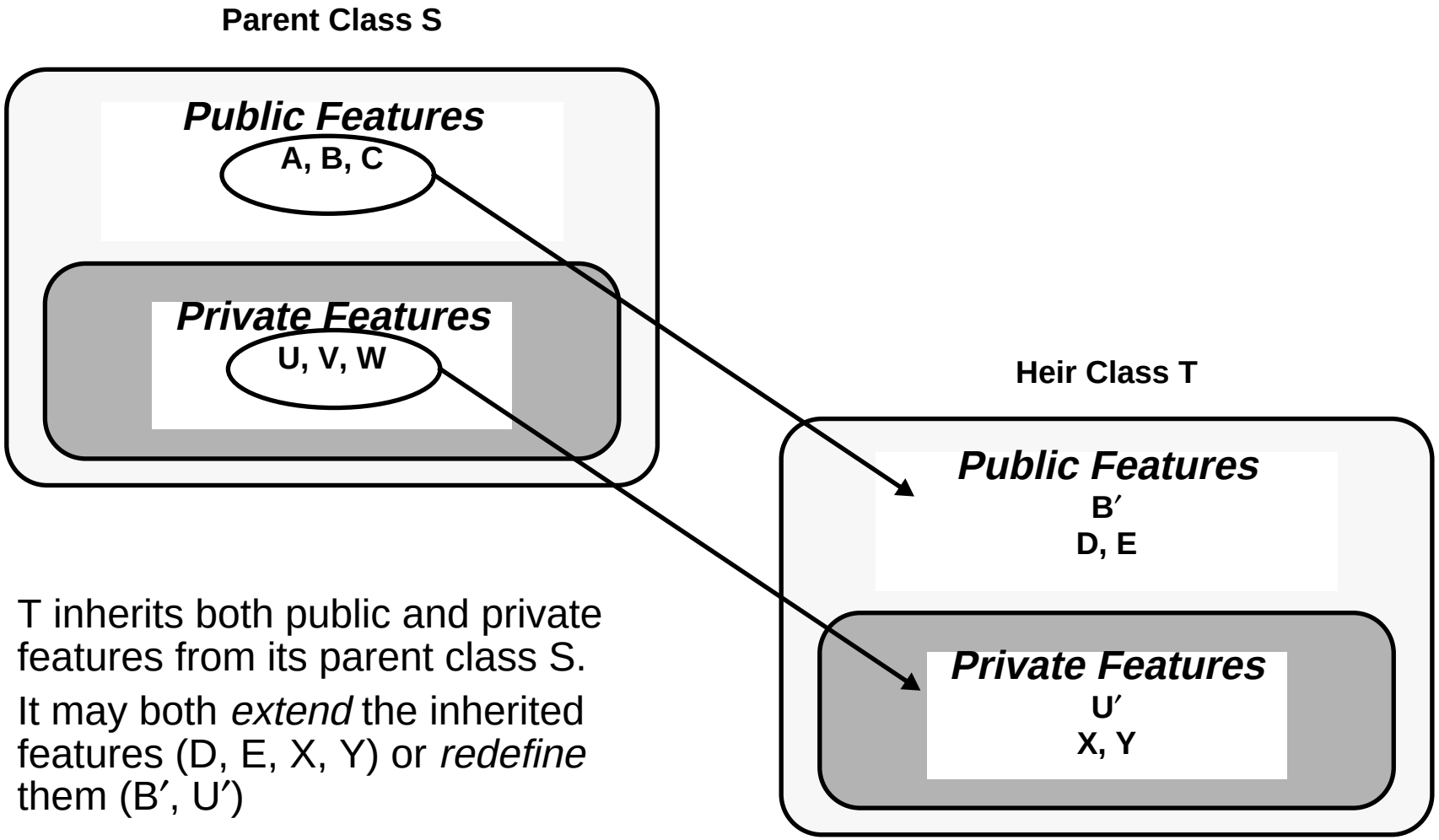
Operation

Instance Variables

Method

An object is an *instance* of a class, sharing the same interface, structure and implementations of methods as other instances of the same class, but with its own private *state* (i.e., its *instance variables*).

Inheritance



T inherits both public and private features from its parent class S.

It may both *extend* the inherited features (D, E, X, Y) or *redefine* them (B′, U′)

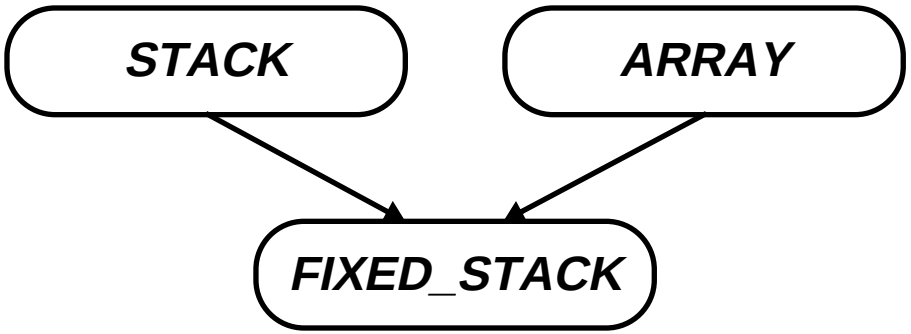
Deferred Features and Classes

Deferred classes define common interfaces and behaviour for a set of implementations

```
deferred class STACK [T]
    export
        nb_elements, empty, full, top, push, pop, change_top, wipe_out
    feature
        nb_elements: INTEGER is deferred end;
        empty: BOOLEAN is
            do
                Result := (nb_elements = 0)
            end;
        full: BOOLEAN is deferred end;
        top: T is deferred end;
        push (v: T) is deferred end;
        pop is deferred end;
        change_top (v: T) is
            do
                pop; push(v)
            end;
        wipe_out is deferred end;
end -- class STACK
```

Multiple Inheritance

Multiple inheritance can be used to combine functionality and implementation:




```
class FIXED_STACK [T]
        export
                max_size, nb_elements, empty, full, top, push, pop, change_top, wipe_out
        inherit
                ARRAY [T]
                        rename      Create as array_Create, size as max_size;
                STACK [T]
                        redefine    change_top
        feature
                ...;
end
```

The Principle of Substitutability

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

— Wegner & Zdonik, ECOOP 88

```
...
move(obj: GRAPHIC_OBJECT, x, y: REAL) is        -- operation of class BIT_MAP_SCREEN
        do
                obj.display_off;                 -- clear from screen
                obj.translate(x, y);
                obj.display_on                   -- display on screen
        end;
...

s: SQUARE; r: RECTANGLE;                         -- both subtypes of GRAPHIC_OBJECT
screen: BIT_MAP_SCREEN;

screen.move(s, 1.5, 1.5); screen.move(r, 1.5, 1.5);
```

Polymorphism & Dynamic Binding

The *static type* of a variable is its declared type.
Its *dynamic type* is the type of the object to which it is currently bound.

```
p: POLYGON;
r: RECTANGLE;

x := p.perimeter;        -- OK
x := r. perimeter;       -- OK

x := r.diagonal;         -- OK
x := p.diagonal;         -- ERROR

p := r;                  -- OK
x := p.perimeter;        -- OK
x := p.diagonal;         -- ERROR

r := p;                  -- ERROR
```

Subtyping

Consider a *type* to be the specification of the interface to an object (i.e., the messages that are understood, together with their argument and return types).

Message send:

- It is always safe to send a message m(a1,...an) understood by instances of type X to an instance of a subtype of X

Assignment:

- It is always safe to assign an instance of a subtype of X to a variable of type X

Subtyping:

- A subtype Y of a type X may add new message types to the interface
- Y may specialize the return type of a message (covariance)

Covariance and Contravariance

Can a subtype also specialize the *argument* types of a message?

```
class VECTOR
        export move, add, ...
        feature
                move (x, y : REAL) : VECTOR ...
                add (v : VECTOR) : VECTOR ...
end
class COLOUREDVECTOR
        export move, add, ...
        feature
                move (x, y : REAL) : COLOUREDVECTOR ...
                add (v : COLOUREDVECTOR) : COLOUREDVECTOR ...        -- add colours too
end
v, v1, v2 : VECTOR;
c : COLOUREDVECTOR;             -- initialized elsewhere ...
v := c;
v1 := v.move (1,3);             -- OK; return type is specialized
v1 := v.add (v2);               -- not OK; can't by sure v2 is a COLOUREDVECTOR!
```

Argument types may only be more general (contravariance) if substitutability is to be guaranteed; but this is seldom useful for solving real problems!

Inheritance is not Subtyping

Various object-oriented programming languages (notably Eiffel and C++) attempt to unify the notions of *types* and *classes*, and therefore constrain inheritance in order to achieve reasonably subtyping rules. This can lead to various conflicts:

- *Covariance vs. contravariance:* for complex modelling problems, it is often convenient to specialize both argument and return types of methods in subclasses, but instances of such subclasses will not be substitutable for superclass instances.
- *Multiple inheritance:* sometimes multiple inheritance is used to combine an abstract interface with a particular representation (implementation reuse). This may necessitate renaming (hiding) of features inherited from the representation class, which violates any reasonable subtyping rule tied to inheritance.
- *Post-hoc type equivalence:* separately defined classes may actually have compatible types, though they do not share any common superclass.

The Inheritance Interface

A class has two different kinds of clients: run-time clients of their instances, and inheriting classes.

The interface to run-time clients is defined by the *exports* declaration in the class. The interface to heirs (subclasses) is defined by the programming language:

- Heirs have full access to the implementation of parents
- Heirs may only access the public features of parents
- Heirs may only access features exported in an *inheritance interface*

Run Time Support

- **Garbage collection:** memory occupied by objects that are no longer referenced may be automatically reclaimed
- **Persistence:** objects may be automatically committed to persistent storage
- **Distribution:** objects may be shared within a distributed environment
- **Reflection:** class definitions may be accessed and (self-) modified at run-time
- **Concurrency:** multiple objects may be concurrently active; individual objects may manage multiple concurrent threads

Dimensions of Object-Oriented Languages

- **Object-Based** languages support *encapsulation* of behaviour and state (objects)
- **Class-Based** languages support *instantiation* of objects from object classes
- **Object-Oriented** languages support *inheritance* between classes
- **Fully Object-Oriented** languages model *all* data types as objects; classes are also objects
- **Strongly-Typed** object-oriented languages guarantee that all expressions are type-consistent
- **Concurrent** object-oriented languages allow multiple objects to serve requests concurrently; individual objects can schedule and synchronize concurrent requests
- **Persistent** object-oriented languages support objects whose lifetime may span multiple user sessions

— Wegner, OOPS Messenger, Vol. 1, #1

A Brief History of OO Languages

- **Simula** (1962): extended Algol with *classes* and *inheritance*; designed for writing simulation applications
- **Smalltalk** (1970s): *"pure"* OOPL; developed by Xerox PARC to drive graphic workstations
- **Modules** (1972): Parnas promoted encapsulation and *information hiding*
- **Abstract Data Types** (1974): Liskov and Zilles promoted formal specification
- **Ada** (1983)
- **Objective C, Beta, etc.**(1980s)
- **C++, Eiffel** (1986)
- **Emerald, ABCL, ConcurrentSmalltalk, Oz ...** *and many others*

Current Trends in Research and Practice

- Objects + X *where X is ...*
- Object-based concurrency
- Type theories for objects (mostly functional)
- Semantic models of objects (both functional and non-functional)
- Components
- Distribution and Interoperability (CORBA and ODP)
- Frameworks
- Design Patterns
- Role Modelling

Logic Programming

Overview

- Facts and Rules
- Searching and Backtracking
- Recursion, Functions and Arithmetic
- Lists and other Structures
- Implementing a Simple Interpreter

Texts:

- Sterling and Shapiro, *The Art of Prolog*, MIT Press, 1986
- Clocksin and Mellish, *Programming in Prolog*, Springer Verlag, 1981

Facts and Rules

A Prolog program consists of *facts*, *rules*, and *questions*:

- *Facts* are named relations between objects:
  - parents(charles, elizabeth, philip).
- *Rules* are relations (goals) that can be inferred from other relations (subgoals):
  - uncle(U,C) :- brother(U,P), parent(P,C).
- Both rules and facts are instances of *Horn clauses,* of the form:
  - A_0 if A_1 and A_2 and ... A_n
- *Questions* are statements that can be answered using facts and rules:
  - ? brother(charles, X)
- Questions are answered by *matching* goals against facts or rules, *unifying* variables with terms, and *backtracking* when subgoals fail
- A question is always answered with **true** or **false**, given some binding of variables to terms
- Prolog adopts a *closed world assumption* — whatever cannot be proved to be true, is assumed to be false

Prolog Databases

```
male(philip).
female(elizabeth).
male(charles).
female(anne).
male(andrew).
male(edward).
female(diana).
male(william).
male(harry).
parents(charles, elizabeth, philip).
parents(anne, elizabeth, philip).
parents(andrew, elizabeth, philip).
parents(edward, elizabeth, philip).
parents(william, diana, charles).
parents(harry, diana, charles).
```

```
?- male(charles).
?- male(anne).
?- male(mickey)
?- male(X).
?- parents(X,elizabeth,_).
```

Rules, Searching and Backtracking

A Rule defines a relation as a conjunction of subgoals:

```
brother(X, Y) :-    male(X),
                    parents(X, M, F),
                    parents(Y, M, F),
                    X \== Y.
```

?- brother(charles, edward).

?- brother(charles, X).

?- brother(X, charles).

