# Statement-Level Control

- **Three types:**
  - → **sequencing (not much to be said about this)**
  - → **selection**
  - → **repetition**
- **Structured Programming requires the use of only three structues:**
  - → **sequence, selction, repetition**
  - → **See Bohm, Jacopini, CACM, 1965.**
  - → **structure can also be shown in Nassi-Scheiderman diagrams**
  - → **See Nassi, I., Shneiderman, B., Flowchart techniques for structured programming, *SIGPLAN Notices*, ACM August 1973.**

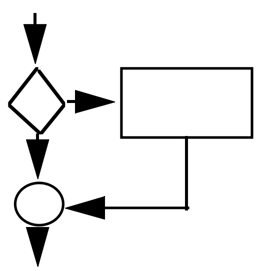
# PROPER PROGRAMS

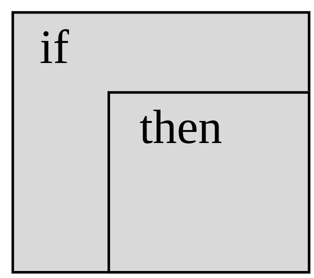
- **A proper program has:**
  - → **one in point and one out point**
  - → **every statement/instruction lies on a path between the in point and the out point**
- **Proper programs are constructed from one in-one out substructures**
  - → **sequence**
  - → **selection**
  - → **repetition**

# Selection

- **Issues:**
  - **How is selection controlled?**
  - **How many choices?  Can none of the choices be selected**

- **Single-way selectors**



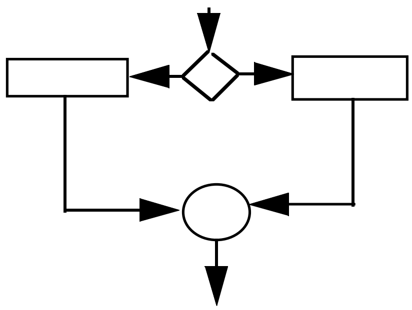


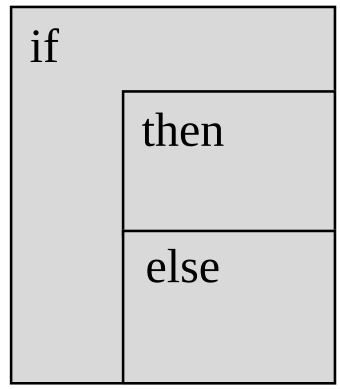

if <bool> then <stmt>

  - **Without compound statements, need GOTO**

# Two-Way Selection

- **Two-way selectors**






if <bool> then <stmt> else <stmt>

```
procedure Check_card_validity ( Bank_identifier: BANK ;
      Bank_list: SET_OF_BANK ;  Number: ACCOUNT_NUMBER ;
            Last_transaction_date, Todays_date: DATE ;
            Last_transaction: POSITIVE ; Card_status: in out STATUS ) is
            -- Function Is_a_member_of checks if the given Bank identifier is
            -- held in the list of participating banks
            -- Valid_format  checks the account number to ensure that the
            -- initial check digit is correct
begin
   if Expiry_date >= Todays_date and
            Last_transaction_date <= Todays_date and
            Valid_format (Account_number) and
            Is_a_Member_of (Bank_list, Bank_identifier) then
                      Card_status := OK;
else
                      Card_status := Invalid;
end if ;
end Check_card_validity ;
```

# Dangling Elses

- **Recall potential ambiguity with nested 2-way selectors:**

if b1 then if b2 then s1 else s2

→ **Several possible remedies. . .**

# Avoiding Ambiguity

- **Disallow nested conditionals**

  <stmt> --> <a_stmt> | <if_stmt> | <c_stmt>

  <c_stmt> --> begin <stmt_list> end

  <if_stmt> --> if <bool> then < s_stmt>

  | if <bool> then <s_stmt> else <stmt>

  <s_stmt> --> <a_stmt> | <c_stmt>

  if b1 then

  begin if b2 then s1 else s2 end

  if b1 then

  begin if b2 then s1 end

  else s2

# Multiple-Way Selection

- **Multiple-way Selection**
  - → **New issues:**
    - How to handle unrepresented selector values?
    - Should selectable segments be followed by implicit branches out of construct?
  - → **Lots of forms of multiple selectors. . .**

