Section 6

Commands

6. Commands
   6.1 Expressions vs. commands
   6.2 Recursive definitions
   6.3 Atomic commands
   6.4 Block commands
   6.5 Conditional commands
   6.6 Iterative commands
   6.7 Structured programming
   6.8 Sequencers
   6.9 Exceptions
Commands: a visual mindmap

- Commands vs. Expressions
  - Expression Oriented Languages
- Command Expressions
- Variations
  - Command Expressions
  - Expressions
- Commands vs. Expressions
- Basics
- Skip
- Assignement
- Varieties
  - Vanilla
  - Update
  - Multiple
  - Simultaneous
  - Collateral
- Examples
  - Pascal
  - C
  - Java
- Atomic Commands
- Command Constructors
  - Block
  - Conditional
  - Iterative
- Structured vs. Non-Structured Programs
- Jump
- Escape
- Varieties
  - Definite/Indefinite
  - Sequential/Simultaneous/Collateral
- Nassi-Shneiderman Diagrams

Varieties
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
- Sequential/Simultaneous/Collateral

Examples
- Pascal
- C
- Java

Varieties
- Vanilla
- Update
- Multiple
- Simultaneous
- Collateral

Commands vs. Expressions
- Basics
- Skip
- Assignement

Varieties
- Sequential
- Simultaneous
- Collateral

Varieties
- Definite/Indefinite
6. Commands

6.1. Expressions vs. commands
Commands: what are they?

Commands are characteristic of imperative languages (No commands in purely functional languages)

Definition (Command)

A command is a part of a computer program, which:
- does not produce a value,
- whose main purpose is altering the program’s state.
- even vacuously

Examples
- I/O: print, read,…
- Assignment
- Loops
- Conditional
- “nop”
- “\relax”
- “;”
The misnomer statement is (much) more frequently used in the literature.

But, there is nothing declarative in commands!

“Statement” also means:
- definitions
- declarations
- anything ending with a “;”
Commands vs. expressions

Ideally, they should be distinct

Commands
- Change state
- No value

Expression
- No state change
- Produce a value

In practice, the borderline is not so clear
Expressions changing the program’s state?

Nasty CS101 Exam Question
You are given a seemingly innocent Pascal code, and asked...

Procedure Hamlet;
VAR
  happy: Boolean;
Function toBe:Boolean;
Begin
...
  happy := not happy;
  toBe := happy
End;
Begin
...
  happy := false;
If toBe and not toBe
  WriteLn("The Answer!");
End;

Could "The Answer" ever be written?

- Suppose that toBe is a function nested in procedure Hamlet,
- which may have access to a global variable,
- whose initial value is false,
- In fact, function toBe returns the value of this global variable,
- just after flipping it!
- So, the answer is,...
Expressions changing the program’s state?

Nasty CS101 Exam Question

You are given a seemingly innocent Pascal code, and asked...

Procedure Hamlet;
VAR
    happy: Boolean;
Function toBe:Boolean;
Begin
    ...
    happy := not happy;
    toBe := happy
End;
Begin
    ...
    happy := false;
    If toBe and not toBe
        WriteLn("The Answer!");
End;

Could "The Answer" ever be written?

- Suppose that toBe is a function nested in procedure Hamlet,
- which may have access to a global variable,
- whose initial value is false,
- In fact, function toBe returns the value of this global variable,
- just after flipping it!
- So, the answer is,...

Yes!
Expressions changing the program’s state?

Nasty CS101 Exam Question
You are given a seemingly innocent Pascal code, and asked...
Procedure Hamlet;
VAR
  happy: Boolean;
Function toBe: Boolean;
Begin
...
  happy := not happy;
  toBe := happy
End;
Begin
...
  happy := false;
  If toBe and not toBe
    WriteLn("The Answer!");
End;

Could "The Answer" ever be written?

- Suppose that toBe is a function nested in procedure Hamlet,
- which may have access to a global variable,
- whose initial value is false,
- In fact, function toBe returns the value of this global variable,
- just after flipping it!
- So, the answer is,...
  Yes!
Expressions without side-effects?

What happened here?

- Expressions do not make sense without function calls
- Functions may invoke commands
- Commands, by definition, alter the program state!
- Worse, in some PLs, certain operators have side-effects

Would it be possible to prevent side-effects at the PL design level?

- Representation of state?
- How would you do I/O?
- In general, tough, but awkward

Obvious example, pure-ML
“Statement-expressions” in Gnu-C

An excerpt from Section 6.1 Statements and Declarations in Expressions of Chapter 6 Extensions to the C Language Family of the Gnu-C manual:

```c
({ int y = foo (); int z;
  if (y > 0) z = y;
  else z = - y;
  z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo()`.

Note

Gnu-C uses the misnomer “statement” instead of command
and in **Mock** ...

```c
return
if (  
    (while (*s++ = *t++) ;)
  >
  (while (*t++ == *s++) ;)
)
  3;
else
  while (*s++ != *t++)
  return 7;
```

**Huh?**

*What does this mean? Is this useful to anyone?*
“Command-expression”

Command expressions is an **idealistic notion**:

- Any expression may be substituted by a command
- Every command is an expression, so every command returns a value:
  - **Atomic**  atomic commands are expressions
  - **Sequence** the last expression
  - **Conditional** the selected branch
  - **Iteration** the last iteration? What if there were no iterations?
  - **Return** What value should “return 3” return?
Reasonable realizations of command-expression I

- In “Statement-Expressions” of Gnu-C
- ML, in with the semicolon, “;” operator:
  - takes 2 operands
  - computes the 1\text{st} operand…
  - and then discards it!
  - computes the 2\text{nd} operand…
  - and then returns it.

The ancient BCPL

```
Standard ML of New Jersey ...
- (1;2)*(3;4);
val it = 8 : int
```
Reasonable realizations of command-expression II

- **PostScript**
- **Icon**, in which every expression is a generator;
  - atomic expressions are things such as values, which can only yield one value;
  - iterations return a sequence of values;
  - sequencing means concatenating the output of generators

...
6. Commands

6.2. Recursive definitions
Expressions are recursively defined

*Naturally, each PL is different, but the general scheme is:*

Atomic expressions
- literals
- variable inspection

Expression constructors
- Operators such as “+”, “-”, ...
- Function call:

*The set of atomic expressions and the constructors’ set are PL dependent, but the variety is not huge.*
Function call expression constructor

Definition (Function call expression constructor [dynamic typing version])

If f is a function taking $n \geq 0$ arguments, and $E_1, \ldots, E_n$ are expressions, then

$$f(E_1, \ldots, E_n)$$

is an expression.

Definition (Function call expression constructor [static typing version])

Let f be a (typed) function of $n \geq 0$ arguments,

$$f \in \tau_1 \times \cdots \times \tau_n \rightarrow \tau.$$ 

Let $E_1, \ldots, E_n$ be expressions of types $\tau_1, \ldots, \tau_n$. Then,

$$f(E_1, \ldots, E_n)$$

is an expression of type $\tau$. 
Commands are also recursively defined!

Each PL is different. The scheme is the same, but the variety is huge:

Atomic commands (each PL is different)
- the empty command
- assignment
- “sequencers” (WTF? sequencers will be discussed later)

(huge variety)

Command constructors
- Block command constructor
- Conditional command constructor
- Iterative command constructor
- “try···catch···finally” command constructor
- “with” command constructor
Three atomic commands in **Pascal**

(Ignoring **goto**, the only sequencer of the language)

**Empty** Can you figure out where it hides?