Conjunctions and Disjunctions

The same information can be represented in various forms:

```
mother(M,C) :-   parents(C,M,_).
father(F,C) :-   parents(C,_,F).
```

We could have chosen to represent parents/3 in terms of mother/2 and father/2:

```
parents(C,M,F) :- mother(M,C), father(F,C).
```

Both conjunctions and disjunctions can be easily represented:

```
uncle(U,C) :-    brother(U,P),
                 parent(P,C).

parent(P,C) :-   mother(P,C).
parent(P,C) :-   father(P,C).
```

Recursion

Recursive relations are defined in the obvious way:

```
ancestor(A,P) :-  parent(A,P).
ancestor(A,P) :-  parent(A,C),
                  ancestor(C,P).
```

```
?- ancestor(philip, harry).
?- ancestor(philip, X).
?- ancestor(X, harry).
```

Negation as Failure

Searching can be controlled by explicit failure:

```
printall(X) :-      X, print(X), nl, fail.
printall(_).

?- printall(brother(_,_)).
```

The *cut* operator (!) commits Prolog to a particular search path:

```
parent(P,C) :-      mother(P,C), !.
parent(P,C) :-      father(P,C).
```

Negation can be implemented by a combination of cut and fail:

```
not(X) :-           X, !, fail.
not(_).
```

Changing the Database

The Prolog database can be modified dynamically by means of assert and retract:

```
changename(X,Y) :-      rename(X,Y),
                        retract(parents(X,M,F)),
                        assert(parents(Y,M,F)).

rename(X,Y) :-          retract(male(X)),
                        assert(male(Y)).
rename(X,Y) :-          retract(female(X)),
                        assert(female(Y)).
```

```
?- changename(charles, mickey).
```

Functions and Arithmetic

Functions are relations between expressions and values:

```
X is 5 + 6 .
```

Yields:

```
X = 11 ?
```

And is syntactic sugar for:

```
is(X, +(5,6))
```

User-defined functions are written in a relational style:

```
fact(0,1).
fact(N,F) :-    N > 0,
                N1 is N - 1,
                fact(N1,F1),
                F is N * F1.
```

Lists

Lists are pairs of elements and lists:

Formal object	*Cons pair syntax*	*Element syntax*
.(a , [ ])	[a | [ ]]	[a]
.(a , .(b, [ ]))	[a | [b | [ ]]]	[a , b]
.(a , .(.(b , [ ]) , .(c , [ ])))	[a | [[b | [ ]] | [c | [ ]]]]	[a , [b] , c]
.(a , X)	[a | X]	[a | X]
.(a , .(b , X))	[a | [b | X]]	[a , b | X]

Pattern Matching with Lists

```
member(X, [X | _ ]).
member(X, [ _ | L]) :-        member(X, L).

?- member(a, [a,b,c]).

?- member(X, [a,b,c]).

?- member(a, L).
        L = [ a | _A ] ? ;
        L = [ _A , a | _B ] ? ;
        L = [ _A , _B , a | _C ] ? ;
        L = [ _A , _B , _C , a | _D ] ? ;
```

Exhaustive Searching

Searching for permutations:

```
perm([ ],[ ]).
perm([C|S1],S2) :-          perm(S1,P1),
                            append(X,Y,P1),
                            append(X,[C|Y],S2).

append([ ],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3) .


?- printall(perm([a,b,c,d],_)).
```

A declarative, but hopelessly inefficient sort program:

```
ndsort(L,S) :-              perm(L,S),
                            issorted(S).

issorted([ ]).
issorted([ _ ]).
issorted([N,M|S]) :-        N =< M,
                            issorted([M|S]).
```

Operators

Calculator example [Schmidt]:

```
P   ::=  'on' S
S   ::=  E 'total' S        |   E 'total' 'OFF'
E   ::=  E1 '+' E2          |   E1 '*' E2   |   'if' E1 'then' E2 'else' E3
     |   'lastanswer'       |   '(' E ')'   |   N
```

Syntax trees can be modelled directly as Prolog terms.

Operator type and precedence can be defined to achieve convenient syntax:

```
:- op(900, fx, on).
:- op(800, xfy, total).
:- op(600, fx, if).
:- op(590, xfy, then).
:- op(580, xfy, else).
% op(500, yfx, +).
% op(400, yfx, *).

on 2+3 total lastanswer + 1 total off          = on(total(2+3, total(lastanswer+1, off)))

on if lastanswer then 3*4 else 3+4 total off   = on(total(if(then(lastanswer, else(3*4, 3+4))), off))
```

Building a Simple Interpreter

Top level programs:

```
on S :- seval(S, 0).
```

Statements:

```
seval(E total off, Prev) :-     xeval(E, Prev, Val),
                                print(Val), nl.
seval(E total S, Prev) :-       xeval(E, Prev, Val),
                                print(Val), nl,
                                seval(S, Val).
```

Expressions:

```
xeval(N, _, N) :-               number(N).
xeval(E1+E2, Prev, V) :-        xeval(E1, Prev, V1),
                                xeval(E2, Prev, V2),
                                V is V1+V2.
xeval(E1*E2, Prev, V) :-        xeval(E1, Prev, V1),
                                xeval(E2, Prev, V2),
                                V is V1*V2.
xeval(lastanswer, Prev, Prev).
xeval(if E1 then E2 else _, Prev, Val) :-  xeval(E1, Prev, 0),   !,
                                           xeval(E2, Prev, Val).
xeval(if _ then _ else E3, Prev, Val) :-   xeval(E3, Prev, Val).
```

Concurrent Programming

Overview

- Concurrency issues
- Process creation
- Synchronizing access to shared variables
- Message Passing Approaches

Texts:

- G. R. Andrews and F. B. Schneider, “Concepts and Notations for Concurrent programming,’“ACM Computing Surveys, vol. 15, no. 1, Mar. 1983, pp. 3-43.
- M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall, 1990.
- L. Wilson & R. Clark, *Comparative Programming Languages*, Addison-Wesley, 1988.

Concurrency and Parallelism

"A *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*."

A concurrent program can be executed by:

1. *Multiprogramming:* processes share one or more processors
2. *Multiprocessing*: each process runs on its own processor but with shared memory
3. *Distributed processing:* each process runs on its own processor connected by a network to others

Assume only that all processes make positive finite progress.

Atomicity

Programs P1 and P2 execute concurrently:

```
        { x = 0 }
P1:     x := x+1
P2:     x := x+2
        { x = ? }
```

What are possible values of x after P1 and P2 complete?
What is the *intended* final value of x?

Synchronization mechanisms are needed to restrict the possible interleavings of processes so that sets of actions can be seen as atomic.

Mutual exclusion ensures that statements within a *critical section* are treated atomically.

Concurrency Issues

There are two principal difficulties in implementing concurrent programs:

- Ensuring consistency:
  - *Mutual exclusion* — shared resources must be updated atomically
  - *Condition synchronization* — operations may need to be delayed if shared resources are not in an appropriate state (e.g., read from empty buffer)
- Ensuring progress:
  - *Deadlock* — some process can always access a shared resource
  - *Starvation* — all processes can eventually access shared resources

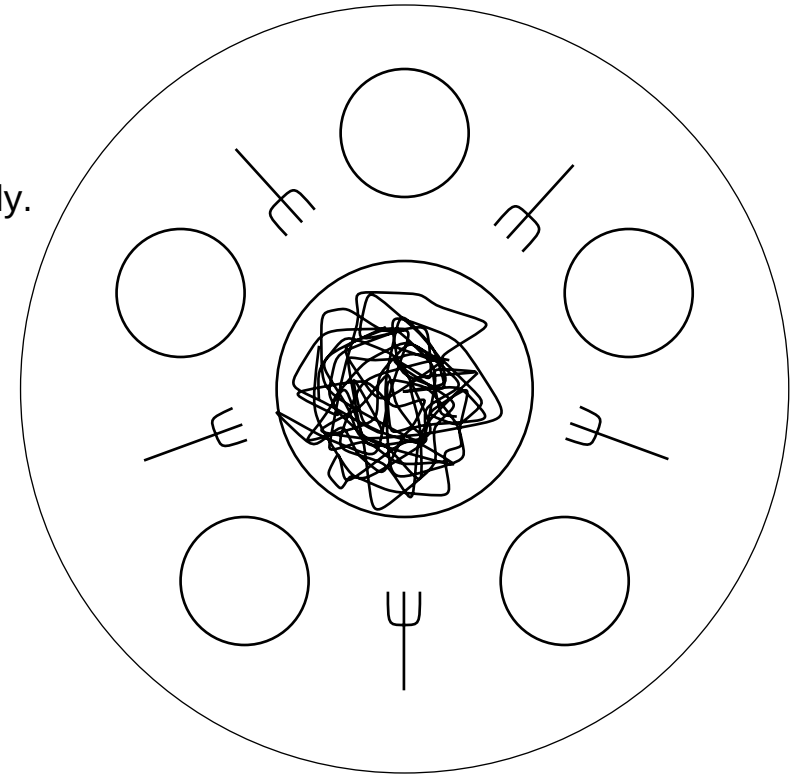
Notations for expressing concurrent computation must address:

1. **Process Creation:** how is concurrent execution specified?
2. **Communication:** how do processes communicate?
3. **Synchronization:** how is consistency maintained?

Deadlock and Starvation

Dining Philosophers

Philosophers alternate between thinking and eating.
A philosopher needs two forks to eat.
No two philosophers may hold the same fork simultaneously.
No deadlock and no starvation.
Efficient behaviour under absence of contention.



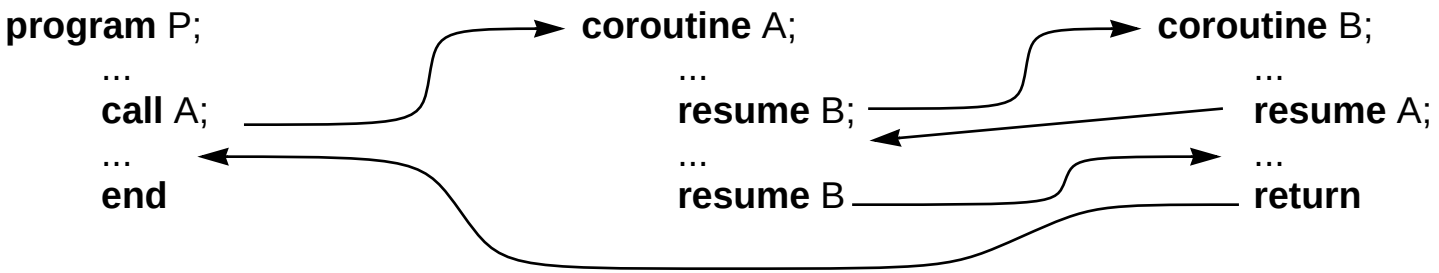
Fairness

There are subtle differences between definitions of fairness:

- **Weak fairness:** If a process *continuously* makes a request, *eventually* it will be granted.
- **Strong fairness:** If a process makes a request *infinitely often*, *eventually* it will be granted.
- **Linear waiting:** If a process makes a request, it will be granted before any other process is granted the request more than once.
- **FIFO (first-in first out):** If a process makes a request, it will be granted before that of any process making a *later* request.

Process Creation

Co-routines:



Fork and Join:

```
program P1
    ...
    fork P2;
    ...
    join P2;
    ...
```

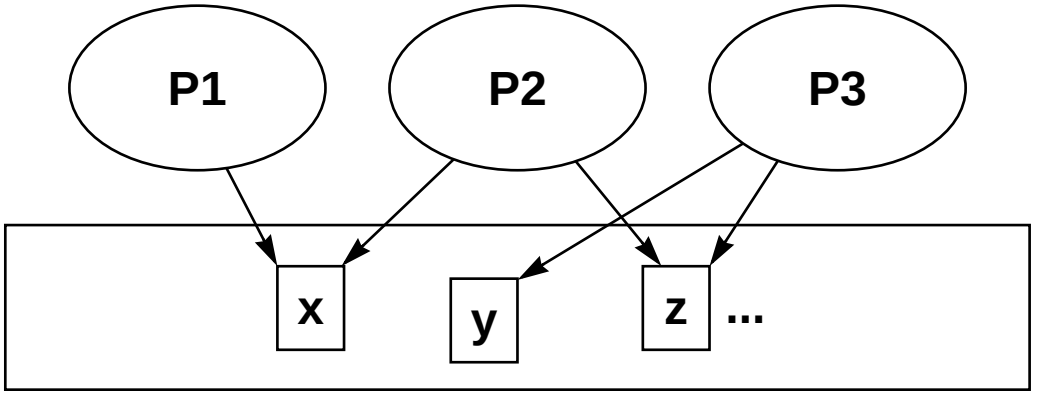
```
program P2
    ...
    ...
    end
```

Cobegin:

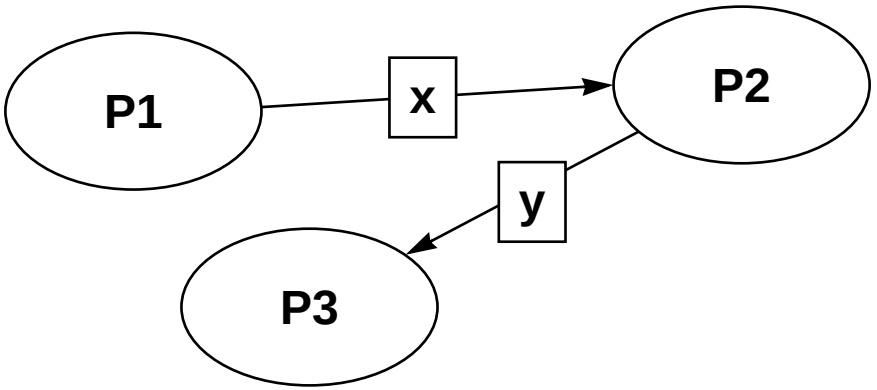
cobegin S1 || S2 || ... || Sn **coend**

Communication and Synchronization

Shared Variables:

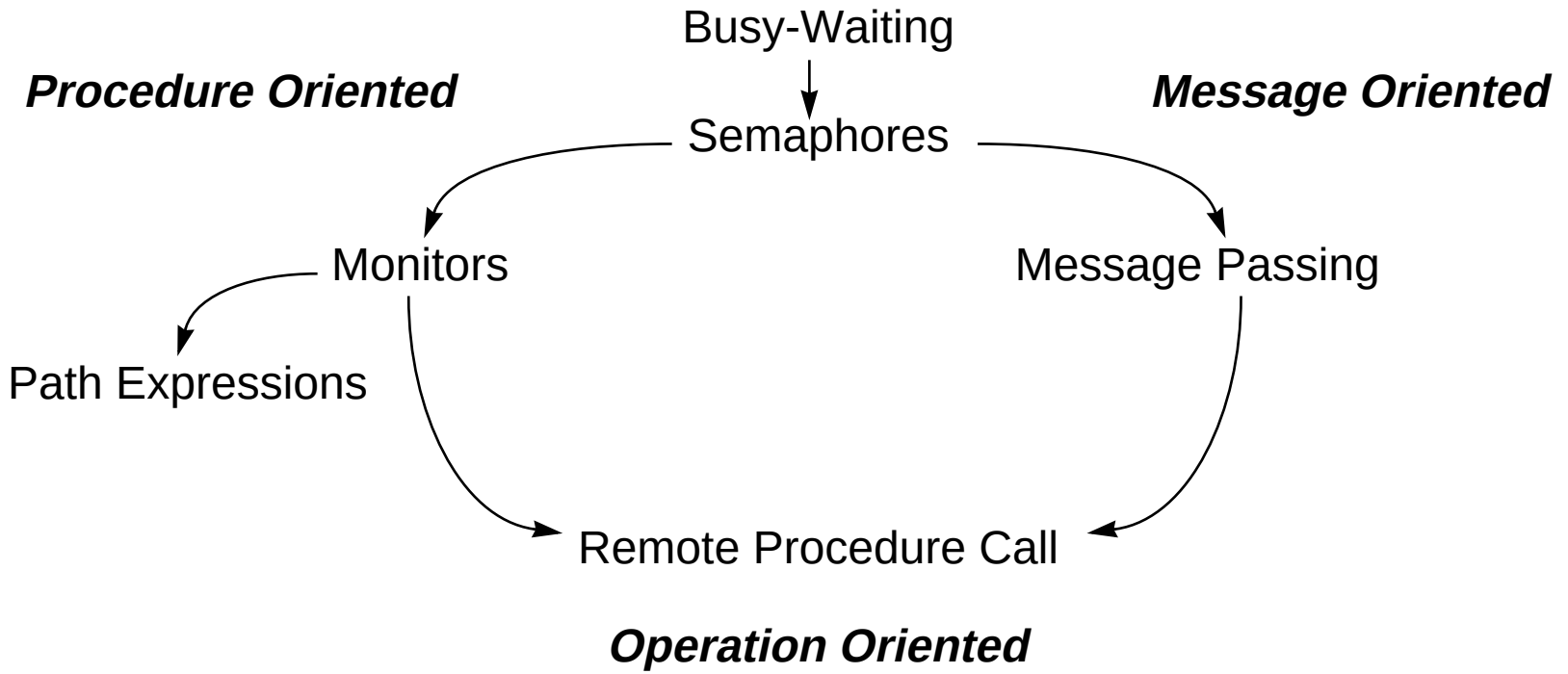


Message-Passing:



Synchronization Techniques

Different approaches are roughly equivalent in expressive power and can generally be implemented in terms of each other.