# Multiple-Way Selection -- Early Approaches

- **Three-way selectors (Fortran)**

```
IF (<arith_exp>) L1, L2, L3
```

  → **Must jump out of segment:**

```
      IF (...) 10, 20, 30
10    ...
      GO TO 40
20    ...
      GO TO 40
30    ...
40    ...
```

- **Computed GO TO (Fortran)**

```
GO TO (L1, L2, ..., Ln), exp
```

  → **goes to first label if exp=1, second if exp=2, etc.**

  → **if exp < 1 or exp > n, no effect**

  → **must still jump out of segment**

# Multiple-Way Selection -- Modern Approaches

- **Pascal example**

```
case <exp> of
    <const_list> : <stmt>
    ...
    <const_list> : <stmt>
end
```

  - → **<exp> of ordinal type**
  - → **implicit branch to end after each segment**
  - → **"otherwise" option**
  - → **selector value unrepresented => runtime error**

# More Modern Approaches

- **C example**

```
switch (<exp>)
{
case <const_exp> : <stmt>
 ...
case <const_exp> : <stmt>
[default : <stmt>]
}
```

  - **<exp> and <const_exp> yield integers**
  - **Cases are not mutually exclusive**
  - **No implicit branches -- use "break" to leave switch**

    Is this a good idea?

- **Elsif (Ada)**
  - **Like LISP cond**
  - **Cuts down on indenting**
  - **More flexible than case statement**

# Ada elsif

```
procedure Get_command is
begin
	-- track the cursor until it is over a menu selection or in the workspace
	loop
		Cursor_position := Get_cursor_position ;
		exit when positioned in workspace or
					(positioned over menu and button clicked) ;
		Display_cursor_position ;
	end loop ;
	if In_workspace (Cursor_position) then
		Command := Edit_workspace ;
	elsif In_command_menu  (Cursor_position) then
		Display_command_menu ;
		Command := Get_command_from_menu ;
	elsif In_Known_indexes  (Cursor_position) then
		Command := Display_indexes ;
	elsif In_Current_indexes  (Cursor_position) then
		Command := Display_current_indexes ;
	elsif In_Clear_button then
		Command := Clear_workspace ;
	elsif In_Document_name  (Cursor_position) then
		Command := Edit_document_name ;
	elsif In_Qualifier  (Cursor_position) then
			Command := Edit_qualifier ;
	elsif In_Documents  (Cursor_position) then
			Command := Display_documents ;
	elsif In_Quit_button  (Cursor_position) then
		Command := Quit ;
	else
		System_error ;
	end if ;
end Get_command ;
```

# Repetition

- **Iteration (statement-level)**
- **Recursion (unit-level) -- later**
- **Issues:**
  - → **How is iteration controlled?**
  - → **Where in loop is control mechanism?**
    - top
    - bottom
    - anywhere

# Counter-Controlled Loops

- **Issues:**
  - → **Type and scope of loop variable**
  - → **Value of loop variable at loop term**
  - → **Change loop parameterss in body?  Effect?**
  - → **Branch into loop?**
  - → **Test at top or bottom?**
  - → **Loop parameters evaluated when?**

# FORTRAN loops

- **DO <label> <var> = <init>,<term>**
  - → **Can't change loop variable/parameters inside loop**
  - → **Loop variable undefined after normal termination; last value if jump out**
  - → **Can jump out of, back into loop**

```
DO 100 I = 1,10
...
100  CONTINUE
```

  - → **Note!**

```
DO 100 I = 1.10
```

# More Loop Examples

- **ALGOL 60**
  - → **Very complicated! see book.**

- **ALGOL 68**

```
for i from j by k to m
while b do . . . od
```

- **C**

```
for (<exp>;<exp>;<exp>) <stmt>
       init   term   incr
```

```
for (i = 0; i <= 10; i++)
    sum = sum + a[i];
```

  - → **No special loop variables.**
  - → **Can have multiple statements for each <exp>**

# Ada Loop

```
procedure Request_PIN (PIN: in out PIN_STRING )
        -- the PIN is represented as a string of 4 characters
    D: CHARACTER ;
begin
    User_request ("Please type your personal number") ;
    for i in 1.. 4  loop
    Get_character _from_keypad (D) ;
        Put_onto_screen  ('*') ;
        PIN (i) := D ;
    end loop ;
end Request_PIN ;
```