```pascal
Procedure swap(Var a, b: Integer);
Begin
  a := a + b; b := a - b; a := a - b;
end;
```

**Assignment** As in the above,

```pascal
happy := not happy
```

**Procedure call** As in the above,

```pascal
WriteLn("The Answer!")
```

**Pascal** is a separatist language; the semicolon is not part of the command; hence, an empty command is hiding here.
More on PASCAL’s atomic commands

**Empty** no change to state; no computation; no textual representation; existence determined solely by context.

**Definition (Assignment atomic command)**

Let $v$ be a variable of type $\tau$, and let $E$ be an expression of type $\tau$, or of compatible type $\tau'$, $\tau' \leq \tau$. Then,

$$v := E$$

is an atomic command.

**Definition (Procedure call atomic Command)**

If $p$ is a procedure taking $n \geq 0$ arguments of types $\tau_1, \ldots, \tau_n$, and $E_1 \in \tau_1, \ldots, E_n \in \tau_n$ are expressions, then the procedure call

$$p(E_1, \ldots, E_n)$$

is an atomic command.
The advent of “expression oriented languages”

**Pascal** sharp distinction between expressions and commands
- distinction between *Function* and *Procedure*
- distinction between *expression* and *command*

**C, Java, Go, ...** blurred distinction:
- a procedure is a function returning *Unit*
- an *expression* is a *command*, more or less, and subject to PLs variety.
Two kinds of atomic commands in C++
(ignoring sequencers)

The empty command does not change the program state; does not perform any computation; textual representation is the semicolon, i.e., “;”

Expression marked as command An atomic command is also “an expression followed by a semicolon”, e.g.,

In C, assignment is an operator taking two arguments L (left) and R (right). The operator returns R, and as side-effect, assigns R into L.
# Command expressions in C

**Definition (Command expressions in C)**

*If* $E$ *is an expression, then* $E;$ *is a command.*

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expression</strong></td>
<td>evaluates to a value</td>
<td>$f() \ ? \ a + b \ : \ a - b$</td>
</tr>
<tr>
<td><strong>Command</strong></td>
<td>change program state (even vacuously)</td>
<td>$f() \ ? \ a + b \ : \ a - b;\ i = 0;$</td>
</tr>
<tr>
<td><strong>Variable definition</strong></td>
<td>creates a variable and binds a name to it</td>
<td>int $i;$</td>
</tr>
<tr>
<td><strong>Variable declaration</strong></td>
<td>makes a binding; variable must be created elsewhere</td>
<td>extern int $i;$</td>
</tr>
<tr>
<td><strong>Definition &amp; initializer</strong></td>
<td>creates a variable, binds a name to it, and initializes it</td>
<td>int $i = 3;$</td>
</tr>
<tr>
<td><strong>Declaration &amp; initializer</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All C’s atomic commands (including sequencers) are semicolon terminated

- *Not every command includes a semicolon*
- *Not every semicolon is part of a command*

Can you locate the atomic command(s) in this code?

```c
struct Complex {
    double x, y;
};