Each approach emphasizes a different style of programming.

Busy-Waiting

A simple approach to implement synchronization is to have processes set and test shared variables. Condition synchronization is easy to implement, but mutual exclusion is more difficult to realize *correctly* and *efficiently*.

Condition synchronization:

- to signal a condition, a process sets a shared variables (bufferEmpty = FALSE)
- to wait for a condition, a process repeatedly tests the variable

Mutual exclusion:

- condition variables are used to implement *entry* and *exit protocols* to access and release shared resources

```
process P1;
    loop
        enter1 := true;     { wants to enter }
        turn := "P2";       { but yields priority }
        while enter2 and turn = "P2"
            do skip;
        Critical Section;
        enter1 := false;    { exits }
        Non-critical Section;
    end;
end;
```

```
process P2;
    loop
        enter2 := true;
        turn := "P1";
        while enter1 and turn = "P1"
            do skip;
        Critical Section;
        enter2 := false;
        Non-critical Section;
    end;
end;
```

Semaphores

Semaphores were introduced by Dijkstra (1968) as a higher-level primitive for process synchronization.

A *semaphore* is a non-negative integer-valued variable *s* with two operations:

- **P**(s): delays until s>0; when s>0, *atomically* executes s := s-1
- **V**(s): atomically executes s:= s+1

Many problems can be solved using *binary semaphores*, which take on values 0 or 1.

```
process P1;
    loop
        P(mutex);          { wants to enter }
        Critical Section;
        V(mutex);          { exits }
        Non-critical Section;
    end;
end;
```

```
process P2;
    loop
        P(mutex);
        Critical Section;
        V(mutex);
        Non-critical Section;
    end;
end;
```

Monitors

A *monitor* encapsulates resources and operations that manipulate them:

- operations are invoked with usual procedure call semantics
- procedure invocations are guaranteed to be mutually exclusive
- condition synchronization is realized using *signal* and *wait* primitives
  - ☞ there exist many variations of *wait* and *signal* ...

```
type buffer(T) = monitor
        var
        slots : array [0..N-1] of T;
        head, tail : 0..N-1;
        size : 0..N;
        notfull, notempty : condition;

procedure deposit(p : T);
        begin
                if size = N then notfull.wait
                slots[tail] := p;
                size := size + 1;
                tail := (tail+1) mod N;
                notempty.signal
        end

procedure fetch(var it : T);
        begin
                if size = 0 then notempty.wait
                it := slots[head];
                size := size - 1;
                head := (head+1) mod N;
                notfull.signal
        end

begin
        size := 0; head := 0; tail := 0;
end
```

Problems with Monitors

Although monitors provide a more structured approach to process synchronization than semaphores, they suffer from various shortcomings.

A signalling process is temporarily *suspended* to allow waiting processes to enter!

- Monitor state may change between *signal* and resumption of signaller
- Simultaneous *signal* and *return* is not supported
- Unlike semaphores, multiple signals are not saved
- Boolean expressions are not explicitly associated to condition variables
- Nested monitor calls must be specially handled to prevent deadlock

Message Passing

Message Passing combines both communication and synchronization:

- A message is *sent* by specifying the *message* and a *destination*
  - The destination may be a process, a port, a set of processes, ...
- A message is *received* by specifying message *variables* and a *source*
  - The source may or may not be explicitly identified
  - Source and destination may be statically fixed or dynamically computed
- Message transfer may be *synchronous* or *asynchronous*
  - With *asynchronous* message passing, send operations never block
  - With *buffered* message passing, sent messages pass through a bounded buffer ; the sender may block if the buffer is full
  - With *synchronous* message passing, both the sender and receiver must be ready for a message to be exchanged

Unix Pipes

Unix pipes are bounded buffers that connect producer and consumer processes (*sources*, *sinks* and *filters*):

```
cat file
| tr -c ‘a-zA-Z’ ‘\012’
| sort
| uniq -c
| sort -rn
| more
```

Processes should read from standard input and write to standard output streams.

Process creation and scheduling are handled by the O/S, and synchronization is handled implicitly by the I/O system.

Send and Receive

In CSP or Occam, source and destination are explicitly named:

```
PROC buffer(CHAN OF INT give, take, signal)
  VAL INT size IS 10:
  INT inindex, outindex, numitems:
  [size]INT thebuffer:
  SEQ
    numitems := 0
    inindex := 0
    outindex := 0
    WHILE TRUE
    ALT
      numitems ≤ size & give ? thebuffer[inindex]
        SEQ
          numitems := numitems + 1
          inindex := (inindex + 1) REM size
      numitems > 0 & signal ? any
        SEQ
          take ! thebuffer[outindex]
          numitems := numitems - 1
          outindex := (outindex + 1) REM size
```

Remote Procedure Calls and Rendezvous

In Ada, the caller identity need not be known in advance:

```
task body buffer is
        size : constant integer := 10;
        the_buffer : array (1 .. size) of item;
        no_of_items : integer range 0 .. size := 0;
        in_index, out_index : integer range 1 .. size := 1;
begin
        loop
                select
                        when no_of_items < size =>
                                accept give(x : in item) do
                                        the_buffer(in_index) := x;
                                end give;
                                no_of_items := no_of_items + 1;
                                in_index := in_index mod size + 1;
                or      when no_of_items > 0 =>
                                accept take(x : out item) do
                                        x := the_buffer(out_index);
                                end take;
                                no_of_items := no_of_items - 1;
                                out_index := out_index mod size + 1;
                end select;
        end loop;
end buffer;
```

Other Issues

Atomic Transactions:

- RPC with possible failures
- failure atomicity
- synchronization atomicity

Real-Time Programming:

- embedded systems
- responding to interrupts within strict time limits

Process Calculi

Overview

- SOS Style
- Process calculi and transition semantics
- A tiny language with pure synchronization
- Implementing the transition semantics
- Value passing across channels
- Replicated processes

Texts:

- R. Milner, *Communication and Concurrency*, Prentice Hall, 1989
- B. Pierce, *Programming in the Pi-Calculus*, Tutorial Notes for PICT Version 3.6a, 1995
- G. Kahn, “Natural Semantics,” INRIA Report No. 601, Feb. 1987

Limitations of Denotational Semantics

Denotational Semantics:

- ☞ ⟦ program ⟧ = function from program input to output

Concurrent programs are not functions

- ☞ Input and output are on-going
- ☞ Same input may produce different results at different times
- ☞ Concurrent inputs may produce non-deterministic results
- ☞ Correct programs may not terminate
- ☞ “True concurrency” cannot be captured by interleaving

Structural Operational Semantics

SOS Style:

- ☞ ⟦ program ⟧ = logical inferences about the program

$$\frac{\text{fact} \qquad \text{fact}}{\text{fact}}$$

Transition Semantics:

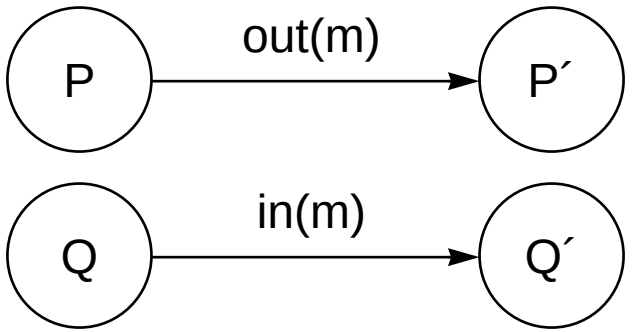
- ☞ Facts are statements about possible transitions from program states (represented as expressions) to other states

Natural Semantics:

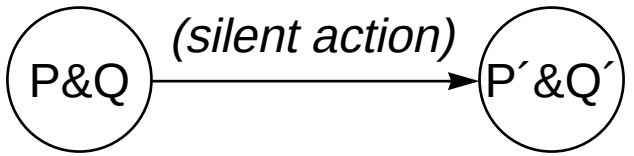
- ☞ Facts take the form: $E \vdash$ c where E is an environment and c is a statement about a program fragment

Transition Semantics

Concurrent processes can be viewed as *state machines* that evolve by *named transitions* to new states. A concurrent system can be viewed as a *composition* of processes whose possible transitions are synchronized as actions.



⇒



Process Calculi

A process calculus is a formal language for describing concurrent processes together with its transition semantics.

- processes evolve by synchronizing communications along named channels
- concurrency is reduced to:
  - input, output, choice, hiding/renaming, composition, replication
- close affinity with the lambda calculus:
  - a function is a process with only one input channel called "λ"
  - minimal syntax and inference rules
- pioneered by Milner (CCS: Calculus of Communicating Systems) and by Hoare (CSP: Communicating Sequential Processes

Pure Synchronization

A tiny process calculus: P ::= C?P | C!P | P&P | nil

$$\frac{}{C!P \xrightarrow{out(C)} P} \qquad \frac{}{C?P \xrightarrow{in(C)} P}$$

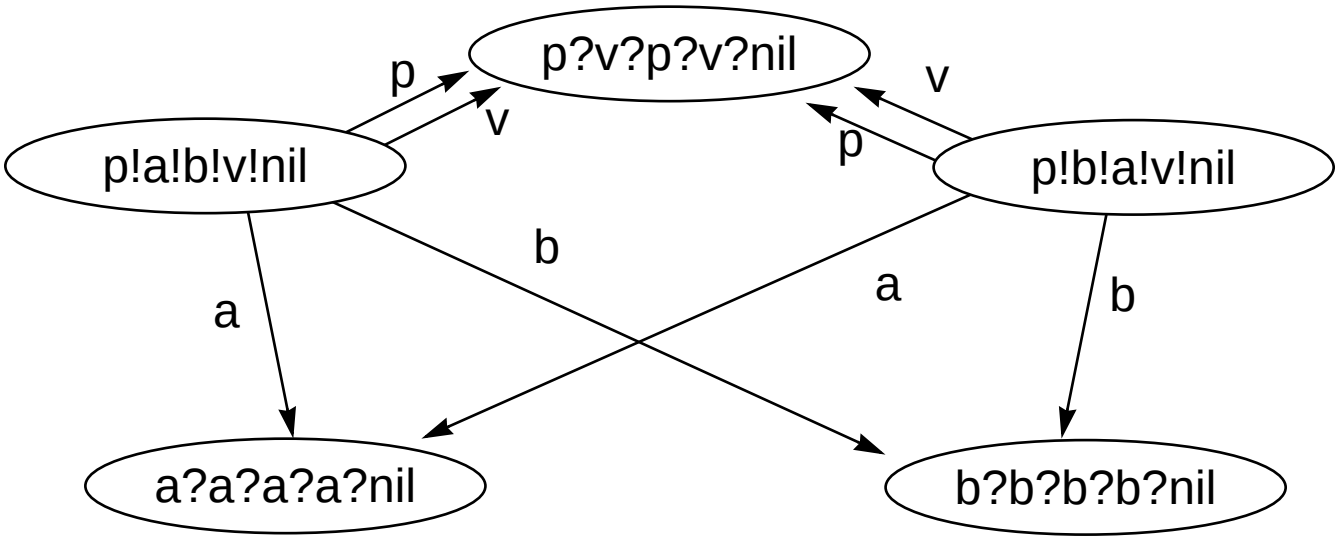
$$\frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \,\&\, Q \xrightarrow{\tau} P' \,\&\, Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \,\&\, Q \xrightarrow{\alpha} P' \,\&\, Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \,\&\, Q \xrightarrow{\alpha} P \,\&\, Q'}$$

NB: $\overline{out(C)} = in(C)$, $out(C) = \overline{in(C)}$; τ stands for a "silent" action.

