Structuring the Computation

Expressions

infix notation | **prefix notation** | **postfix notation**

infix notation	prefix notation	postfix notation
$a * (b + c)$	$* \, a + b \, c$	$a \, b \, c + *$

operator associativity and operator preceedence

$a + b * c$	corresponds to	$a + (b * c)$	(Pascal, C, ...)
$a = b < c$	corresponds to	$(a = b) < c$	(Pascal)
$a == b < c$	corresponds to	$a == (b < c)$	(C)

conditional expressions

$(a > b) \, ? \, a : b$	(C)
if $a > b$ then a else b	(ML)
case x of 1 => f1(y) | 2 => f2(y) | _ => g(y)	(ML)

Conditional Statements

if x > 0 **then** **if** x < 10 **then** x := 0 **else** x := 1000
if x > 0 **then** **begin** **if** x < 10 **then** x := 0 **end** **else** x := 1000
if x > 0 **then** **if** x < 10 **then** x := 0 **end** **else** x := 1000 **end**
if a **then** S1 **else** **if** b **then** S2 **else** **if** c **then** S3 **else** S4 **end**

```
switch (operator) {
   case '+': result = operand1 + operand2; break;
   case '-': result = operand1 - operand2; break;
   default:  break; }
```

case OPERATOR **is**
  when '+' => result := operand1 + operand2;
  when '−' => result := operand1 − operand2;
  when **others** => **null**;
end **case**

Loops

for var := lower **to** upper **do** statement (Pascal)
for (int i = 0; i < 10; i++) {...} (C++)
for var **in** discrete_range **loop** body **end loop** (Ada)

while condition **do** statement (Pascal)
while (expression) statement; (C)
while condition **loop** loop_body **end loop** (Ada)

repeat statement **until** condition (Pascal)
do statement **while** (expression); (C)
loop statement; **exit when** condition **end loop** (Ada)

A: **loop** ... **loop** ... **exit** A; ... **end loop** ... **end loop** A;

Routines

procedure p(**var** x: T; y: Q; **function** f(z: R): integer);

```
void proc(int* x, int y)
   { *x = *x + y; }
void proc(int& x, int y)
   { x = x + y; }
```

alias problem: aliases reduce readability and prevent optimizations

$$u := x + z + f(x,y) + f(x,y) + x + z$$

but, aliases are also a source of expressiveness and flexibility

Exception Handling in Ada

predefined exceptions: Constraint_Error
Program_Error
Storage_Error
Tasking_Error

user defined exception (example): Help: **exception**;

explicit raise of an exception (example): **raise** Help;

```
begin -- block with exception handling
   ... statements ...
   exception when Help => ... statements ...
             when Constraint_Error => ... statements ...
             when others => ... statements ...
end;
```

Implementation of Exception Handling

compiler assigns unique name to exception

if an exception occurs, search handler in outer blocks within routine,
if not found, propagate exception to calling routine (along the dynamic link)

each activation record holds pointer to a static table,
each table entry associates an exception with a handler

alternative implementation: global exception table (no cost without exception),
binary search on ip addresses to find handler (costs when exception occurs)

exceptions can be propagated to the outside of the scope

Exception Handling in C++

arbitrary data can be propagated as exceptions

throwing an exception:

> **`throw`** Help(MSG1);

declaration of propagated exceptions in functions (depricated):

> void foo() **`throw`**(Help, Zerodivide);

special functions:

unexpected()	when propagating wrong exception, depricated
terminate()	when no exception handler found

Example in C++

```
class Help { ... };
class Zerodivide { ... };

    ...
    try { ... }
    catch(Help& msg) {
        switch(msg.kind) {
            case MSG1:
                ...;
            case MSG2:
                ...;
            ...
        }
    }
    catch(Zerodivide& z) {
        ...
    }
```

Exception Handling in Java

propagated exceptions must be declared (except `RuntimeException`, `Error`):

> void foo() **throws** Help;

usual try-catch-finally block:

```
try { ... }
catch(Exception1 ex) { ... }
catch(Exception2 ex) { ... }
...
finally { ... } // exception in finally supresses that in try
```

try with resources (since Java 7):

```
try (Reader r = new FileReader(path)) { ... }
// r closed; exception in try suppresses that from closing
```

Exception Handling in ML

```
exception Neg

fun fact(n) =
  if n < 0 then raise Neg
  else if n = 0 then 1
    else n * fact(n - 1)

fun fact_0(n) =
  fact(n) handle Neg => 0;
```

Exception Handling in Eiffel

```
try_several_methods is
    local
        i: INTEGER    -- automatically initialized to 0
    do
        try_method(i);
    rescue
        i := i + 1;
        if i < max_trials then
            retry     -- retry the execution of try_several_methods
        end
    end
```