main() {
    int i, a[100];
    for (i = 0; i < 100; i++)
        a[i] = 100;
    return 0;
}
```

- *In fact, I am the only atomic command around here!*
- *We are three expressions, separated by semicolons, but none of us is a command!*
- *Yes, but you are a *sequencer*, and sequencers wait even longer!*
- *Not every semicolon makes a command*
- *Not every lonely semicolon makes an empty command*
- *Not every expression followed by a semicolon makes a command*
Two kinds of atomic commands in JAVA (ignoring sequencers)

Just as in C,

;  the empty command is a lonely semicolon;

Expression;  provided that the first step in the recursive decomposition of expression is “something” that has (might have) side-effects:

- Function call
- Operator with side effects:
  - Assignment e.g., =, +=, <<=,...
  - Increment/decrement ++ and --; either prefix or postfix.
- Object creation e.g., new Object()
- Nothing else!
Not all JAVA expressions make commands

- ; // ✓
- i++; // ✓
- ++i; // ✓
- ++i // ✗
- i++, j++ // ✗
- ;; // ✓ (two commands)
- i = f(); // ✓
- f(); // ✓
- new String(); // ✓
- new String() // ✗

no comma operator in JAVA

- j <<= g(); // ✓
- 0; // ✗
- f; // ✗
- ;f(); // ✓ (two commands)
- i++ + j++; // ✗
- i++ && j++; // ✗
- f() + 0; // ✗
- a ? f() : g() // ✗
- 1 << f(); // ✗
## 6. Commands

### 6.3. Atomic commands
Vanilla assignment command

\[ v \leftarrow e \] (3.1)

- Expression \( e \) is evaluated
- Its value is assigned to variable \( v \)
## Two variation of vanilla assignment

<table>
<thead>
<tr>
<th>Multiple</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1, v_2, \ldots, v_n \leftarrow e$</td>
<td>$v \leftarrow \varphi(\cdot, e_1, e_2, \ldots, e_n)$</td>
</tr>
</tbody>
</table>

- Expression $e$ is evaluated
- Its value is assigned to variables

### Multiple

- $v_1, \ldots, v_n$

- syntactic sugar for

$$v \leftarrow \varphi(v, e_1, e_2, \ldots, e_n)$$

### Update

- as in COBOL’s
  - Add 1 to a
- as in C/JAVA
  - `i++`
- as in C/JAVA
  - `i *= 3`
Two more varieties of the assignment command

**Collateral**

\[ v_1, v_2 \leftarrow e_1, e_2 \]  \hspace{1cm} (3.4)

- \( e_1 \) is evaluated and assigned to \( v_1 \)
- \( e_2 \) is evaluated and assigned to \( v_2 \)
- the two actions take place \textit{collaterally}
- cannot be used for swapping contents of variables
- theoretically possible, but not very useful

**Simultaneous**

\[ \langle v_1, v_2 \rangle \leftarrow \langle e_1, e_2 \rangle \]  \hspace{1cm} (3.5)

- \( e_1 \) is evaluated and then assigned to \( v_1 \) (as in collateral assignment)
- \( e_2 \) is evaluated and then assigned to \( v_2 \) (as in collateral assignment)
- the two actions take place \textit{simultaneously}
- can be used for swapping
- we had tuples of \textit{values}; \( \langle v_1, v_2 \rangle \)
- can be thought of as a tuple of \textit{variables}
And, what about the “forgotten” atomic commands?

The SKIP command aka NOP, aka \relax, aka “;”, aka …
- is not really interesting
- syntactically necessary on occasions

Procedure call command
- is not really interesting
- occurs only when procedures are distinct from functions; in most PLs, a procedure is just a function that returns \texttt{void} aka the Unit type.
6. Commands

6.4. Block commands
Sequential block constructor

**Definition (Sequential block constructor)**

If $C_1, \ldots, C_n$ are commands, $n \geq 0$, then

$$\{C_1; C_2; \ldots; C_n\} \quad (4.1)$$

is a composite command, whose semantics is **sequential**: $C_{i+1}$ is executed after $C_i$ terminates.

- **Most common constructor**
- **Makes it possible to group several commands, and use them as one**, e.g., inside a **conditional**
- **If your language has no skip command, you can use the empty sequence, {}**.

**Separatist Approach**: semicolon separates commands; used in **Pascal**; mathematically clean; error-prone.

**Terminist Approach**: semicolon terminates commands (at least atomic commands); used in **C/C++/Java/C# and many**
Collateral block constructor

Definition (Collateral block constructor)

If $C_1, \ldots, C_n$ are commands, $n \geq 0$, then

$$\{C_1 \sim C_2 \sim \cdots \sim C_n\} \quad (4.2)$$

is a composite command, whose semantics is that $C_1, \ldots, C_n$ are executed **collaterally**.

Very rare, yet (as we shall see) important

- Order of execution is **non-deterministic**
- An optimizing compiler (or even the runtime system) can choose “best” order
- Good use of this constructor, requires the programmer to design $C_1, \ldots, C_n$ such that, no matter what, the result is
  - *programmatically identical, or*
  - at least, *semantically equivalent*
Programmatically identical vs. semantically equivalent

Programmatically Identical

Now these are the generations of the sons of Noah, Shem, Ham, and Japheth: and unto them were sons born after the flood.

1 The sons of Japheth; Gomer, and Magog, and Madai, and Javan, and Tubal, and Meshech, and Tiras...

2 And the sons of Ham; Cush, and Mizraim, and Phut, and Canaan

3 The children of Shem; Elam, and Asshur, and Arphaxad, and Lud, and Aram

Semantically Equivalent

{  
    humanity.add("Adam");
    ~  
    humanity.add("Eve");
}

At the end, both "Adam" and "Eve" will belong to humanity; but the internals of the humanity data structure might be different.
Concurrent block constructor

Definition (Concurrent block constructor)

If $C_1,\ldots,C_n$ are commands, $n \geq 0$, then

\[
\{C_1|C_2|\cdots|C_n\}
\]

is a composite command, whose semantics is that $C_1,\ldots,C_n$ are executed concurrently.

Common in concurrent PLs, e.g., Occam

- Just like “collateral”...
- Commands can be executed in any order; order of execution is non-deterministic
- An optimizing compiler (or even the runtime system) can choose “best” order
- Good use of this constructor, requires the programmer to design $C_1,\ldots,C_n$; such that, no matter what, the result is, programmatically identical, or semantically equivalent
Collateral vs. concurrent collateral

Collateral really means “not guaranteed to be sequential”, or “undefined”; PL chooses the extent of defining this “undefined”, e.g.,

“the order of evaluation of \( a \) and \( b \) in \( a + b \) is unspecified. Also, the runtime behavior is undefined in the case \( a \) and \( b \) access the same memory”.

Concurrent may be executed in parallel, which is an extent of definition of a collateral execution.

“the evaluation of \( a + b \) by executing \( a \) and \( b \) concurrently; as usual, this concurrent execution is fair and synchronous, which means that…”.
Concurrent execution in **Occam**

**The cow**

```
PROC cow(CHAN INT udder!)
    INT milk: -- definitions are ':'
    terminated
    SEQ
    milk := 0
    WHILE TRUE
        SEQ
        udder ! milk
        milk := milk + 1
    : -- end of PROC cow
```

**The calf**

```
PROC calf(CHAN INT nipple?)
    WHILE TRUE
    INT milk:
    SEQ
    nipple ? milk
: -- end of PROC calf
```

**The cowshed**

```
PROC cowshed()
    CHAN INT mammaryGland:
    PAR
        calf(mammaryGland?)
        calf(mammaryGland?)
        calf(mammaryGland?)
        calf(mammaryGland?)
        cow(mammaryGland!)
: -- end of PROC cowshed
```
6. Commands

6.5. Conditional commands
Conditional commands

Definition (Conditional command constructor)

If $C_1, \ldots, C_n$ are commands, $n \geq 1$, and $E_1, \ldots, E_n$ are boolean expressions, then

$$\{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n\}$$

is a conditional command.
Semantics of conditional commands

\[ \{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n \} \]

Can be:

**Sequential:** Evaluate \( E_1 \), if true, then execute \( C_1 \), otherwise, recursively execute the rest, i.e., \( \{E_2 ? C_2 : \cdots : E_n ? C_n \} \).

**Collateral:** Evaluate \( E_1, E_2, \ldots, E_n \) collaterally. If there exists \( i \) for which \( E_i \) evaluates to true, then execute \( C_i \). If there exists more than one such \( i \), arbitrarily choose one of them.

**Concurrent:** Same as collateral, except that if certain \( E_i \) are slow to execute, or blocked, the particular concurrency regime, prescribes running the others.

Example of a concurrency regime:

**Strong fairness:**

*In any infinite run, there is no process which does not execute infinitely many times.*
CSP: Communicating sequential processes

**Occam** features a concurrent conditional command:

If none of the “guards” is ready, then the **ALT** commands waits, and waits, and waits.

- Deep theory of “communicating sequential processes”
- **ALT** is only a small part of it
- **but we must proceed in our course...**
The “else” variants

Definition (Conditional command constructor with else clause)

If $C_1, \ldots, C_n, C_{n+1}$ are commands, $n \geq 1$, and $E_1, \ldots, E_n$ are boolean expressions, then

$$\{E_1?C_1 : E_2?C_2 : \cdots : E_n?C_n : C_{n+1}\} \quad (5.2)$$

is a conditional command, whose semantics is the precisely the same as the familiar

$$\{E_1?C_1 : E_2?C_2 : \cdots : E_n?C_n\},$$

where we define

$$E_n = \neg E_1 \land \neg E_2 \land \cdots \land \neg E_{n-1} \quad (5.3)$$

The “else” clause is sometimes denoted by:

- default
- otherwise
Variant #1 / many: the “else” clause

Almost all languages use “else”
If thou Wilt Take The Left Hand
then
iWillGoToTheRight
else
iWillGoToTheLeft

**Pascal uses “Otherwise”**