Modeling Non-determinism

	p!a!b!v!nil	— *a client of resources a and b*
&	p!b!a!v!nil	— *a competing client*
&	p?v?p?v?nil	— *a (non-reusable) semaphore*
&	a?a?a?a?nil	— *a server for resource a*
&	b?b?b?b?nil	— *a server for resource b*



Implementing the Transition Semantics

```
:-  op(700,xfy,&).      % concurrent composition
:-  op(600,xfy,?).      % input
:-  op(600,xfy,!).      % output

out(Channel!Process, Channel, Process).

out(P & Q, Comm, P & R)   :-  out(Q, Comm, R).
out(P & Q, Comm, R & Q)   :-  out(P, Comm, R).

in(Channel?Process, Channel, Process).

in(P & Q, Comm, P & R)    :-  in(Q, Comm, R).
in(P & Q, Comm, R & Q)    :-  in(P, Comm, R).

act(P1&Q1, Comm, P2&Q2)   :-  out(P1,Comm,P2),
                              in(Q1,Comm,Q2).

act(P1&Q1, Comm, P2&Q2)   :-  in(P1,Comm,P2),
                              out(Q1,Comm,Q2).

act(P&Process, Comm, P&NewProcess)   :-  act(Process, Comm, NewProcess) .
act(Process&P, Comm, NewProcess&P)   :-  act(Process, Comm, NewProcess) .
```

Searching for Executions Paths

```
:-      op(900,xfx,===>).       % actions till stop

P ===> End                      :-      act(P,Comm,R),
                                            print(''), print(Comm), nl,
                                            print('=> '), print(R), nl,
                                        R ===> End.

P ===> P                        :-      dead(P).

dead(P)                         :-      act(P,_,_), !, fail.
dead(_).
```

Running the Example

| ?- p!a!b!v!nil&p!b!a!v!nil & p?v?p?v?nil & a?a?a?a?nil & b?b?b?b?nil ===> X.

p

=> a!b!v!nil & p!b!a!v!nil & v?p?v?nil & a?a?a?a?nil & b?b?b?b?nil

a

=> b!v!nil & p!b!a!v!nil & v?p?v?nil & a?a?a?nil & b?b?b?b?nil

b

=> v!nil & p!b!a!v!nil & v?p?v?nil & a?a?a?nil & b?b?b?nil

v

=> nil & p!b!a!v!nil & p?v?nil & a?a?a?nil & b?b?b?nil

p

=> nil & b!a!v!nil & v?nil & a?a?a?nil & b?b?b?nil

b

=> nil & a!v!nil & v?nil & a?a?a?nil & b?b?nil

a

=> nil & v!nil & v?nil & a?a?nil & b?b?nil

v

=> nil & nil & nil & a?a?nil & b?b?nil

Finding Alternative Execution Paths

```
X = (nil & nil & nil & a?a?nil & b?b?nil) ? ;


        p
=> p!a!b!v!nil & b!a!v!nil & v?p?v?nil & a?a?a?a?nil & b?b?b?b?nil
        b
=> p!a!b!v!nil & a!v!nil & v?p?v?nil & a?a?a?a?nil & b?b?b?nil
        a
=> p!a!b!v!nil & v!nil & v?p?v?nil & a?a?a?nil & b?b?b?nil
        v
=> p!a!b!v!nil & nil & p?v?nil & a?a?a?nil & b?b?b?nil
        p
=> a!b!v!nil & nil & v?nil & a?a?a?nil & b?b?b?nil
        a
=> b!v!nil & nil & v?nil & a?a?nil & b?b?b?nil
        b
=> v!nil & nil & v?nil & a?a?nil & b?b?nil
        v
=> nil & nil & nil & a?a?nil & b?b?nil

X = (nil & nil & nil & a?a?nil & b?b?nil) ? ;

no
```

An Asynchronous Value-Passing Calculus

P	::=	C?X>P	| C!V	| P&P	| nil
V	::=	[ ]	| [C]		
X	::=	[ ]	| [C]	| [_]	

$$\frac{}{C!V \xrightarrow{out(C!V)} nil} \qquad \frac{}{C?X{>}P \xrightarrow{in(C!V)} P\{V/X\}}$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P \,\&\, Q \xrightarrow{\tau} P' \,\&\, Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \,\&\, Q \xrightarrow{\alpha} P' \,\&\, Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \,\&\, Q \xrightarrow{\alpha} P \,\&\, Q'}$$

Implementing Value Passing

```
out(Channel!Message, Channel!Message, nil).

out(P & Q, Comm, P & R)          :-   out(Q, Comm, R).
out(P & Q, Comm, R & Q)          :-   out(P, Comm, R).

in(Channel?Pattern>AbsProcess, Channel!Message, NewProcess)
                 :-   AbsProcess @ {Message/Pattern} –-> NewProcess .

in(P & Q, Comm, P & R)           :-   in(Q, Comm, R).
in(P & Q, Comm, R & Q)           :-   in(P, Comm, R).

act(P1&Q1, Comm, P2&Q2)          :-   out(P1,Comm,P2), in(Q1,Comm,Q2).
act(P1&Q1, Comm, P2&Q2)          :-   in(P1,Comm,P2), out(Q1,Comm,Q2).

act(P&Process, Comm, P&NewProcess)   :-   act(Process, Comm, NewProcess) .
act(Process&P, Comm, NewProcess&P)   :-   act(Process, Comm, NewProcess) .
```

For convenience:

```
out(N, Comm, R)                  :-   N := P, !, out(P, Comm, R).
in(N, Comm, R)                   :-   N := P, !, in(P, Comm, R).
act(N, Comm, R)                  :-   N := P, !, act(P, Comm, R).
```

Implementing Substitution

```
D @ {XL/NL} –-> E                    :-  D := P, !, P @ {XL/NL} –-> E .

M @ {[ _ ]/[N]} –-> M                :-  atom(M), M \== N, !.
Expr @ {[ ]/[ ]} –-> Expr .

N @ {[X]/[N]} –-> X                  :-  !.
[N] @ {[X]/[N]} –-> [X] .

CE?AProc @ {XL/NL} –-> C?Proc        :-  CE @ {XL/NL} –-> C,
                                         AProc @ {XL/NL} –-> Proc.

CE!AMsg @ {XL/NL} –-> C!Msg          :-  CE @ {XL/NL} –-> C,
                                         AMsg @ {XL/NL} –-> Msg.

P&Q @ {XL/NL} –-> PR & QR            :-  P @ {XL/NL} –-> PR,
                                         Q @ {XL/NL} –-> QR.

Pat>Abs @ {XL/NL} –-> PatR>AbsR      :-  Pat @ {XL/NL} –-> PatR,
                                         Abs @ {XL/NL} –-> AbsR.
```

NB: The rule for input channels is not quite right — why not?

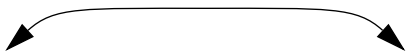
A Value-Passing Example

| ?- (a?[r]>r![ ] & a?[r]>r![ ] & a?[r]>r![ ]) & a![b] & (b?[ ]>nil) & a![c] & (c?[ ]>nil) ===> X.

a![c]

=> (c![ ]&a?[c]>c![ ]&a?[c]>c![ ])&a![b]&(b?[ ]>nil)&nil&c?[ ]>nil

c![ ]



=> (nil&a?[c]>c![ ]&a?[c]>c![ ])&a![b]&(b?[ ]>nil)&nil&nil

a![b]

=> (nil&b![ ]&a?[b]>b![ ])&nil&(b?[ ]>nil)&nil&nil

b![ ]

=> (nil&nil&a?[b]>b![ ])&nil&nil&nil&nil

NB: substituting bound names works here, but not in general. Why not?

Process Replication

P	::=	C?*A	| C?A	| C!V	| P&P	| nil
A	::=	X>P				
V	::=	[ ]	| [C]			
X	::=	[ ]	| [C]	| [_]		

$$C?*X>P \xrightarrow{in(C!V)} C?*X>P \;\&\; P\ \{V/X\}$$

```
in(Channel?*Pattern>AbsProcess, Channel!Message,
        (Channel?*Pattern>AbsProcess) & NewProcess)
                :-      AbsProcess @ {Message/Pattern} –-> NewProcess .

CE?*AProc @ {XL/NL} –-> C?*Proc       :-      CE @ {XL/NL} –-> C,
                                              AProc @ {XL/NL} –-> Proc.
```

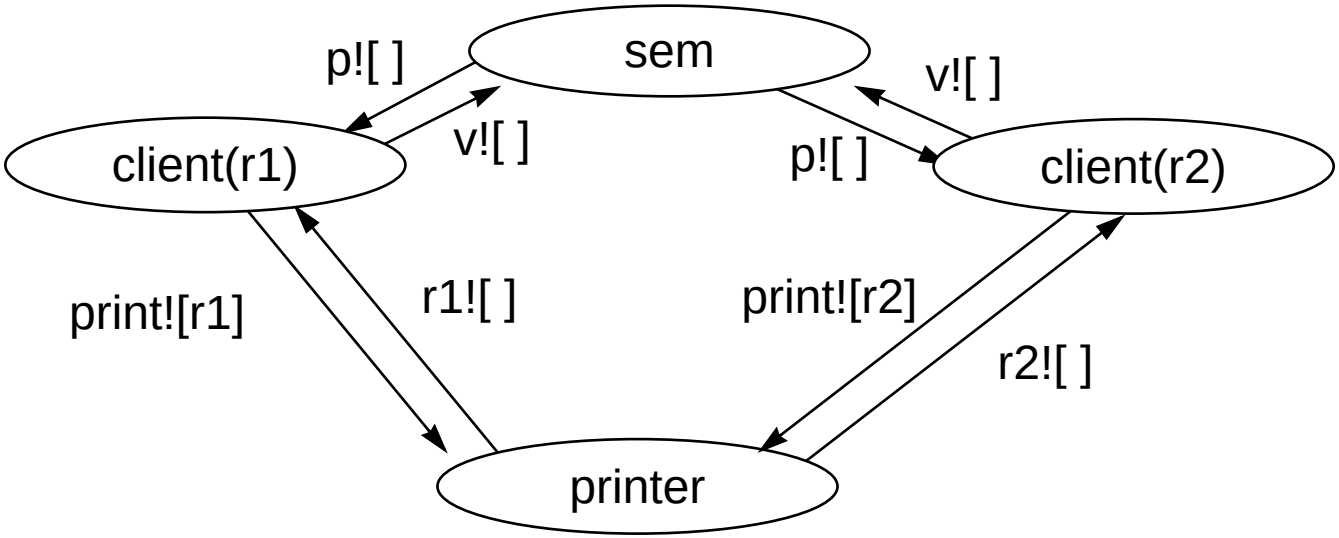
Resources as Replicated Processes

A counting semaphore: sem := p![ ] & v?*[ ]>p![ ].

A printer: printer := print?*[r]>r![ ].

A template for client processes: client(R) := p?[ ]>print![R] & R?[ ]>print![R] & R?[ ]>v![ ].

A configuration with two distinct clients: eg := sem & printer & client(r1) & client(r2).



Running the Example

sem & printer & client(r1) & *client(r2)*

p![ ] **=>** (v?*[ ]>p![ ]) & *(print?*[r]>r![ ])* & client(r1) & *print![r2]* & (r2?[ ]>print![r2] & r2?[ ]>v![ ])

print![r2] **=>** (v?*[ ]>p![ ]) & (print?*[r]>r![ ]) & *r2![ ]* & client(r1) & (*r2?[ ]>print![r2] & r2?[ ]>v![ ]*)

r2![ ] **=>** (v?*[ ]>p![ ]) & (*print?*[r]>r![ ]*) & client(r1) & *print![r2]* & r2?[ ]>v![ ]

print![r2] **=>** (v?*[ ]>p![ ]) & (print?*[r]>r![ ]) & *r2![ ]* & client(r1) & *r2?[ ]>v![ ]*

r2![ ] **=>** (*v?*[ ]>p![ ]*) & (print?*[r]>r![ ]) & client(r1) & *v![ ]*

v![ ] **=>** (v?*[ ]>p![ ]) & *p![ ]* & (print?*[r]>r![ ]) & *client(r1)*

p![ ] **=>** (v?*[ ]>p![ ]) & (*print?*[r]>r![ ]*) & *print![r1]* & (r1?[ ]>print![r1] & r1?[ ]>v![ ])
print![r1] **=>** (v?*[ ]>p![ ]) & (print?*[r]>r![ ]) & *r1![ ]* & (*r1?[ ]>print![r1] & r1?[ ]>v![ ]*)
r1![ ] **=>**(v?*[ ]>p![ ]) & (*print?*[r]>r![ ]*) & *print![r1]* & r1?[ ]>v![ ]
print![r1] **=>** (v?*[ ]>p![ ]) & (print?*[r]>r![ ]) & *r1![ ]* & *r1?[ ]>v![ ]*
r1![ ] **=>** (*v?*[ ]>p![ ]*) & (print?*[r]>r![ ]) & *v![ ]*
v![ ] **=>** (v?*[ ]>p![ ]) & p![ ] & (print?*[r]>r![ ])

Other Issues

- Choice:
  - How to express choice of inputs?
- Encapsulation:
  - How to encapsulate subsystems?
  - How to generate new channel names?
- Structural Equivalence:
  - Simplifying the transition semantics by giving structural equivalence rules — e.g., p&q == q&p
- Semantic Equivalence:
  - When do two expressions represent the same process?

PICT

Overview

- PICT core syntax
- Creating new channels
- Channel types
- Modelling language constructs
- A concurrent queue

Texts:

- R. Milner, “The Polyadic π-Calculus: A Tutorial,” U. Edinburgh, 1991
- B. Pierce, *Programming in the Pi-Calculus*, Tutorial Notes for PICT Version 3.6a, 1995

Abstract Syntax of (Untyped) Core PICT

Proc = *Val* **?** *Abs*
Val **?*** *Abs*
Val **!** *Val*
Proc **|** *Proc*
let new *Name* **in** *Proc* **end**

Abs = *Pat* **>** *Proc*

Pat = *Name*
[*Pat* , ... **]**
record *Id = Pat* , ... **end**
Name @ *Pat*
_

Val = *Name*
BasicVal
[*Val* , ... **]**
record end
Val **with** *Id* **=** *Val* **end**

Name = *Id*

BasicVal = *String*

Binding Channels

All channel names must be bound, either by “let new” or by an input pattern:

```
run
        let new x in
                x![ ]
        |       (x?[ ]>print!"Got it!")
        end
```

NB: print is a built-in channel

Typed Channels

Channels in PICT are typed, and may only carry values matching their type:

Type = **^** *Type*
! *Type*
? *Type*
[*Type* , ... **]**
Record end
Type **with** *Id* **:** *Type* **end**
Top

In most cases, types can be automatically inferred, and declarations are unnecessary:

```
run
    let new x : ^[ ] in
        x![ ]
    |   (x?[ ]>print!"Got it!")
    end
```

Synchrony and Asynchrony

Although PICT uses asynchronous message-passing, synchrony can be recovered by waiting for a response on a (fresh) channel:

```
def sem [p,v] >
        (p?r > r![ ])
  |     (v?*r > r![ ] | (p?r > r![ ]))
```

A definition is syntactic sugar for a (new) replicated process

```
let new sem
run (sem?*[p,v] >
        (p?r > r![ ])
  |     (v?*r > r![ ] | (p?r > r![ ])))
```

Note that all channel names are bound, and that channels can be passed as values.