Pattern Matching

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

fun day_off(Sun) = true
  | day_off(Sat) = true
  | day_off(_) = false

fun reverse(nil) = nil
  | reverse(head::tail) = reverse(tail) @ [head]

fun rev(nil) = nil
  | rev(0::tail) = [0] @ [tail]
  | rev(head::tail) = rev(tail) @ [head]
```

Nondeterminism and Backtracking

A if B or
C or
D.

C if E and
F and
G.

D if I or
H.

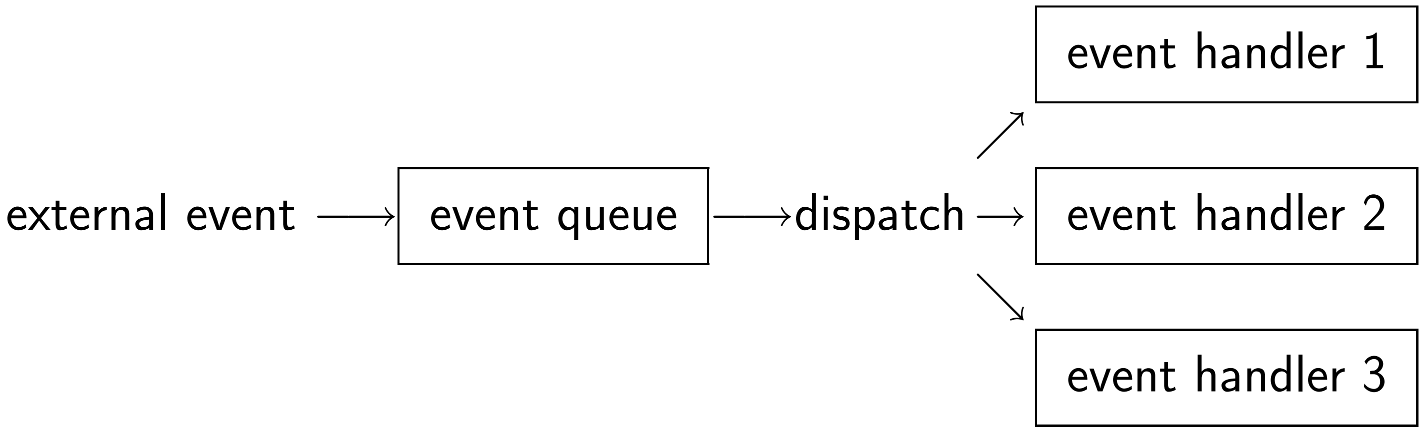
Event Driven Computation

example of trigger in database:

on event
when condition
  do action

on insert in EMPLOYEE
when TRUE
  do emp_number++

Scheme of Event Driven Computation



Scheme of Actors

message to 1 $\longrightarrow$ $\boxed{\text{event queue}}$ $\longrightarrow$ $\boxed{\text{actor 1}}$

message to 2 $\longrightarrow$ $\boxed{\text{event queue}}$ $\longrightarrow$ $\boxed{\text{actor 3}}$

message to 3 $\longrightarrow$ $\boxed{\text{event queue}}$ $\longrightarrow$ $\boxed{\text{actor 3}}$

Coroutines

```
                     int i = 0;

unit client {          | unit main {        | unit next {
  int stop = ...;      |   resume client;   |   int step(){...};
  ...                  | }                  |   ...
  while(i != stop) {   |                    |   while(true) {
    ...                |                    |     i += step();
    resume next;       |                    |     resume client;
  }                    |                    |   }
}                      |                    | }
```

Process Example: Producer and Consumer

```
process producer {
  while (true) {
    produce an element;
    append it to the buffer;
  }
}
```

```
process consumer {
  while (true) {
    take element from buffer;
    operate on it;
  }
}
```

Buffer Operations

```
void append(int x) {
  count++;
  int i = next_in();
  buffer[i] = x;
}
```

```
int remove() {
  count--;
  int j = next_out();
  return buffer[j];
}
```

race conditions occur when executed concurrently, synchronization necessary

Declaration of Processes in Ada

```
task type SERVER is
  entry NEXT_REQUEST(NR: in REQUEST);
  entry SHUT_DOWN;
end SERVER;

type SERVER_PTR is access SERVER;

MY_SERVER: SERVER;

task CHECKER is
  entry CHECK(T: in TEXT);
  entry CLOSE;
end CHECKER;

HIS_SERVER_PTR: SERVER_PTR := new SERVER;
```

Semaphores

low-level synchronization mechanism based on a counters

two atomic actions on each semaphore:

P(s): if $s > 0$ then $s = s - 1$
else suspend current process

V(s): if there is a process suspended on the semaphore
then wake up process
else $s = s + 1$

before entering a critical section controlled by s invoke P(s),
on exit invoke V(s)

Using Semaphores