```pascal
case expression of
    Selector: Statement;
...
    Selector: Statement
otherwise
    Statement;
...
    Statement
end
```
*(the GNU-PASCAL’s EBNF)*

**C uses “default”**

```c
int isPrime(unsigned c) {
    switch (c) {
        case 0:
        case 1: return 0;
        case 2:
        case 3: return 1;
        default:
            return isPrime(c);
    }
}
```
Variant #2 + #3 / many: if-then-else & cases

- Special construct for the case \( n = 1 \) in the form of
  \[
  \text{if \ Condition then \ Statement} \\
  \text{[ else Statement ]}
  \]

  *your syntax may vary*

- Special construct for the case that
  - each of \( E_i \) is in the form \( e = c_i \)
  - \( e \) is an expression (usually integral), common to all \( i = 1, 2, \ldots \)
  - \( c_i \) is a distinct constant expression for all \( i = 1, 2, \ldots \)

  \[
  \text{case \ Expression of} \\
  \{ \text{constantExpression \ Statement} \}^+ \\
  \text{[ otherwise \ Statement ]}
  \]

  *your syntax may vary*
Cases with range in **Pascal**

A selector of Pascal’s `case` statement may contain:
- Multiple entries
- Range entries

---

**ROT13 Filter in Pascal**

Program Rot13(Input, Output);

VAR
    c:Char;
Begin
    While not eof do begin
        Read(c);
        Case c of
            'a'..'m', 'A'..'M':
                Write(chr(ord(c)+13));
            'n'..'z', 'N'..'Z':
                Write(chr(ord(c)-13));
            otherwise
                Write(c);
        end
    end
end.
Why special switch/case statement?

Because the PL designer thought...

- it would be used often
- it has efficient implementation on “wide-spread” machines
  - Dedicated hardware instruction in some architecture
  - Jump-table implementation
  - Binary search implementation

The above two reasons, with different weights, explain many features of PL.

*these are precisely the reasons for the particular specification of conditional in the form of “if-then-else” for the cases $n = 1*
Efficient implementation + usability considerations = wrong conclusion?

Early versions of FORTRAN relied on a very peculiar conditional statement, namely *arithmetic if*

\[
\text{IF ( Expression ) } \ell_1, \ell_2, \ell_3
\]

where

- $\ell_1$ is the label to go to in case $Expression$ is negative
- $\ell_2$ is the label to go to in case $Expression$ is zero
- $\ell_3$ is the label to go to in case $Expression$ is positive

could be efficient, but not very usable in modern standards
Another weird (& obsolete) conditional statement

Early versions of *FORTRAN* had a “computed goto” instruction

\[
\text{GO TO } (\ell_1, \ell_2, \ldots, \ell_n) \text{ Expression} \tag{5.4}
\]

where

- \(\ell_1\) is the label to go to in case Expression evaluates to 1
- \(\ell_2\) is the label to go to in case Expression evaluates to 2
- \(\vdots\)
- \(\ell_n\) is the label to go to in case Expression evaluates to \(n\)

likely to have efficient implementation, but not very usable in modern standards
Cases variants?

- Range of consecutive integer values (in \texttt{Pascal})
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of \texttt{Java} after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in \texttt{Bash}
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in \texttt{ML} and other functional PLs
  - In the spirit of the PL type system
- No cases statement
  - In \texttt{Eiffel} a pure OO language
  - Language designer thought it encourages non OO mindset
Vanilla multiway conditional?

- Exists in many languages, in the form of a special keyword 
- `elseif`, or `elsif` or `ELIF`,
- e.g., in PHP you can write

```
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
```
else if? elseif? what’s the big difference?

- There is no big difference!
  
  ```
  else if  many levels of nesting
  elseif  one nesting level
  ```

- this might have an effect on automatic indentation, but modern code formatters are typically smarter than that!

- another small difference occurs if the PL requires the `else` part to be wrapped within "{" and "}".
6. Commands

6.6. Iterative commands

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage

6. Commands
   6.1 Expressions vs. commands
   6.2 Recursive definitions
   6.3 Atomic commands
   6.4 Block commands
   6.5 Conditional commands
   6.6 Iterative commands
   6.7 Structured programming
   6.8 Sequencers
   6.9 Exceptions
Iterative command constructor

A very general pattern of iterative command constructor

**Definition (Iterative command constructor)**

*If* $S$ *is a “program state generator” and* $C$ *is a command, then*

$$\text{forall } S \text{ do } C$$

*is an iterative composite command whose semantics is the (sequential / collateral / concurrent) execution of* $C$ *in all program states that* $S$ *generates.*

*Note that with “sequencers” such as break and continue, iterative commands can be even richer!*
State generator? answer #1/5

Range of integer (ordinal) values, e.g.,

```
For i := gcd(a,b) to lcm(a,b) do
  If isPrime(i) then
    Writeln(i);
```
The state generator $S$ may be...

Any arithmetical progression, e.g., in **FORTRAN**

```
INTEGER SQUARE11
SQUARE11 = 0
DO 1000 I = 1, 22, 2
  SQUARE11 = SQUARE11 + I
1000 CONTINUE
```
State generator? answer #3/5

The state generator $S$ may be...

Expression, typically boolean:

- expression is re-evaluated in face of the state changes made by the command $C$;
- iteration continues until expression becomes true, or,
- until expression becomes false,
State generator? answer #4/5

The state generator $S$ may be...

Generator, e.g., in **Java**

```
List<Thing> things = new ArrayList<Thing>();
...
for (Thing t : things)
    System.out.println(t);
```
State generator? answer #5/5

The state generator $S$ may be...

Cells in an array, e.g., in Java

```java
public static void main(String[] args) {
    int i = 0;
    for (String arg: args)
        System.out.println(
            "Argument" + ++i + ":" + arg
        );
}
```
Minor varieties of iterative commands

Minimal number of iterations?

**Minimal # Iterations = 0**

```java
while (s < 100)
    s++;
```

**Minimal # Iterations = 1**

```java
do {
    s++;
} while (s < 100);
```

Truth value for maintaining the iteration

**Iteration continues with true**

```pascal
While not eof do
Begin
    ...
end
```

**Iteration continues with false**

```pascal
Repeat
    ...
until eof
```

none of these is too interesting…
The iteration variable

Several iteration constructs, e.g., *ranges* and *arithmetical progressions*, introduce an “iteration variable” to the iteration body, e.g.,

```bash
#!/usr/bin/gawk -f
BEGIN {
  antonym["big"] = "small"
  antonym["far"] = "near"
  ...
  for (w in antonym)
    print w, antonym[w]
}
```