Synchronizing Concurrent Clients

```
def client [p,v] >
        let new r, s1, s2 in
                p!r
        |       (r?[ ] > pr!["FIRST\n",s1])
        |       (s1?[ ] > pr!["SECOND\n",s2])
        |       (s2?[ ] > v!r | (r?[ ] > skip))
        end

run
        let new p, v in
                sem![p,v]
        |       client![p,v]
        |       client![p,v]
        |       client![p,v]
        end
```

Modelling Booleans

```
def tt [b] > b?*[t,_] > t![ ]
def ff [b] > b?*[_,f] > f![ ]

def test [b] >
        let new t, f in
                b![t,f]
        |       (t?[ ] > print!"True")
        |       (f?[ ] > print!"False")
        end

def notB [b,c] > c?*[t,f] > b![f,t]

run
        let new b, c in
                ff![b] | notB![b,c] | test![c]
        end
```

Modelling Language Constructs

Higher-level language constructs are modelled by translation to core PICT:

```
run
    let new x in
        x!false
    |   (x?b >
        if b
        then print!"True"
        else print!"False"
        end)
    end
```

is translated to:

```
run
    let new x in
        x!false
    |   (x?b >
        let new t,f in
            primif![b,t,f]
        |   (t?[ ] > print!"True")
        |   (f?[ ] > print!"False")
        end)
    end
```

Natural Numbers

A natural number n can be modelled by a channel *n* that reads a pair [p,z] of channels, and either sends z![ ] if it is equal to zero, or else sends p![k] where k represents n-1.

```
def zero [p,z] > z![ ]
def one [p,z] > p![zero]
def two [p,z] > p![one]
def three [p,z] > p![two]

def count [n] >
    let new p,z in
            n![p,z]
    |   (z?[ ] > print!"0")
    |   (p?[m] > print!"1+" | count![m])
    end

run count![three]
```

Counting

New numbers can be generated by constructing a *successor* process:

```
def succ [n, r] >
        let new s in
                r!s
        |       (s?*[p,z] > p![n])
        end


run
        let new r in
                succ![three,r]
        |       (r?s > count![s])
        end
```

Arithmetic

Arithmetic operators can be built up in the same way:

```
def add [m,n,r] >
        let new p, z in
                m![p,z]
        |       (z?[ ] > r!n)
        |       (p?[pm] >
                let new rn in
                        succ![n,rn]
                |       (rn?sn>add![pm,sn,r])
                end)
        end


run let new r in
                add![two,three,r]
        |       (r?s > count![s])
        end
```

Functional Notation

Infix notation and functional application are syntactic sugar for communication:

```
run printi!(2+5)
```

translates to:

```
run printi!((+)[2,5])
```

which translates to:

```
run
    let new r in
        (+)![2,5,r] | (r?value > printi!value)
    end
```

Functions as Processes

Functions can be defined as processes:

```
def double [n] = n+n
```

translates to:

```
def double [n,r] > r!(n+n)
```

which translates to:

```
def double [n,r] >
    let new r1 in
        (+)![n,n,r1]
    |   (r1?value > r!value)
    end

run printi!(double[5])
```

Functions as Processes

```
def fact [n] =
        if n == 0
        then 1
        else n * fact[n-1]
        end
```

translates to:

```
def fact [n,r] >
        let new br in
                (==)![n,0,br]
        |       (br?b >
                let new t, f in
                        primif![b,t,f]
                |       (t?[ ] > r!1)
                |       (f?[ ] >
                        let new nr in
                                (-)![n,1,nr]
                        |       (nr?k >
                                let new kfr in
                                        fact![k,kfr]
                                |       (kfr?kf >
                                        let new fr in
                                                (*)![n,kf,fr]
                                        |       (fr?f > r!f)
                                        end)
                                end)
                        end)
                end)
        end
```

```
run printi!(fact[5])
```

120

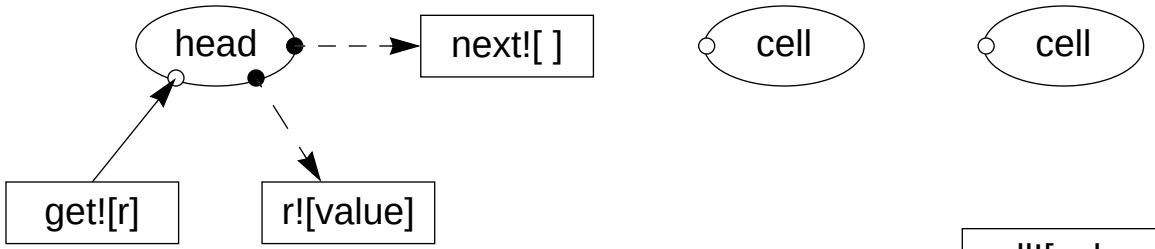
Sequencing

```
run
    pr["hello "];
    pr["world\n"];
    skip
```

translates to:

```
run
    let new r in
        pr!["hello ",r]
    |   (r?[ ] >
        let new r in
            pr!["world\n",r]
        |   (r?[ ] > skip)
        end)
    end
```

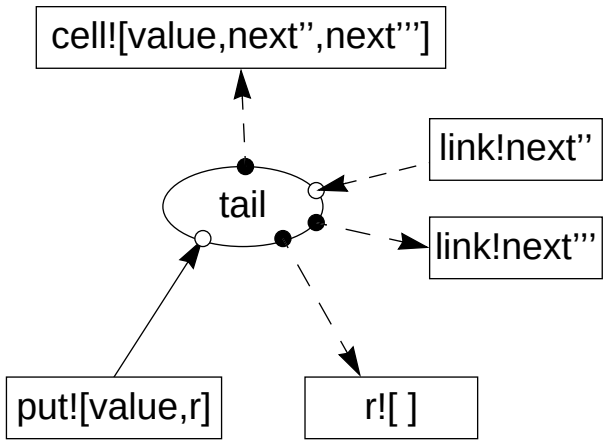
A Concurrent Queue



The head accepts a *get* request to yield its value and trigger the next cell.

A cell waits to be triggered by the head, and then itself becomes the head of the queue.

The tail services *put* requests by constructing a new cell that waits for the *next* trigger from the cell in front of it.



Implementing the Concurrent Queue

```
new get, put

def head[value, next] >
        get?[r] > r!value | next![ ]

def cell[value, ready, next] >
        ready?[ ] > head![value, next]

def tail [ ] >
        let new link, init in
                link!init
        |       (put?*[value,r] >
                link?ready >
                let new next in
                        cell![value,ready,next]
                |       link!next
                |       r![ ]
                end )
        |       init![ ]
        end

run
        let new r in
                tail![ ]
        |       (put["one"]; put["good"]; put["turn"]; put["deserves"]; put["another"]; skip)
        |       get![r]
        |       get![r]
        |       get![r]
        |       get![r]
        |       get![r]
        |       (r ?* s > print!s)
        end
```

Object-Based Concurrency

Overview

- What is an OBCL?
- Dimensions of OO Languages
- Expression of Concurrency
  - Objects and Processes
  - Granularity of Concurrency
  - Creating Processes
- Communication and Synchronization
  - Intra-Object and Inter-Object Synchronization
- Evaluating OBCLs
- Research Topics

What is an OBCL?

An Object-Based Concurrent Language supports:

- Encapsulation
  - ☞ objects encapsulate data and operations
- Concurrency
  - ☞ multiple processes may be concurrently active
  - ☞ need to: specify, create and synchronize processes

Why do we need OBCLs?

- Inherent application (real-world) concurrency
- Distributed applications
- Application integration and interoperability
- Parallel applications

Overview of OBCLs

- Traditional OBLs:
  - Smalltalk-80, C++, Objective C, Ada
  - libraries
- Extended OBLs:
  - CLU: Argus
  - Smalltalk-80: ConcurrentSmalltalk, Actalk, PO
  - C++: ACT++, Arjuna, Avalon, Karos
  - Eiffel//
- Concurrent OBLs:
  - Actors, ABCL, POOL, Guide, Hybrid, Meld

Requirements for OBCLs

- Object autonomy:
  - ☞ protection from concurrent requests
- Internal concurrency:
  - ☞ should be transparent to clients
- Local delay transparency:
  - ☞ handling of local delays should be transparent to the client
- Remote delay transparency:
  - ☞ handling of remote delays should be transparent to the service provider
- Composable synchronization policies:
  - ☞ subclasses should share synchronization code with superclasses

REF: *Papathomas, PhD thesis, 1992.*

Expression of Concurrency

- Objects and Processes:
  - ☞ How are processes and objects related?
- Granularity of Concurrency:
  - ☞ How many processes can be associated with an object?
- Process Creation:
  - ☞ How are processes created?

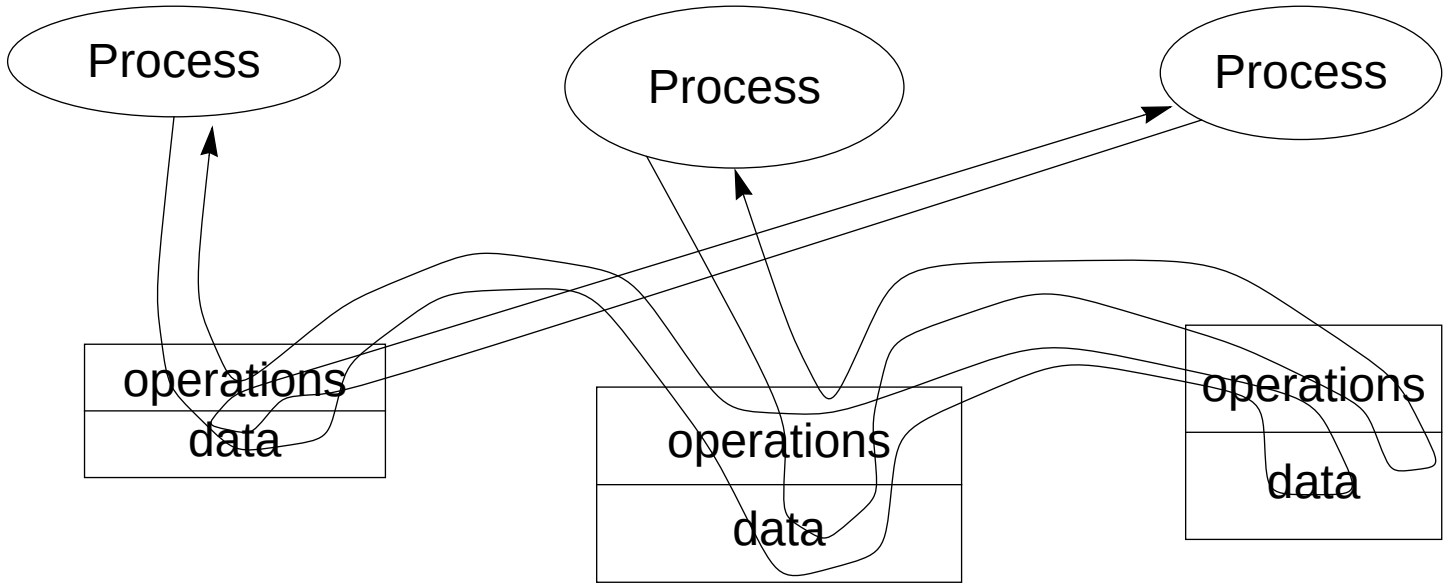
Objects and Processes

How are processes related to objects?

Three Classes of OBCL:

- *Passive Objects:* objects & concurrency independent (Smalltalk-80, C++, Objective-C, Emerald)
- *Active/Passive:* passive + “concurrent” objects (PAL)
- *Active Objects:* objects and processes are unified (ABCL/1, Hybrid, POOL ...)

Passive Object Models



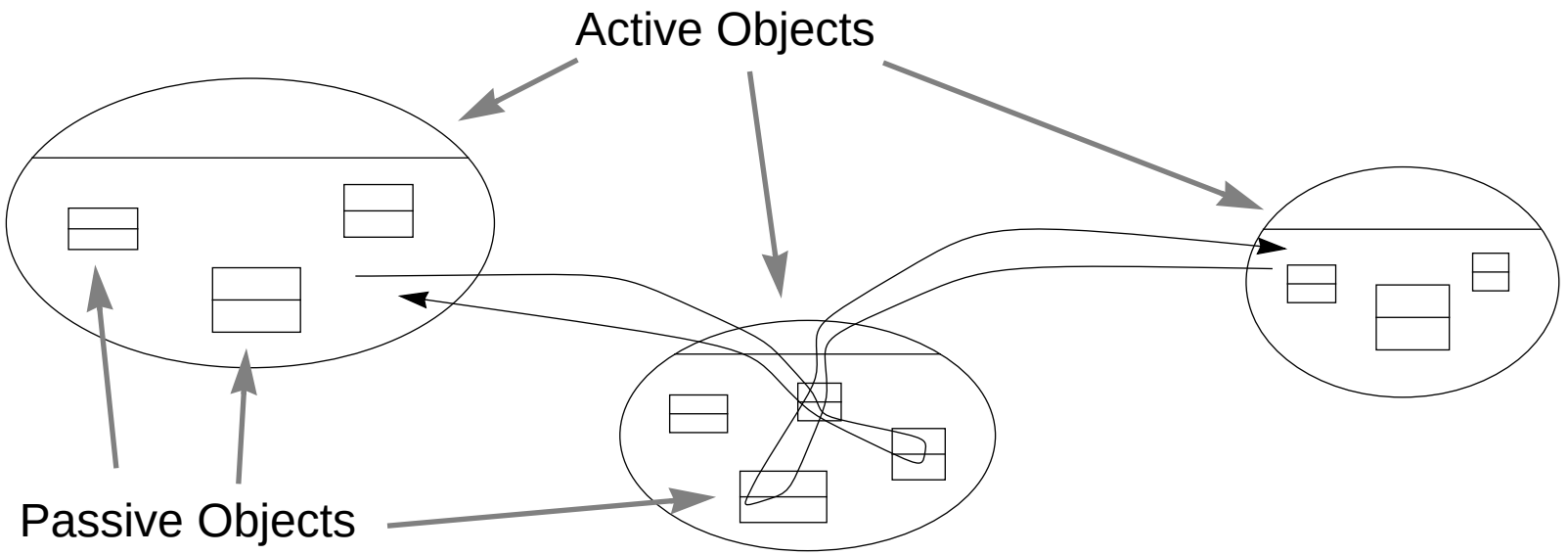
Concurrent processes access passive objects.

Processes synchronize according to a shared memory model:

- ☞ objects must be designed to be shared, or
- ☞ processes must explicitly synchronize via locks, etc.