```
var buffer buf;
semaphore mutex = 1;  in = 0;  spaces = buf.size();
```

```
process producer {
  int i;
  while(true) {
    produce(i);
    P(spaces);
    P(mutex);
    buf.append(i);
    V(mutex);
    V(in);
  }
}
```

```
process consumer {
  int j;
  while(true) {
    P(in);
    P(mutex);
    j = buf.remove();
    V(mutex);
    V(spaces);
    consume(i);
  }
}
```

Using a Monitor in Concurrent Pascal

```
type fifostorage = monitor
  var    contents: array[1..n] of integer;
         total: 0..n;  in, out: 1..n;
         sender, receiver: queue;
  procedure entry append (item: integer)
  begin  if total = n then delay(sender);
         contents[in] := item;
         in := (in mod n) + 1;  total := total + 1;
         continue(receiver)
  end;
  procedure entry remove (var item: integer)
  begin  if total = 0 then delay(receiver);
         item := contents[out];
         out := (out mod n) + 1;  total := total - 1;
         continue(sender)
  end;
begin  total := 0;  in := 1;  out := 1  end
```

Synchronization of Processes via Monitor

```
type fifostorage = ... previous slide ...;

type producer = process (storage: fifostorage)
  var   element: integer;
  begin cycle ...; storage.append(element); ... end end;

type consumer = process (storage: fifostorage)
  var   datum: integer;
  begin cycle ...; storage.remove(datum); ... end end;

var   meproducer: producer;
      youconsumer: consumer;
      buffer: fifostorage;

begin meproducer(buffer);
      youconsumer(buffer);
end
```

Using a Monitor in Java

```
public class IntBuffer100 {
    private int[] cont = new int[100];
    private int in = 0, out = 0, total = 0;

    public synchronized void append(int item) {
        while (total >= 100) try { wait(); }
                              catch(InterruptedException e){}
        cont[in] = item;  total++;  if (++in >= 100) in = 0;
        notifyAll();
    }
    public synchronized int remove() {
        while (total <= 0) try { wait(); }
                            catch(InterruptedException e){}
        int temp = cont[out];  total--;  if (++out >= 100) out = 0;
        notifyAll();
        return temp;
    }
}
```

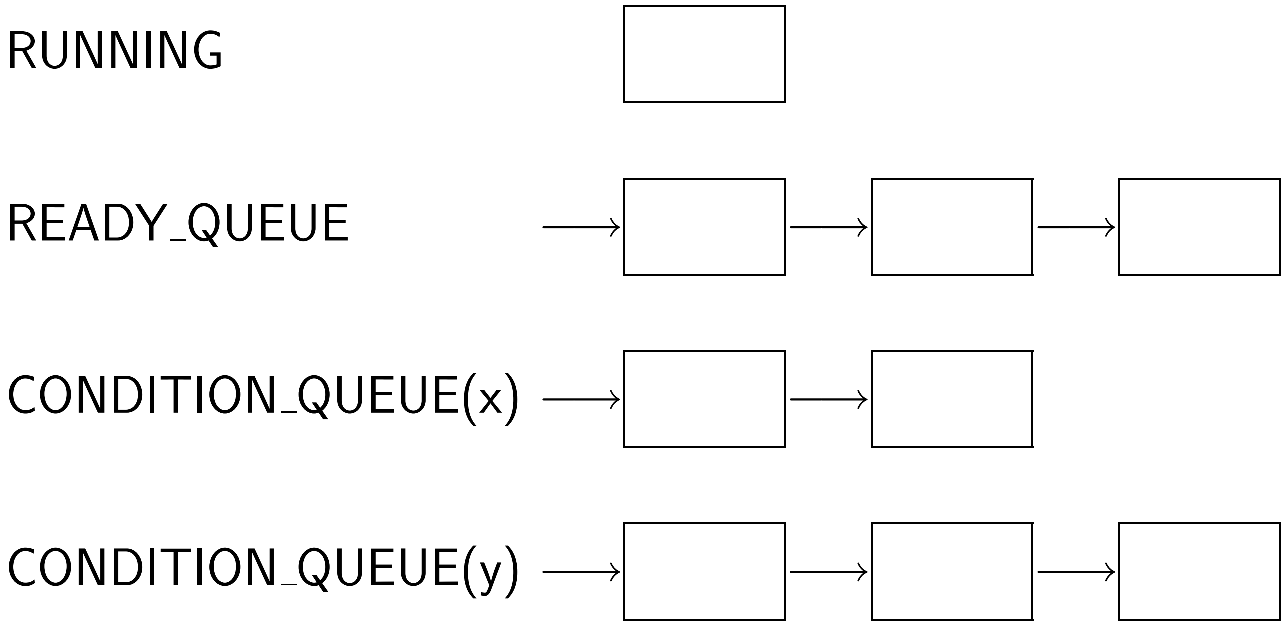
Using a Protected Type (Monitor) in Ada