```java
int[] primes = new int[100];
for (int p = 1, i = 0;
     i < primes.length; i++)
  primes[i] = p = nextPrime(p);
for (int p: primes)
  System.out.println(p);
```
Subtleties of the iteration variable

Can you make an *educated* guess as to what should happen in the following cases

1. the value of the expression(s) defining the range/arithmetical progression change during iteration?
2. the loop’s body tries to change this variable?
3. the value of the iteration variable is examined after the loop?
Definite vs. Indefinite Iteration

To make an educated guess, let’s educate ourselves:

**Definite Loop**  Number of iterations is known before the loop starts

**Indefinite Loop**  A loop which is not definite

It is easier to optimize definite loops.

- Many PL try to provide specialized syntax for definite loops, because they perceived as more efficient and of high usability.
- Only definite loops may have collateral or concurrent semantics
- Even if a PL does not specify that loops are definite, a clever optimizing compiler may deduce that certain loops are definite, e.g.,

```c
for (int i = 0; i < 100; i++)
  ...
  // If loop body does not change i
  // the loop is effectively definite
```
So, let’s make our guesses...

1. the value of the expression(s) defining the range/arithmetic progression change during iteration...
   The iteration range, as well as the step value are computed only at the beginning of the loop. (Check the Fortran/Pascal manual if you are not convinced)

2. the loop’s body tries to change this variable...
   The loop body should not change the iteration variable; The PL could either issue a compile-time error message (Pascal), runtime error message (Java), or just state that program behavior is undefined.

3. What’s the value of the iteration variable after the loop?
   The iteration variable may not even exist after the loop (Java); or, its value may be undefined (Pascal).
   - the PL designer thought that programmers should not use the iteration variable after the loop ends
   - if the value is defined, then collateral implementation is more difficult
   - many architectures provide a specialized CPU instructions for iterations;
   - the final value of the iteration variable with these instructions is not always the same.
### 6. Commands

#### 6.7. Structured programming

<table>
<thead>
<tr>
<th>6. Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Expressions vs. commands</td>
</tr>
<tr>
<td>6.2 Recursive definitions</td>
</tr>
<tr>
<td>6.3 Atomic commands</td>
</tr>
<tr>
<td>6.4 Block commands</td>
</tr>
<tr>
<td>6.5 Conditional commands</td>
</tr>
<tr>
<td>6.6 Iterative commands</td>
</tr>
<tr>
<td>6.7 Structured programming</td>
</tr>
<tr>
<td>6.8 Sequencers</td>
</tr>
<tr>
<td>6.9 Exceptions</td>
</tr>
</tbody>
</table>

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage
Flowcharts: another method for command composition

- **Nodes:**
  - I/O:
    - read
    - print
    - display
    - ...
  - Controls:
    - start
    - stop
  - Empty: **skip**
  - assignment
  - Decision point
  - ...

  *nodes are in fact atomic commands*

- **Edges:** **goto**
Flowchart for “getting things done (GTD)"

How to process items in your incoming mailbox?

- **Inbox**
  - **Important item?**
    - yes: @SOMEDAY / MAYBE
    - no: some-thing youWant?
      - yes: @FILE
      - no: Urgent item?
        - yes: @PROJECTLIST
        - no:
          - has a deadline?
            - yes: @CALENDAR / TICKLER FILE
            - no: can be done in two minutes?
              - yes: Do it Now!
              - no: can be done now?
                - yes: Dele-gate!
                - no: Reference material?
                  - yes: Action-able item?
                    - yes: @PROJECTLIST
                    - no: Dele-gate!
                  - no: best use of my time if I do it myself?
                    - yes: @WAITINGFOR
                    - no: Trash

CS 234319: Programming Languages  Fall 2017 (self reading)  December 19, 2017  Unit 6.7: p. 2/11
Pros & cons of flowcharts

Pros
- Very visual
- Very colorful
- Can be aesthetically pleasing
- Can be understood by almost any one

Cons
- Do not scale
- Many competing standards
- Not necessarily planar
- Spaghetti code
- No one can really understand them
Challenge of understanding spaghetti code

- The program on the right does something useful!
Structured programming

...is a programming paradigm, characterized by

- **“Three Controls:** precisely three ways for marshaling control:
  1. **Sequence,** e.g., \( \text{begin } C_1; C_2 \ldots; C_n \text{ end} \) for \( n \geq 0 \)
  2. **Selection,** e.g., \( \text{if } \ldots \text{ then } \ldots \text{ elseif } \ldots \text{ else } \ldots \text{ endif} \).
  3. **Iteration,** e.g., \( \text{while } \ldots \text{ do } \ldots \text{ done} \)

- **Structured Control:**
  - All control commands are in fact, command constructors.
  - Control is marshalled through the program structure.

---

**Theorem (The structured programming theorem (Böhm-Jacopini, 1966))**

*Every flowchart graph \( G \), can be converted into an equivalent structured program, \( P(G) \).*
Nassi-Shneiderman diagram
A More Sane Graphical Approach

Main Idea  Programming is like tiling the plane.

Also Called  NSD, and “structograms”

Thought Of As  the visual definition of structured programming

Principles:

1. every command is drawn as a rectangle
2. every command has exactly:
   - One entry point
   - One exit point
3. a command may contain other commands
4. a command may be contained in other commands
### Compound commands in Nassi-Shneiderman diagrams

<table>
<thead>
<tr>
<th>Sequential Command</th>
<th>Conditional Command</th>
<th>Iterative Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Command</em> (_1)</td>
<td><img src="false" alt="Condition Diagram" /></td>
<td><img src="true" alt="Condition" /></td>
</tr>
<tr>
<td><em>Command</em> (_2)</td>
<td><img src="false" alt="Command" /></td>
<td><img src="true" alt="Command" /></td>
</tr>
<tr>
<td>(\vdots)</td>
<td><img src="false" alt="Command" /></td>
<td><img src="true" alt="Command" /></td>
</tr>
<tr>
<td><em>Command</em> (_n)</td>
<td><img src="false" alt="Command" /></td>
<td><img src="true" alt="Command" /></td>
</tr>
</tbody>
</table>

- Compound commands are rectangles which have smaller rectangles in them.
- Each rectangle may contain in it...
- Color is not part of the diagram.
- But we can add it anyway…
Matrix multiplication with Nassi-Shneiderman diagram

\[
\begin{align*}
&\text{do } i := 1 \text{ to } n \\
&\quad \text{do } j := 1 \text{ to } n \\
&\quad \quad \text{sum } := 0 \\
&\quad \quad \text{do } k := 1 \text{ to } n \\
&\quad \quad \quad \text{sum } += A[i,k] \times B[k,j] \\
&\quad \quad \text{C}[i,j] := \text{sum}
\end{align*}
\]

Based on an example provided by the original October 1973 “SIGPLAN Notices” article by Isaac Nassi & Ben Shneiderman
Factorial with Nassi-Shneiderman diagram

is \( n > 1 \) ?

- **No**
  - \( n_{fact} := 1 \)

- **Yes**
  - \( n_{fact} := 2 \)
  - \( \text{do } i := 3 \text{ to } n \)
  - \( n_{fact} := n_{fact} + 1 \)

return \( n_{fact} \)
More Nassi-Shneiderman notations

Switch Command

Repeat ... Until Command

Command

Condition

Case i do

default

C_4

C_7

...
Even more Nassi-Shneiderman notations

Nassi and Shneiderman did not fully work out the semantics of NSD;
- not in any formal notation;
- not in “legalese”.
- not in mock of “legalese”.

Some notation may be intriguing...

Concurrent Command Constructor

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td>$\cdots$</td>
<td>$C_n$</td>
</tr>
</tbody>
</table>
```

Begin ... End Iteration?