Smalltalk-80, C++, Objective-C, Emerald

Active/Passive Models



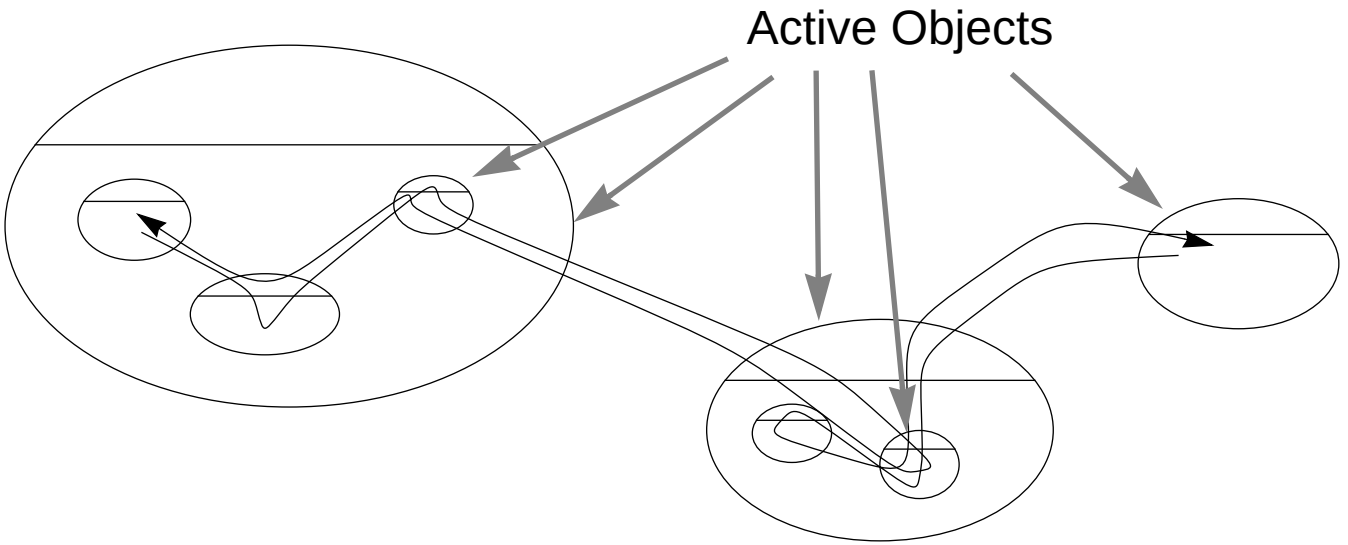
Active Objects are identified with processes

Passive objects are protected by the active objects containing them

- lightweight/heavyweight distinction
- two class hierarchies are incompatible

PAL

Active Object Models



Objects and processes are integrated:

- each operation invocation is a potentially concurrent thread
- an object with a running operation is *active*
- every object is autonomous and synchronizes its own threads

ABCL, Hybrid, POOL, ...

Granularity of Concurrency

Approaches to Concurrency:

Inter-Object Concurrency:

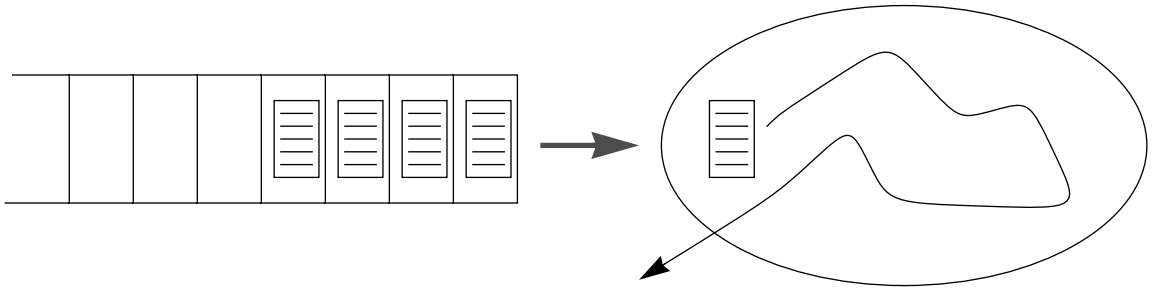
- Sequential Objects — Ada, POOL

Intra-Object Concurrency:

- Quasi-Concurrent Objects — Hybrid

- Concurrent Objects:
  - Client-Driven: Passive Objects
  - Server-Driven: Active Objects

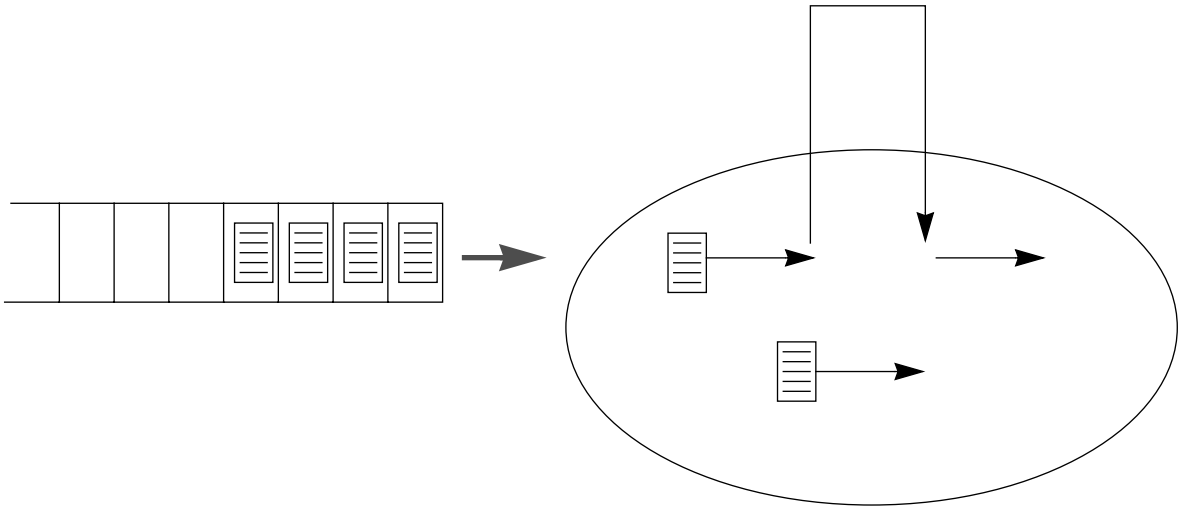
Sequential Objects



In a sequential object model, requests are serialised in a wait-queue

- each operation runs to completion before the next request is handled
- concurrency is introduced by having more objects

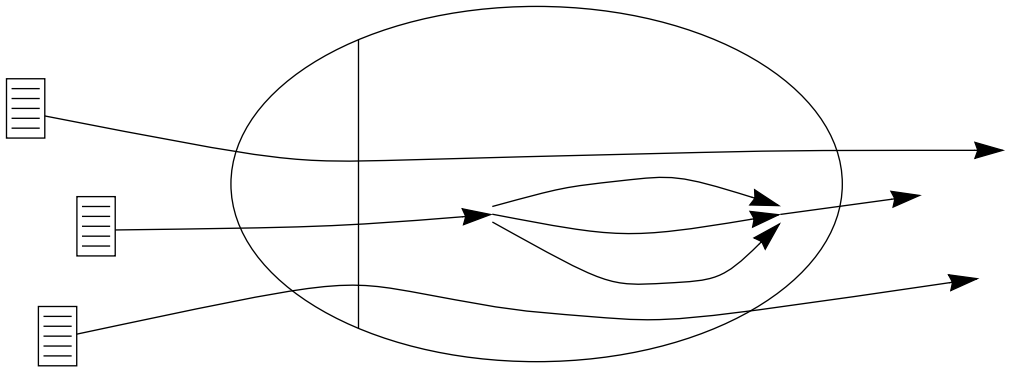
Quasi-Concurrent Objects



Quasi-concurrent objects may switch attention between multiple requests:

- In *Hybrid*, a *delegated call* to another object allows the serving object to switch to another request
- In *ABCL*, an *express message* may interrupt the thread servicing an ordinary invocation

Concurrent Objects



Concurrent Objects may serve multiple requests concurrently:

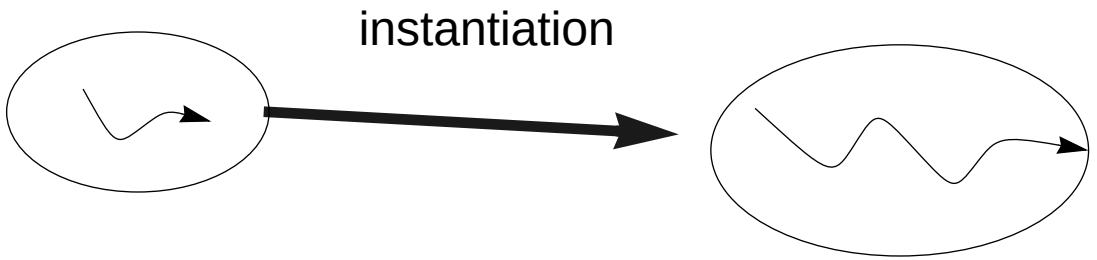
- Passive Objects require explicit synchronization of threads
- Active Objects control when to accept new requests
  - ☞ may create additional internal threads to service a single request

Passive: Smalltalk-80, C++, ...
Active: Sina, PO, Eiffel//, ...

Process Creation

- Asynchronous Objects
  - Explicit bodies
  - Implicit bodies
- Asynchronous Messages
  - one-way message-passing
  - futures

Asynchronous Objects

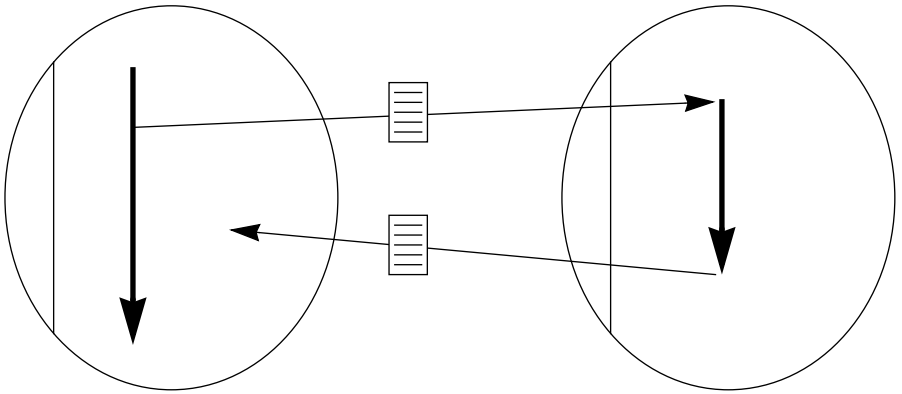


The “body” of an active object may be:

- Implicit and inaccessible — standard scheduler
- Explicit and customizable — initialization, scheduling, synchronization ...

Implicit: Actalk, Act++, Actors
Explicit: Ada, Eiffel//, Pool

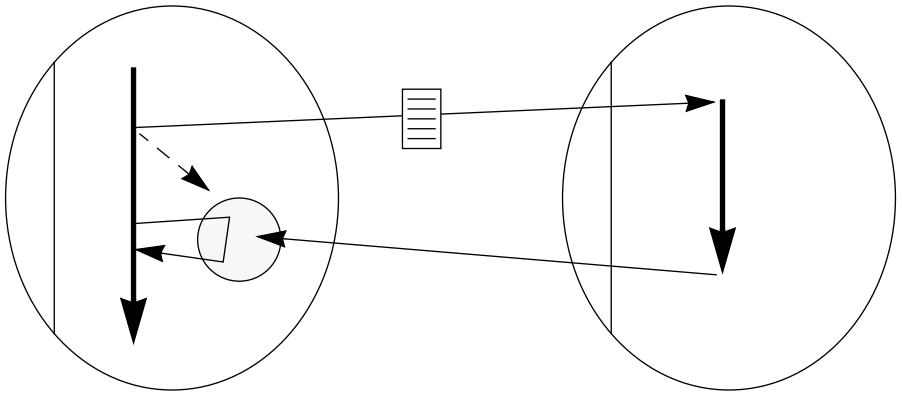
Asynchronous Invocation



Clients do not wait for the reply to continue executing

- one-way message-passing:
  - reply (if any) sent by *another* invocation
- futures:
  - reply sent to a *future object*

Futures



The reply to an asynchronous request is sent to a *future object*..

- The client obtains the result when needed.
- Clients block only if the result is not yet available when needed

Futures may be created either *explicitly* by clients or *implicitly* for all requests.

Explicit: ACT++, ABCL, PO, ConcurrentSmalltalk
Implicit: Eiffel//, Karos, Meld

Communication and Synchronization

- Intra-Object Synchronization:
  - Remote Delays: asynchronous invocations
  - Local Delays: condition synchronization
- Inter-Object Synchronization:
  - Transactions

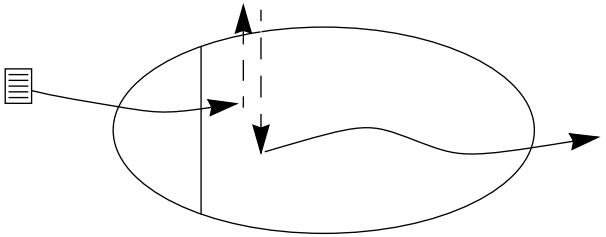
Local Delays

An object may need to delay selected requests to avoid local inconsistency.