```
protected type Fifo_Storage is
    entry Append (Item: Integer);
    entry Remove (Item: out Integer);
private  Contents: array(1..100) of Integer;
         In, Out: Integer range 1..100 := 1;
         Total: Integer range 0..100 := 0;
end Fifo_Storage;
protected body Fifo_Storage is
    entry Append (Item: Integer) when Total < 100 is begin
        Contents(In) := Item;
        In := (In mod 100) + 1;  Total := Total + 1;
    end Append;
    entry Remove (Item: out Integer) when Total > 0 is begin
        Item := Contents(Out);
        Out := (Out mod 100) + 1;  Total := Total - 1;
    end Remove;
end Fifo_Storage;
```

Rendezvous in Ada

```
task Buffer_Handler is
    entry Append (I: Integer);
    entry Remove (I: out Integer);
end Buffer_Handler;
task body Buffer_Handler is
    Cont: array(1..100) of Integer;
    In, Out: Integer range 1..100 := 1;
    Total: Integer range 0..100 := 0;
begin loop select when Total < 100 =>
               accept Append(I: Integer) do Cont(In) := I end;
               In := (In mod 100) + 1;  Total := Total + 1;
           or when Total > 0 =>
               accept Remove(I: out Integer) do I := Cont(Out) end;
               Out := (Out mod 100) + 1;  Total := Total - 1;
         end select;
      end loop;
end Buffer_Handler;
```

State Management by Operating System



Operations of Operating System

dispatch module in operating system is ADT supporting these operations:

enqueue: Queue $\times$ Descriptor $\longrightarrow$ Queue
dequeue: Queue $\longrightarrow$ Queue $\times$ Descriptor
empty: Queue $\longrightarrow$ Boolean

each clock interrupt executes this operation:

```
Suspend_and_Select() {
    RUNNING = process_status;
    READY_QUEUE.enqueue(RUNNING);
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
}
```

Implementation of Semaphore

```
Suspend_on_Condition(c) {
    RUNNING = process_status;
    CONDITION_QUEUE(c).enqueue(RUNNING);
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
}

Awaken(c) {
    RUNNING = process_status;
    READY_QUEUE.enqueue(RUNNING);
    READY_QUEUE.enqueue(CONDITION_QUEUE(c).dequeue());
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
}
```

Implementation of Monitor

"mutual exclusion" by disabling interrupts while being in the monitor

```
Continue(c) {
    RUNNING = process_status (interrupts enabled,
        program counter set to return point from monitor call);
    READY_QUEUE.enqueue(RUNNING);
    if not CONDITION_QUEUE(c).empty() {
        READY_QUEUE.enqueue(CONDITION_QUEUE(c).dequeue());
    }
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
}
```

Implementation of Rendezvous (1)

each "entry" has descriptor with these fields:

O: Boolean; true if entry is open
W: waiting queue (task descriptors of callers)
T: descriptor of task owning entry
I: pointer to first instruction of accept body

```
Call_Entry(e) {
    RUNNING = process_status;
    DESCR(e).W.enqueue(RUNNING);
    if DESCR(e).O {
        for all entries oe of DESCR(e).T do { oe.O = false; }
        RUNNING = DESCR(e).T;
        RUNNING.ip = DESCR(e).I;
    } else { RUNNING = READY_QUEUE.dequeue(); }
    process_status = RUNNING;
}
```

Implementation of Rendezvous (2)

```
At_End_Of_Accept_Body(e) {
    RUNNING = process_status;
    READY_QUEUE.enqueue(DESCR(e).W.dequeue());
    READY_QUEUE.enqueue(RUNNING);
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
}

Execute_Accept_Statement(e) {
    if DESCR(e).W.empty() {
        DESCR(e).O = true;
        DESCR(e).T = process_status;
        RUNNING = READY_QUEUE.dequeue();
        process_status = RUNNING;
    } -- else simply continue executing the accept body
}
```

Implementation of Rendezvous (3)

```
Execute_Select_Statement() {
    LOE = list of open entries in selection;
    if LOE is empty { raise Program_Error; }
    else {
        if DESCR(e).W.empty() (for all e in LOE) {
            for all e in LOE do {
                DESCR(e).O = true;
                DESCR(e).T = process_status;
            }
            RUNNING = READY_QUEUE.dequeue();
            process_status = RUNNING;
        } else { choose an e with not DESCR(e).W.empty(); }
    }
    proceed execution from instruction DESCR(e).I;
}
```