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>End</td>
</tr>
</tbody>
</table>
```
# 6. Commands

## 6.8. Sequencers
What are sequencers?

**Definition (Sequencers)**

*Sequencers are atomic commands whose execution alters the “normal” (structural) flow of control.*

Examples:

- **goto** from any program point to another
- **return** to the end of an enclosing function
- **break** out of an enclosing iteration
- **continue** to the head of an enclosing iteration
- **throw** exception, that transfers control to a handler in an invoking function
Labels

To denote where \texttt{goto} will go to, one needs a \textit{label}

\begin{definition}{Label}
A \textit{label} an entity which denotes an empty command in the program text; typically there are non-empty commands before and after the empty command that the label denotes.
\end{definition}

Labels are a \textit{deliberately disprivileged} entities.
Label “literal” & first class labels

Literal labels
The label itself “is” the empty command which it denotes:

- **Identifiers.** as in C and assembly PLs
- **Integers.** as in Pascal, Basic and Fortran

In Basic, all commands must be labeled in a strictly ascending order

First class labels
Basic, PL/I, and some other obscure languages, treat labels as first class values, which can be

- stored in variables
- passed as arguments,
- returned by functions, etc.

```
l1: label v = a > b?
l1: l2;
...
l2: goto v;
```
Declared vs. ad-hoc labels

Declared label as in **Pascal**; labels must be declared before they are used.

Ad-hoc labels as in C/C++

```c
int main() {
    http://www.cs.technion.ac.il/yogi;
    printf("Page loaded successfully\n");
}
```
Ad-hoc labels may generate subtle bugs

```c
int isPerfect(unsigned n) {
    switch (n) {
    default:
        return 0;
    case 6:
    case 28:
    case 496:
    case 8128:
    case 0x1FFF000u:
        return 1;
    }
    return 0;
}
```
The persecuted goto

Restrictions on goto...

- Only within a block structure. **FORTRAN**,
- Only goto within a function.
- C does not allow inter-functional goto, but gotos are allowed in and out of a block.
- No goto from a bracketed command into itself. **Pascal**
- No goto into a loop or into a conditional. **C**
- No goto into a compound command. **Pascal**
- No goto into a nested function. **Pascal and Algol**
- No goto at all. **Java!**
Goto to a nesting function

Labels obey scope rules:

- If a variable of a nesting function is recognized in a nested function, the nested function can also `goto` to a label defined the nesting function.
- In case of recursion, labels denote a program point in the current activation.
Problems of structured programming

A large portion of all software is dedicated to dealing with exceptional cases, erroneous inputs and funny situations, e.g., PASCAL code tends to be heavily nested and difficult to read:

```
If some error was discovered then Begin
  deal with it
End else Begin
  do a little bit more processing
  If another error was discovered then Begin
    deal with this error
  end else Begin
    continue processing
  If another problem has occurred then Begin
    deal with it
  end else Begin
    work a little bit more
    If oops, a problem of a different kind was found then
    Begin
      do something about it
    end else Begin
      continue to work
    end
End else Begin
  work a little bit more
  If oops, a problem of a different kind was found then
  Begin
    do something about it
  end else Begin
    continue to work
  end
End
```
Escapes

**Definition (Escape)**

An escape is a special kind of goto which terminates the execution of a compound command in which it is nested.

Makes single entry, multiple exit commands:

- **exit** in **Ada**
- **break** in **C/C++/Java**

Useful for simplifying nesting structure.
Varieties of escape

- *Escape any enclosing Loop.* `exit ℓ` in *Ada* and `break ℓ` in *Perl/JAVA*, where ℓ is a label of an enclosing loop.

- *Escaping out of a Function.* `return` in *C* and *Fortran*.

- *Terminal escape.* terminate the execution of the whole program; `halt` in *Fortran*.

- *Specialized escape.* `break` out of a `switch` command in *C*.
Continue

Definition (Continue)

A continue is a special kind of an escape which can only occur within a iteration command; it terminates the execution of the current iteration and if there is a next iteration, it proceeds to it.

Just like break, useful for simplifying nesting structure.

- Continue any Enclosing Loop. continue \( \ell \) in JAVA, where \( \ell \) is a label of an enclosing iteration command, proceeds to the next iteration of the of the iteration command marked by \( \ell \).

- Cannot be emulated by PASCAL goto, due to restrictions that PASCAL places on goto.
6. Commands

6.9. Exceptions

1. Preliminaries

2. Introduction

3. Values and types

4. Advanced typing

5. Storage

6. Commands

6.1 Expressions vs. commands

6.2 Recursive definitions

6.3 Atomic commands

6.4 Block commands

6.5 Conditional commands

6.6 Iterative commands

6.7 Structured programming

6.8 Sequencers

6.9 Exceptions

6.9.1 Robustness

6.9.2 Policy I: resumption

6.9.3 Policy II: error rolling

6.9.4 Policy III: setjmp/longjmp of C

6.9.5 Policy IV: exceptions

6.9.6 Kinds of exceptions

6.9.7 Resource acquisition is (or isn’t) initialization
Sometimes... it does not make sense to proceed as usual!

Two kinds of abnormal and unusual situations

**Bugs**

An error in the program code:

*Runtime type error* (in dynamically typed languages)

*Pseudo type error* division by zero, array bounds overflow, dereferencing null pointer,...

*Resources issue* stack overflow, memory exhaustion,...

**Unusual environment**

The program encounters an environment for which normal execution does not make sense:

- wrong password
- file not found
- low battery
- no connection to server
- cannot locate GPS
- volume is off

(Recall (Subunit 3.2.2) that a pseudo-type error is subjecting a value to an operation which is inappropriate it, but is appropriate for other values of the same type.)
6. Commands

6.9. Exceptions

6.9.1. Robustness
When you cannot proceed as usual...

It is tough to write robust programs; sometimes 80% of the code is dedicated to abnormal situations.

There is an "exception"!

**Bugs** The "language runtime environment" must take action, printing error messages, and enforcing graceful termination

**Unusual environment** programmer must deal with the error; if the programmer fails to detect an "unusual environment", then it is a bug.
Robust PL vs. robust programs

**Definition (Robust PL)**

The PL’s runtime system recovers gracefully from all program bugs

“the programmer cannot make the runtime system crash”

**Definition (Robust program)**

The program recovers gracefully even in the face of weird end-user and execution environment errors

“the user cannot make the runtime system crash”
The fundamental theorem of exception handling

**Theorem (The robust programs theorem)**

No one really knows how to write robust programs

**Proof.**

Exceptions are detected at lower level of abstraction:
- Wrong keyboard input
- Missing file
- Internet problem
- Low battery
- Out of memory

but they must be handled in higher levels of abstraction.
Handling exceptions

Handling must be done at a high abstraction level. Challenges include:

- **Consistency**: Deal with similar errors similarly (tough because many details of the errors are lost at higher abstraction level)
- **Coverage**: make sure that all errors are covered appropriately, and that no two dissimilar errors are grouped together for the purpose of handling. (tough, because the programmer does not always know what errors may happen at lower levels of abstraction)
- **Smart recovery**: Sometimes, by the time the exception is caught, there is nothing useful that can be done.
- **Systematic testing**: It is tough to systematically generate all exceptions.
Corollaries of the robust programs theorem

Since no one know how to do it, no one can design for it

Humanity cannot do what it does not know how to do

**Corollary (Adequate support for exception handling)**

No PL provides adequate support for exception handling.

**Corollary (Rareness of robust programs)**

Very few programs are truly robust

And since programmers are lazy,

**Corollary (Graceful termination)**

Many PLs offer graceful termination in case of bugs.

**Corollary (Input errors considered bugs)**

Most programs convert input errors into bugs.
Example: Heron’s formula for the area of the triangle

Hero of Alexandria (10–70 AD)
Also known as Heron

\[ A = \sqrt{s(s-a)(s-b)(s-c)} \]
\[ s = \frac{a+b+c}{2} \]  

Exceptions
1. Cannot read \( a \)
2. Cannot read \( b \)
3. Cannot read \( c \)
4. \( a < 0 \)
5. \( b < 0 \)
6. \( c < 0 \)
7. \( s < 0 \)
8. \( s - a < 0 \)
9. \( s - b < 0 \)
10. \( s - c < 0 \)
Non-robust program for the area of the triangle I

1. Let’s start implementing it.
2. First, essential include files
   ```
   #include <stdio.h>
   #include <math.h>
   ```
3. Then, a function to prompt for and read real numbers:
   ```
   double get(const char *s) {
     double r;
     printf("Please enter %s: ", s);
     scanf("%lg", &r);
     return r;
   }
   ```
4. Invalid input? End of file?
   We do not care, so let’s carry on…
Non-robust program for the area of the triangle II