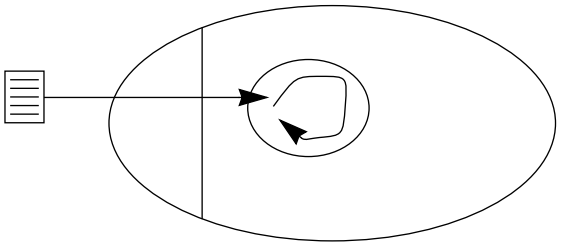
- Unconditional acceptance — Emerald, Trellis, Smalltalk-80
- Conditional acceptance
  - Centralized acceptance
    - Explicit acceptance — Ada, POOL, ABCL
    - Reflective computation — Actalk, ABCL/R
  - Distributed activation conditions
    - Representation specific — Guide, Hybrid, SINA
    - Abstract — Procol, ACT++, Rosette

Local Delays

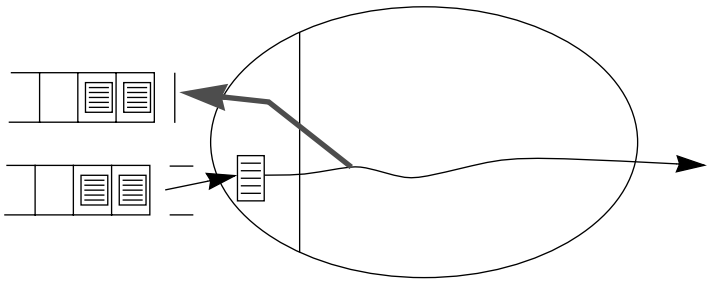
Unconditional acceptance



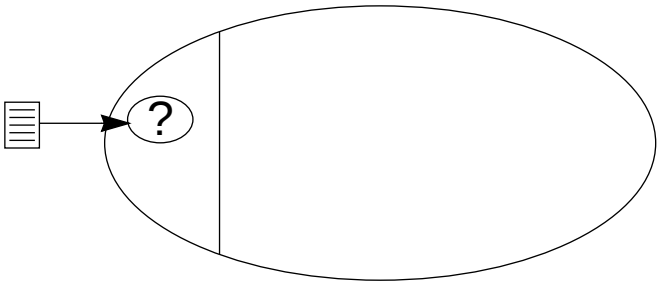
Explicit acceptance



Representation specific delays



Abstract synchronization conditions



Transactions

- Concurrency atomicity:
  - ☞ intermediate effects on shared objects are invisible to other transactions *(serialisability or isolation)*

- Failure atomicity:
  - ☞ transactions either complete successfully, or are aborted with no visible effect on shared objects *(the "all-or-nothing" property)*

Transactions may be associated with *transaction blocks* (explicit start and end), or may be realized as *atomic invocations* (implicit with operation start and end).

Classifying OBCLs

- Object Models
  - Active or Passive Objects?
- Granularity of Concurrency
  - Sequential, Quasi-Concurrent or Concurrent?
- Process Creation
  - Asynchronous Objects or Asynchronous Invocations?
- Local Delays
  - Conditional or Unconditional Acceptance?
  - Centralized or Distributed Activation Conditions?
  - Explicit or Reflective / Abstract or Representation-specific?

Evaluation

- Object autonomy:
  - ☞ active objects
- Internal concurrency:
  - ☞ server-driven
- Local delay transparency:
  - ☞ various approaches ...
- Remote delay transparency:
  - ☞ futures or internal threads
- Composable synchronization policies:
  - ☞ composable abstract synchronization policies ...

Text Processing Languages

Overview

- Text processing languages
- Sed and AWK
- Perl

Texts:

- L. Wall and R.L. Schwartz, *Programming Perl*, O'Reilly and Associates, 1992
- Bill Kinnersley, *The Language List* — Version 2.3, September 1994 (http://cuiwww.unige.ch/langlist, ftp://ftp.wustl.edu/doc/misc/lang-list.txt)

What are Text Processing Languages?

Common features:

- Strings as built-in data types
- Pattern matching
- Textual substitution
- Regular expressions
- Lists and associative arrays
- Automatic conversion between strings and numeric data types
- Formatting and report generation

Some Text Processing Languages

Selected from the Language List:

AWK — Aho Weinberger Kernighan. 1978. Text processing/macro language. “The AWK Programming Language” A. Aho, B. Kernighan, P. Weinberger, A-W 1988. (See Bawk, Gawk, Mawk, Nawk, Tawk.) ftp://netlib.att.com/research/awk*

AXLE — An early string processing language. Program consists of an assertion table which specifies patterns, and an imperative table which specifies replacements. “AXLE: An Axiomatic Language for String Transformations”, K. Cohen et al, CACM 8(11):657-661 (Nov 1965).

bawk — Bob Brodt. AWK-like pattern-matching language, distributed with Minix.

CONVERT —

1. String processing language, combined the pattern matching and transformation operations of COMIT with the recursive data structures of Lisp. “Convert”, A. Guzman et al, CACM 9(8):604-615 (Aug 1966).

EMACS LISP — Richard Stallman. Variant of LISP used by the EMACS editor. (This is the ”offical“ name, based on the EMACS FAQ file. See ELISP.)

Gawk — GNU’s implementation of a superset of POSIX awk, a pattern scanning and data manipulation language. ftp://prep.ai.mit.edu/pub/gnu/gawk-2.15.4.tar.Z //archive.umich.edu/mac/utilities/developerhelps/macgawk2.11.cpt.hqx

LOGOL — Strings are stored on cyclic lists or ’tapes’, which are operated upon by finite automata. J. Mysior et al, “LOGOL, A String manipulation Language”, in Symbol Manipulations Languages and Techniques, D.G. Bobrow ed, N-H 1968, pp.166-177.

mawk — Mike Brennan <brennan@bcsaic.boeing.com> 1991. An implementation of nawk, distributed under GNU license but distinct from GNU’s gawk. ftp://oxy.edu/public/mawk

Nawk — New AWK. AT&T. Pattern scanning and processing language. An enhanced version of AWK, with dynamic regular expressions, additional built-ins and operators, and user-defined functions.

PANON — A family of pattern-directed string processing languages based on generalized Markov algorithms. “String Processing Languages and Generalized Markov Algorithms”, A. C. Forino, Proc IFIP Working Conf on Symb Manip Languages, pp.141-206, Amsterdam 1968. PANON-1, based on Simple GMA’s and PANON-2 based on Conditional Functional GMA’s.

Perl — Practical Extraction and Report Language. Larry Wall <lwall@netlabs.com> An AWK-like interpreted language for scanning text and printing formatted reports. Regular expression primitives, dynamically- scoped variables and functions, extensible runtime libraries, exception handling, packages. Version 5 adds nested data structures and object- oriented features. “Programming Perl”, Larry Wall et al, O’Reilly & Assocs. ftp://ftp.netlabs.com/pub/outgoing/perl.4.0 for Unix, MS-DOS, Amiga //ftp.netlabs.com/pub/outgoing/perl5.0/perl5a1.tar.Z for Sparc //rascal.utexas.edu/programming/Perl_402_MPW_CPT_bin for Mac
uucp: osu-cis

Sed — Stream editor. The input language used by the Unix stream editor.

SNOBOL — StriNg Oriented symBOlic Language. David Farber, Ralph Griswold & I. Polonsky, Bell Labs 1962-3. String processing language for text and formula manipulation. ”SNOBOL, A String Manipulating Language“, R. Griswold et al, J ACM 11(1):21 (Jan 1964).

SPRING — String PRocessING language. ”From SPRING to SUMMER: Design, Definition and Implementation of Programming Languages for String Manipulation and Pattern Matching“, Paul Klint, Math Centre, Amsterdam 1982.

TAWK — Tiny AWK.

Tcl —

1. (“tickle”) Tool Command Language. John Ousterhout, UCB. <ouster@sprite.berkeley.edu> A string language for issuing commands to interactive programs. Each application can extend tcl with its own set of commands. “Tcl: An Embeddable Command Language”, J. Ousterhout, Proc 1990 Winter USENIX Conf. ftp:// ucbvax.berkeley.edu

TECO — Text Editor and COrrector. (Originally “Tape Editor and COrrector”). Macro language for text editing, screen handling and keyboard management. Has a reputation for being cryptic and hard to learn. (TECO programs are said to resemble line noise.) The first EMACS editor was written in TECO. ftp://usc.edu, for VAX/VMS, Unix, MS-DOS, Mac, Amiga

VULCAN —

3. Early string manipulation language. ”VULCAN - A String Handling Language with Dynamic Storage Control“, E.P. Storm et al, Proc FJCC 37, AFIPS (Fall 1970).

ZUG — Geac. [?] A low-level Awk?

Regular Expressions (Perl)

Each character matches itself, unless it is one of the special characters: +?.*()[]{}|\

- . matches an arbitrary character *except* a newline
- (...) groups a series of pattern elements to a single element
- + matches the preceding pattern element one or more times
- ? matches zero or one times
- * matches zero or more times
- {N,M} matches from N to M times; {N} exactly N; {N,} N or more
- [...] denotes a class of characters to match; [^...] negates the class
- (...|...|...) matches one of the alternatives
- \w matches alphanumerics and "_"; \W matches non-alphanumerics
- \b matches word boundaries; \B negation
- \s matches whitespace; \S matches non-whitespace
- \d matches digits; \D matches non-digits
- \n \r \t match newlines, carriage returns, tabs
- \1...\9 refer to matched sub-expressions grouped with (...)
- $& string matched by the last pattern match

SED

SED performs substitutions on a stream of text:

```
#! /bin/sed -f
# escape special characters for Framemaker
s/\\/&&/g
s/    /\\t/g
s/[<>]/\\&/g
s/–/<endash>/g
s/“/<quotedblleft>/g
s/’\’\”/<quoteblright>/g
```

Alternatively, at command level:

```
sed -e ’s/\\/&&/g’                  \
    -e ’s//\\t/g’                   \
    -e ’s/[<>]/\\&/g’               \
    -e ’s/–/<endash>/g’             \
    -e ’s/“/<quotedblleft>/g’       \
    -e ’s/’\’\”/<quoteblright>/g’   \
    $*
```

AWK

AWK modifies text streams by transformation rules:

```
#! /bin/awk -f
#
# pgs      --- count pages in %P fields of refer files

/%P [0-9]*$/ { pgs += $2 ; next }

/%P [0-9]* *- *[0-9]*$/ {
        p = substr($0,4)
        n = split(p, pp, "-")
        pgs += 1 + pp[2] - pp[1]
}

END { print pgs }
```

Perl

“Practical Extraction and Report Language”

or

“Pathologically Eclectic Rubbish Lister”

Principle features:

- uniform selected merge of: *sed, awk, csh, c ...*
- numbers, text, binary data
- file, string processing, regular expressions
- built-in lists, associative arrays
- special variables to control processing ($/, $[...)
- common systems calls (files, directories, sockets, ...)
- compilation; dynamic evaluation; error-handling
- packages

Regular Expressions

Sed-like behaviour can be obtained with the -p flag:

```
#! /usr/local/bin/perl -p
#
# caps    --- change initial letters of words to upper case
#
# But don't capitalize isolated letters!

s/\w/\l$&/g;              # convert all alphabetics to lower case
s/\b\w\w/\u$&/g ;         # change initial characters of words to upper case
```

is equivalent to:

```
#! /usr/local/bin/perl
while (<>) {              # read a line of input into $_
    s/\w/\l$&/g;          # perform a substitution on $_
    s/\b\w\w/\u$&/g ;
    print;                # print $_
}
```

Sed and AWK scripts can be automatically translated to Perl.

Arrays

```
#! /local/bin/perl -s
#
# rsort    –- sort a file of records

$/ = "";          # blank line separates records

print sort(@input=<>);
```

- Special variables control default behaviour
- Values are interpreted as scalars, arrays or associative arrays depending on the current context
- Built-in functions efficiently implement common text processing operations

Subroutines

```
#! /usr/local/bin/perl -s
# rsort    --- sort a file of records

$usg = "Usage : rsort [-r(everse)] [-u(nique)] [<file> ...]\n";
die $usg if $h;

$/ = "";          # blank line separates records
if ($r) {
      if ($u) { &uniq(sort({$b cmp $a} @input=<>)); }
      else { print sort({$b cmp $a} @input=<>); }
} else {
      if ($u) { &uniq(sort(@input=<>)); }
      else { print sort(@input=<>); }
}

sub uniq {
      foreach $current (@_) {
            next if ($current eq $previous);
            print $previous = $current;
      }
}
```

File I/O

```
#! /local/bin/perl -s
# rsplit–- split a file of records into two parts by a keyword


$usg = "Usage: rsplit <key> <file>\n";
# blank line is record separator
$/ = "";
$key = $ARGV[0];
if ($#ARGV == 1) { $IN = $ARGV[1]; $MATCH = "$IN.1"; $REST =
"$IN.2"; }
else { die $usg; }
open(IN,$IN);
open(MATCH,">$MATCH");
open(REST,">$REST");
while (<IN>) {
      /$key/o && do { print MATCH $_ ; next; };
      print REST $_;
}
```

Dynamic Compilation

```
#! /usr/local/bin/perl -s
#
# rgrep   --- extract records matching a pattern from files

$u = "Usage: rgrep [-i] <pattern> [<file> ...]\n" ;

($pattern,@files) = @ARGV ;
defined($pattern) || die($u) ;
@ARGV = @files ;

$/ = "" ;          # set blank line to be record separator

if ($i) { $i = "i"; }

# patterns with alternatives are slow to evaluate,
# so construct a logical alternative instead:
foreach $p (split(/\|/,$pattern)) {
        $mpat .= "/$p/o$i && (print, next);\n";
}
eval "while(<>) { $mpat }";
```

Packages

```
#! /local/bin/perl
#
# pre      –- produced pre-formatted HTML text
unshift(@INC,"/user/oscar/Cmd/PerlLib");
require("url.pl");
if ($#ARGV >= $[) {
        foreach $file (@ARGV)
                { open(FILE,$file); &pre($file,FILE); close(FILE); }
}
else { &pre("stdin", stdin); }
sub pre {
        local($file,$input) = @_;
        print "<TITLE>Ascii file: $file</TITLE>\n<PRE>\n";
        while(<$input>) {
                study;
                s/&/&amp;/g; s/</&lt;/g; s/>/&gt;/g;
                &url'href;      # recognize hypertext links and make them live
                print;
        }
        print "</PRE>\n";
}
```

Standard System Calls

```
sub http {
    local($host,$port,$request) = @_;
    ($fqdn, $aliases, $type, $len, $thataddr) = gethostbyname($host);
    $that = pack($sockaddr, &AF_INET, $port, $thataddr);
    socket(FS, &AF_INET, &SOCK_STREAM, $proto) || return undef;
    bind(FS, $thissock) || return undef;
    local($/);
    unless (eval q!
        $SIG{'ALRM'} = "url'timeout";
        alarm(30);
        connect(FS, $that) || return undef;
        select(FS); $| = 1; select(STDOUT);
        print FS "GET $request\r\n";
        $page = <FS>;
        $SIG{'ALRM'} = "IGNORE";
        !) { return undef; }
    close(FS);
    $page;
}
```

Perl: Pros and Cons

Pros:

- Highly optimized for text processing
- Convenient for writing Unix administration scripts
- Acceptable support for writing modules
- On-the-fly compilation (+ error detection)

Cons:

- Weak encapsulation (global variables)
- No facility for defining complex data types
- Easy to introduce type errors

Scripting Languages

Overview

- Shell Languages, Command Languages, Scripting Languages, Fourth Generation Language and Coordination Languages
- The Bourne Shell

Texts:

- Bill Kinnersley, *The Language List* — Version 2.3, September 1994 (http://cuiwww.unige.ch/langlist, ftp://ftp.wustl.edu/doc/misc/lang-list.txt)
- S.R. Bourne, “An Introduction to the UNIX Shell,” UNIX User’s Manual, 1978

Scripting Languages and Their Kin

The distinctions between the following languages classes are fuzzy at best.