# Logically Controlled Loops

- **Issues:**
  - → **Pretest or posttest?**
  - → **Jump into loop?**

- **Pascal constructs**

  while <exp> do <stmt>

  repeat <stmt> until <exp>

# Ada While Loop

```
procedure Binary_search (Key: ELEM ; T: ELEM_ARRAY ;
            Found: in out BOOLEAN ; L: in out ELEM_INDEX ) is
    -- Assume that T'FIRST and T'LAST are both
    -- greater than or equal to zero and T'LAST >= T'FIRST
    Bott : ELEM_INDEX := T'FIRST ;
    Top : ELEM_INDEX := T'LAST ;
    Mid : ELEM_INDEX;
begin
    L :=  (T'FIRST + T'LAST ) / 2;
    Found := T( L ) = Key;
    while Bott <= Top and not Found loop
        Mid := (Top + Bott) / 2;
        if T( Mid ) = Key  then
            Found := true;
            L := Mid;
        elsif T( Mid ) < Key then
            Bott :=  Mid + 1;
        else
            Top :=  Mid - 1;
        end if;
    end loop;
end Binary_search;
```

# Loop Exits

- **Other exits, e.g. Modula 2, Ada:**

```
loop
...
if <exp> then EXIT
...
end
```

- **CONTINUE, BREAK**

# More Statement-Level Control Constructs

- **Explicit loop exits**
  - → **Basic model: (Modula)**

```
loop
. . .
if . . . then EXIT
. . .
end
```

- **Multi-level exits (Ada)**
  - → **named loops**
  - → **conditional EXIT**

exit [<loop name>] [when <condition>]

# Ada Example

```
Outer_loop:
    for i in 1. .10 loop
    Inner_loop:
      for j in 1. .20 loop;
        ...
        EXIT Outer_loop when <cond>;
        ...
    end loop Inner_loop;
      ...
end loop Outer_loop;
```

# Ada — ATM Design

```
loop
      loop
            Print_input_message (" Welcome - Please enter your card") ;
            exit when Card_input  ;
      end loop ;
      Account_number := Read_card ;
      Get_account_details (PIN, Account_balance, Cash_available) ;
      if Validate_card (PIN) then
            loop
                  Print_operation_select_message ;
                  case Get_button is
                        when Cash_only  =>
                              Dispense_cash (Cash_available, Amount_dispensed) ;
                        when Print_balance  =>
                              Print_customer_balance (Account_balance) ;
                        when Statement =>
                              Order_statement (Account_number) ;
                        when Check_book    =>
                              Order_check_book (Account_number) ;
                  end case ;
                  Eject_card ;
                  Print ("Please take your card or press CONTINUE") ;
                  exit when Card_removed ;
             end loop ;
            Update_account_information (Account_number,  Amount_dispensed) ;
      else
            Retain_card ;
      end if ;
end loop ;
```

# Ada — Summary

```
-- An Ada if statement
if A = B and X = Y then
        Some_action ;
else
        Some_other_action ;
end if ;

-- An Ada for loop
for J  in (1..10) loop
        Do_something (J) ;
end loop ;

-- An Ada while loop
while J < 20 loop
        Do_something (J) ;
        J := J + Some_function (J) ;
end loop ;

-- An Ada case statement
case Sensor is
        when Red => Do_danger ;
        when Amber => Do_warning ;
        when Green => Do_safe ;
end case ;
```

# Iterators (Clu)

- **Lets user specify range of values over which a loop iterates.**

```
for atom:node in list(x) do
    ...

list = iter(z: linked_list) yields (node)
    ...
    yield (n)
    ...
end list
```

→ **list produces elements of type node one at a time**

# Guarded Commands

```
if   <bool> -> <stmt>
□    <bool> -> <stmt>
     ...
□    <bool> -> <stmt>
fi
```

**Each** <bool> **is a *guard.***

- **Semantics**
  - **Evaluate all <bool>s**
  - **If $\geq$ 1 is true, choose one nondeterministically and execute its <stmt>**
  - **If none true, <u>error</u>**

# Guarded Commands (continued)

```
do   <bool> -> <stmt>
□  <bool> -> <stmt>
     ...
□  <bool> -> <stmt>
od
```

**We avoid overspecifying order of actions.**

- **Semantics**
  - → **Evaluate all <bool>s**
  - → **If $\geq$ 1 is true, choose one nondeterministically and execute its <stmt>**
  - → **Repeat until none true, then normal termination**