```cpp
double A(double s, double sa, double sb, double sc) {
    return sqrt(s * sa * sb * sc);
}
```

5. Square root of a negative number?
   We don't care!

6. Proceeding…

```cpp
double A(double a, double b, double c) {
    double s = (a+b+c) / 2;
    return A(s, s-a, s-b, s-c);
}
```

7. Is either of \( s - a \), \( s - b \), or \( s - c \) negative?
   We proceed…
Non-robust program for the area of the triangle III

```c++
int main() {
    double a = get("A");
    double b = get("B");
    double c = get("C");
    printf("Area is: \%g \n", A(a, b, c));
    return 0;
}
```

And, what happens if the three sides do not satisfy the triangle inequality?

**Answer**

If you check closely, you will find that this case is covered above.
6. Commands

6.9. Exceptions

6.9.2. Policy I: resumption
Resumption

- The offending command continues as usual after the handler was invoked.
- Found in PL/I and Eiffel.
- In C++, function `set_new_handler()` takes as argument a pointer to a handler function for failure of the `new` memory allocation operator.

Motivating example: memory exhausted exception

the memory exhaustion handler can:

- free some non-essential memory
- invoke the garbage collector

but, these efforts are not guaranteed to be sufficient
Pros & cons of resumption

Pros   The offending command is not aware of the problem nor of the solution

Cons   • **Perplexing situation:** you need the context to deal with the error, but you don’t have it.
        • **Hardly ever works:** in most cases, the handler can try, but cannot promise to fix the problem
        • **Hardly used:** experience shows that resumption policy is not used very often even in PLs that support it
Emulating resumption in Heron’s program

Original: naive read

```c
double get(const char *s) {
    double r;
    printf("Please enter %s:", s);
    scanf("%lg", &r);
    return r;
}
```

- Unpredicted result in case of invalid input
- Repeated unpredictable result

Retrying the Read

```c
double get(const char *s) {
    double r;
    do {
        printf("Please enter %s:", s);
        scanf("%lg", &r);
    } while (1 != scanf("%lg", &r));
    return r;
}
```

- Loop forever on EOF.
- Loop forever in case of invalid input

`scanf` does not consume unexpected characters in the input stream

Correct and useful “corrective” action?

Easier said than done!
6. Commands

6.9. Exceptions

6.9.3. Policy II: error rolling
Explicit error handling: returning error code

**Step 1:** make low-level functions return error code

```c
double get(const char *s) {
    double r;
    int n;
    printf("Please enter \%s: ", s);
    n = scanf("%lg", &r);
    if (n < 0) { // Error or EOF
        if (errno != 0) perror(s);
        return -1; // error code
    }
    if (n == 0) // Unexpected input
        return -1; // error code
    if (r < 0)
        return -1; // error code
    return r;
}
```
More error code returned

Step I: another low-level function...

Computing the area

```cpp
double A(double s, double sa, double sb, double sc) {
    if (s < 0 || sa < 0)
        return -1; // error code
    if (sb < 0 || sc < 0)
        return -1 // error code
    return sqrt(s * sa * sb * sc);
}
```

Issues:

- Which error code to use?
- Which errors to detect?
- Should we report some errors?
Error handling

Step II: dealing with the propagated errors

More issues:

- How would the user know whether the error was in a, b, or c?
- When should we retry reading?
- Can all errors happen?
Error rolling: summary

Every procedure returns a special error code

**Assembler** usually the carry flag

**ICON** each function returns its success value in addition to its real value

**C convention** returns 0 or negative value for integers, **NaN** for floating point values,

**Go convention** procedures return a pair of values:
- the actual value, *and*
- error status or code.

**Pascal** and old C do not allow functions returning structure, so a specialized error value was easy to select.
Summary: using the error code

The invoking procedure checks this code and tries to recover.

Assembler  call proc; jc error

C  if ((f = fopen(...)) == 0) ...

heavy responsibility on the programmer;
- always remember to test error codes;
- propagate errors up sensibly;
- recover from errors gracefully;

most programmers prove to be irresponsible
6. Commands

6.9. Exceptions

6.9.4. Policy III: setjmp/longjmp of C
How does it work?

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf b;

void g(void) {
    // will print:
    printf("g()\n");
    // jumps back to where setjmp was
called and saved current CPU state
    // in b making setjmp now return 1:
    longjmp(b, 1);
}

void f(void) {
    g();
    // will not print
    printf("f()\n");
}

int main() {
    // when executed first,
    // function setjmp returns 0
    if (0 == setjmp(b))
        f();
    else // when longjmp jumps back,
        // function setjmp returns 1
        // will print
        printf("main()\n");
    return 0;
}
```
What’s stored in a jump buffer?