Shell Language:

- ☞ language for interacting with an application or operating system

Command Language:

- ☞ interactive language for issuing commands to a system

Scripting Language:

- ☞ language for controlling and composing components of a system

Fourth Generation Language:

- ☞ high-level language for specialized (usually database) applications

Coordination Language:

- ☞ language for coordinating multi-agent systems

Shell Languages

AppleScript — An object-oriented shell language for the Macintosh, approximately a superset of HyperTalk.

bash — Bourne Again SHell. GNU's command shell for Unix. ftp://prep.ai.mit.edu/pub/gnu/bash-1.10.tar.Z

csh — C-Shell. William Joy. Command shell interpreter and script language for Unix.

es —

1. Extensible Shell. Unix shell derived from rc, includes real functions, closures, exceptions, and the ability to redefine most internal shell operations. "Es - A Shell with Higher Order Functions", P. Haahr et al, Proc Winter 1993 Usenix Technical Conference. ftp://ftp.sys.utoronto.ca/pub/es/es-0.84.tar.Z

FOCL — Expert system shell, a backward chaining rule interpreter for Mac. ftp://ics.uci.edu/pub/machine-learning-programs/KR-FOCL-ES.cpt.hqx
info: pazzani@ics.uci.edu

GEST — Generic Expert System Tool. Expert system shell with frames, forward and backward chaining, fuzzy logic. Version 4.0. For Symbolics LISP machines only. ftp://ftp.gatech.edu/pub/ai/gest.tar.Z
info: John Gilmore <John.Gilmore@gtri.gatech.edu>

ksh — Korn Shell command interpreter for Unix.

MIKE — Micro Interpreter for Knowledge Engineering. Expert system shell for teaching purposes, with forward and backward chaining and user- definable conflict resolution strategies. In Edinburgh Prolog. BYTE Oct 1990. Version 2.03 ftp://hcrl.open.ac.uk/pub/software/src/MIKE-v2.03
info: Marc Eisenstadt <M.Eisenstadt@hcrl.open.ac.uk>

rc — Tom Duff. AT&T Plan 9 shell. Lookalike by Byron Rakitzis <byron@archone.tamu.edu> ftp://archone.tamu.edu

sh — (or "Shellish"). S.R. Bourne. Command shell interpreter and script language for Unix. "Unix Time-Sharing System: The Unix Shell", S.R. Bourne, Bell Sys Tech J 57(6):1971-1990 (Jul 1978).

TACL — Tandem Advanced Command Language. Tandem, about 1987. The shell language used in Tandem computers.

Command Languages

GCL — General Control Language. A portable job control language. ”A General Control Interface for Satellite Systems“, R.J. Dakin in Command Languages, C. Unger ed, N-H 1973.

IBEX — Command language for Honeywell’s CP-6 OS.

LE/1 — Langage External. ”An Evaluation of the LE/1 Network Command Language Designed for the SOC Network“, J. du Masle, in Command Languages, C. Unger ed, N-H 1973.

PCL —

3. Peripheral Conversion Language. Honeywell. Command language for file transfer between I/O devices on the CP-V and CP-6 operating systems.

POCAL — PETRA Operator’s CommAnd Language.

RCL — Reduced Control Language. A simplified job control language for OS360, translated to IBM JCL. “Reduced Control Language for Non- Professional Users”, K. Appel in Command Languages, C. Unger ed, N-H 1973.

RECOL — REtrieval COmmand Language. CACM 6(3):117-122 (Mar 1963).

SCL —

1. System Control Language. Command language for the VME/B operating system on the ICL2900. Block structured, strings, superstrings (lists of strings), int, bool, array types. Can trigger a block whenever a condition on a variable value occurs. Macros supported. Commands are treated like procedure calls. Default arguments. ”VME/B SCL Syntax“, Intl Computers Ltd 1980.

TACL — Tandem Advanced Command Language. Tandem, about 1987. The shell language used in Tandem computers.

Command Languages ...

Tcl —

1. (“tickle”) Tool Command Language. John Ousterhout, UCB. <ouster@sprite.berkeley.edu> A string language for issuing commands to interactive programs. Each application can extend tcl with its own set of commands. “Tcl: An Embeddable Command Language”, J. Ousterhout, Proc 1990 Winter USENIX Conf. ftp:// ucbvax.berkeley.edu

Tcl —

2. Terminal Control Language. The command language used in the Pick OS. “Exploring the Pick Operating System”, J.E. Sisk et al, Hayden 1986.

tcsh — Command language for Unix, a dialect of csh.

UNIQUE — A portable job control language, used. ”The UNIQUE Command Language - Portable Job Control“, I.A. Newman, Proc DATAFAIR 73, 1973, pp.353-357.

WFL — Work Flow Language. Burroughs, ca 1973. A job control language for the B6700/B7700 under MCP. WFL was a compiled block-structured language similar to ALGOL-60, with subroutines and nested begin-end’s. ”Work Flow Management User’s Guide“, Burroughs Manual 5000714 (1973). ”Burroughs B6700/B7700 Work Flow Language“, R.M. Cowan in Command Languages, C. Unger ed, N-H 1975.

Scripting Languages

AppleScript — An object-oriented shell language for the Macintosh, approximately a superset of HyperTalk.

Cmm — C Minus Minus. Scripting language. ftp://ftp.std.com/vendors/CEnvi-Cmm/share

csh — C-Shell. William Joy. Command shell interpreter and script language for Unix.

DCL —
1. DIGITAL Command Language. The interactive command and scripting language for VAX/VMS.

ECSS II — Extendable Computer System Simulator. An extension of SIMSCRIPT II. ”The ECSS II Language for Simulating Computer Systems“, D.W. Kosy, R- 1895-GSA, Rand Corp.

expect — A script language for dealing with interactive programs. Written in Tcl. ”expect: Scripts for Controlling Interactive Tasks“, Don Libes, Comp Sys 4(2), U Cal Press Journals, Nov 1991. ftp://ftp.uu.net/languages/tcl/expect/*

Hyperscript — Informix. The object-based programming language for Wingz, used for creating charts, graphs, graphics, and customized data entry.

HyperTalk — Bill Atkinson and Dan Winkler. A verbose semicompiled language with loose syntax and high readability. Relies on HyperCard as an object management system, development environment, and interface builder. Programs are organized into ”stacks“ of ”cards“, each of which may have ”buttons“ and ”fields“. All data storage is in zero-terminated strings in fields, local, or global variables; all data references are through ”chunk expressions“ of the form last item of background field ”Name List“ of card ID 34217’. Flow of control is event-driven and message-passgin among scripts that are attached to stack, background, card, field and button objects. ”Apple Macintosh HyperCard User Guide“, Apple Computer 1987. ”HyperTalk Language Reference Manual“, A-W 1988. Available from Claris Corp.

Lakota — Scripting language, extends existing OS commands.
info: Richard Harter <rh@smds.UUCP> SMDS Inc.

Lingo — An animation scripting language. MacroMind Director V3.0 Interactivity Manual, MacroMind 1991.

Scripting Languages ...

Obliq — Luca Cardelli, 1993. A distributed object-oriented scripting language. Small, statically scoped, untyped, higher order, and concurrent. State is local to an address space, while computation can migrate over the network. The distributed computation mechanism is based on Modula-3 network objects. ftp://gatekeeper.dec.com/pub/DEC/Modula-3/contrib

PSML — Processor System Modeling Language. Simulating computer systems design. A preprocessor to SIMSCRIPT. “Processor System Modeling - A Language and Simulation System”, F. Pfisterer, Proc Symp on Simulation of Computer Systems (Aug 1976).

QUIKSCRIPT — Simulation language derived from SIMSCRIPT, based on 20-GATE. “Quikscript - A Simpscript-like Language for the G-20”, F.M. Tonge et al, CACM 8(6):350-354 (June 1965).

REXX — Restructured EXtended eXecutor. M. Cowlishaw, IBM ca. 1979. (Original name: REX. They also call it “System Product Interpreter”). Scripting language for IBM VM and MVS systems, replacing EXEC2. “The REXX Language: A Practical Approach to Programming”, M.F. Cowlishaw, 1985. Versions: PC-Rexx for MS-DOS, and AREXX for Amiga.
list: REXX-L@UIUCVMD.BITNET. ftp://rexx.uwaterloo.ca/pub/freerexx/* REXX interpreters for Unix

sh — (or ”Shellish“). S.R. Bourne. Command shell interpreter and script language for Unix. ”Unix Time-Sharing System: The Unix Shell“, S.R. Bourne, Bell Sys Tech J 57(6):1971-1990 (Jul 1978).

SIMSCRIPT — Harry Markowitz et al, Rand Corp 1963. Implemented as a Fortran preprocessor on IBM 7090. Large discrete simulations, influenced Simula. ”SIMSCRIPT: A Simulation Programming Language“, P.J. Kiviat et al, CACI 1973. Versions: SIMSCRIPT I.5 (CACI 1965 - produced assembly language), SIMSCRIPT II, SIMSCRIPT II.5. CACI, (619)457-9681.

TUTOR — Scripting language on PLATO systems from CDC. “The TUTOR Language”, Bruce Sherwood, Control Data, 1977.

Fourth Generation Languages (4GLs)

Clarion — MS-DOS 4GL.

D —

1. ”The Data Language.“ MS-DOS 4GL.

Linc — Burroughs/Unisys 4GL. Designed in New Zealand.

NATURAL — Software AG, Germany. Integrated 4GL used by the database system ADABAS. Menu-driven version: SUPER/NATURAL. Also NATURAL 2?

R:BASE — MS-DOS 4GL from Microrim. Based on Minicomputer DBMS RIM. Was Wayne Erickson the author?

Coordination Languages

Linda — Yale. A ”coordination language“, providing a model for concurrency with communication via a shared tuple space. Usually implemented as a subroutine library for a specific base language. ”Generative Communication in Linda“, D. Gelernter <gelernter@cs.yale.edu> ACM TOPLAS 7(1):80-112 (1985). ”Linda in Context“, N. Carreiro <carreiro@cs.yale.edu> et al, CACM 32(4):444-458 (Apr 1989). (See C-Linda, Ease, Fortran-Linda, LindaLISP, Lucinda, Melinda, Prolog-Linda).

MeldC — Columbia U, 1990. A C-based concurrent object-oriented coordination language built on a reflective architecture. A redesign of MELD. Version 2.0 for Sun4’s and DECstations.
info: Gail Kaiser <meldc@cs.columbia.edu>

Also sometimes classified as coordination languages:

GAMMA —
2. A high-level parallel language. Research Directions in High-Level Parallel Languages, LeMetayer ed, Springer 1992.

LO — Linear Objects. Concurrent logic programming language based on “linear logic”, an extension of Horn logic with a new kind of OR- concurrency. “LO and Behold! Concurrent Structured Processes”, J. Andreoli et al, SIGPLAN Notices 25(10):44-56 (OOPSLA/ECOOP ’90) (Oct 1990).

The Bourne Shell

- Executing programs as commands
- Background commands
- Input and output redirection
- Pipes and filters
- File “globs”
- Shell scripts (parameterized)
- Control flow
- Shell variables (with parameter and command substitution)
- Associated commands (test, echo ...)
- Built-in commands (read, wait, trap, exec)
- Signal handling

Pipes and Filters

```
#! /bin/sh
#
# words  --- produce a sorted list of words in a file
#

cat $*                          |       \
tr -c A-Za-z0-9 '\012'          |       \
sed '/^$$/d'                    |       \
sort -u -f
```

Example

```
#! /bin/sh
#
# glue     --- glue two files side-by-side

a=a$$
b=b$$

sed 's/^/^A/' $1 | cat -n > $a
sed 's/^/^A/' $2 | cat -n > $b

clean='BEGIN { FS = "^A" }
{ printf "%s%s\n", $2, $3 }'

join -a1 -a2 -t^A $a $b | awk "$clean"
rm $a $b
```

Argument processing

```
#! /bin/sh
#
# nsort   --- sort lines by name (final word)

for arg
do
      case $arg in
      -* )  flags="$flags $arg" ;;
      * )   files="$files $arg" ;;
      esac
done

sed 's/.* \([^ ]*\)$/\1?&/' $files          |      \
sort $flags                                 |      \
sed 's/.*?//'
```

Command Substitution

```
#! /bin/sh
# rdiff         --- merge of two files with diffs marked by > or <
# deleted fields are prefixed with "<" and new fields with a ">"
plus='> '
min='< '
u='Usage: rdiff [+=<string>] [-=<string>] <old> <new>'
for arg
do
        case $arg in
        +=* )  plus=`echo "$arg" | sed 's/^+=//'` ;;
        -=* )  min=`echo "$arg" | sed 's/^-=//'` ;;
        -* )   echo "$u" 1>&2 ; exit ;;
        * )    files="$files $arg" ;;
        esac
done
diff -D diff $files | awk '
        /^#ifdef/      { prefix = plus ; next }
        /^#ifndef/     { prefix = min ; next }
        /^#else/       {       if (prefix == min)
                                        prefix = plus
                               else   prefix = min
                               next
                       }
        /^#endif/      { prefix = "" ; next }
        { printf "%s%s\n", prefix, $0 }'              plus="${plus}" min="${min}"
```

Exec

```
#! /bin/sh
#
# src          --- locate source of files and invoke lynx
# Includes $PATH in the list of directories to search.
# Also looks in $BIN, $MAN and $SRC environment variables.
bin="$BIN"
man="$MAN"
src="$SRC /local/src /local/pck /local/gnu"

case $# in
0 )     echo "Usage : src <cmd> ..." 1>&2 ; exit ;;
esac
echo -n "Searching ... "
path=`echo $PATH | sed 's/:/ /g'`
files=`( whereis $* ; \
        whereis -B $path $bin -M $man -S $src -f $* ) | \
        awk 'BEGIN { FS = ":" } { print $2 }' | \
        tr ' ' '\012' | \
        sort -u`

case $files in
"" )    echo "nothing found" ;;
* )     exec lynx $files ;;
esac
```

The Future of Scripting Languages

- Multimedia scripting
- Configuring open applications
- Composing objects, applications
- Coordinating distributed services