- program counter
- stack pointer
- other CPU registers
- any thing else required to restored the CPU state

In file `/usr/include/stdio.h` on my machine, you will find something like (a more complex version of)

```c
struct jmp_buf {
    long int data[8]; // for registers probably
    int mask_was_saved; // who noz?
    unsigned long mask[16]; // mask for interrupts
};

typedef struct jmp_buf jmp_buf[1];
```
Step I: low-level functions call `longjmp`

Reading doubles

```c
double get(const char *s) {
    double r;
    int n;
    printf("Please enter \%s:", s);
    n = scanf("%lg", &r);
    if (n < 0) {
        // Error or EOF
        if (errno != 0) {
            perror(s);
            longjmp(b, 1); // exception code
        }
        longjmp(b, 2); // exception code
    }
    if (n == 0) // Unexpected input
        longjmp(b, 3); // exception code
    if (r < 0) // Negative value
        longjmp(b, 4); // exception code
    return r;
}
```
More low-level functions calling **`longjmp`**

**Computing the area**

```c
double A(double s, double sa, double sb, double sc) {
    if (s < 0) longjmp(b, 5);
    if (sa < 0) longjmp(b, 6);
    if (sb < 0) longjmp(b, 7);
    if (sc < 0) longjmp(b, 8);
    return sqrt(s * sa * sb * sc);
}
```

**Issues:**

- Managing and passing the `jmp_buff` record
- Managing exception codes
- Which errors to detect?
- Should we print an error message on some errors?
- Is it possible that we access uninitialized stack variable?
Step II: catching the longjmp

```c
int main() {
    switch (setjmp(b)) {
    case 0: {
        double a = get("A");
        double b = get("B");
        double c = get("C");
        printf("Area is: \%g\n", area(a,b,c));
        return 0;
    }
    case 1: ... break;
    case 2: ... break;
    ... 
    case 8: ... break;
    }
}
```
Long jump in C & Pascal

Pascal can execute a goto to any nesting block;
- Can only goto to deepest occurrence of nesting block
- No fine control in recursive calls

C setjmp and longjmp allow to jump outside an internally invoked function

setjmp(b) saves all CPU registers in buffer b
longjmp(b, v) restores the registers from b and make setjmp return v

Lots of unsafe properties
6. Commands

6.9. Exceptions

6.9.5. Policy IV: exceptions
Exception language construct I

1. Include files:

```cpp
#include <stdio.h>
#include <math.h>
```

2. Read a floating point number:

```cpp
double get(const char *s) {
    double r;
    printf("Please enter \%s: ", s);
    if (1 == scanf("%lg", &r)) return r;
    throw 1;
}
```

3. Check that a number is positive:

```cpp
void v(double v, const char *error) {
    if (v <= 0) throw error;
}
```
Exception language construct II

Compute triangle’s area using Heron’s formula:

doctor A(double s, double sa, double sb, double sc) {
    v(s, "Circumference must be positive");
    v(sa, "Side \( A \) is too large");
    v(sb, "Side \( B \) is too large");
    v(sc, "Side \( C \) is too large");
    return sqrt(s * sa * sb * sc);
}
Exception language construct III

Now, our main function is both elegant and robust:

```c++
int main() {
    for (;;)
    try {
        double a = get("A");
        double b = get("B");
        double c = get("C");
        printf("Area is: %g\n", A(a,b,c));
        return 0;
    } catch (const char *s) {
        fprintf(stderr, "Error: %s\n", s);
    } catch (...) {
        fprintf(stderr, "Bad input\n");
    }
}
```
6. Commands

6.9. Exceptions

6.9.6. Kinds of exceptions
Summary: policies for exception handling

**Resumption**  
Resume as usual after detection of exception

**Explicit**  
Explicit error handling: every procedure returns an error code which the invoking procedure has to check

**Long jump**  
Moving from a low level of abstraction directly to a higher level of abstraction

**Language construct**  
Exception language mechanism

**Observation**

In C++, you must have exceptions since constructors return no value. You cannot use explicit error handling for constructors.
Exception type?

What can be thrown?

**Pascal** The **Unit** type: goto to a nesting block has no associated value

**C** Integer (with **longjmp**)  

**C++** Any type  

**Java** Subclass of **Throwable**  

**Ada** Any enumerated type

---

Tough language definition issue

Suppose that a function $f$ may throw exceptions of types $E_1, \ldots, E_n$. Then, should the type of $f$ also include $E_1, \ldots, E_n$ as the possible exceptions that $f$ may return?

Exceptions seem to be the right solution, but, no one knows how to do these right
The checked exceptions dilemma

No one really knows how to type-check exceptions in a way that should be productive to the programmer.

**JAVA** tries at systematic classification of exceptions
- Function types includes a list of exceptions
- But not exceptions thought to be *bugs* such as null pointer and division by zero
- Static type checking on compile type
- But lax checking by the JVM at runtime

**C#** learned **JAVA**’s bitter lesson
- Exceptions are not part of the function type
- No checked exceptions

**C++** Similar to **C#**.
- Function types *may* include a list of exceptions
- If the list exists, it is checked in runtime
- But, in recent versions of **C++**, there is no runtime checking
Finally?

Exception catching in Java

```java
try {
    C // Commands which may raise an exception
} catch(E1) { // First exception type
    H1 // Commands for handling first exception
} catch (E2) { // Second exception type
    H2 // Commands for handling second exception
}
...
finally ... {
    // Commands to execute regardless of
    // whether an exception was thrown
}
```

Why finally?

Simple answer  Because the handlers may throw exceptions themselves.

Real answer   For managing allocated resources.
6. Commands

6.9. Exceptions

6.9.7. Resource acquisition is (or isn’t) initialization
The multiple resource allocation problem

Consider an application that needs to open two files, $f_1$ and $f_2$. If it opens $f_1$ and discovers an error in $f_2$ is must remember to close $f_1$ as well.

```java
void f(File f1, File f2) {
    try {
        PrintWriter o1 = new PrintWriter(f1)
        try {
            PrintWriter o2 = new PrintWriter(f2);
            
            // Not all exceptions are caught here:
        } catch (IOException e) {
            // The following may also raise an exception:
            System.out.println(f1 + "I/O error");
        }
        finally {
            o1.close();
        }
    } catch (IOException e) {
        System.out.println(f1 + "I/O error");
    }
}
```

a bit cumbersome
Resource acquisition is initialization (RAII)

When a block goes out of scope

**Java** (and other language with garbage collection) Local variables wait for the GC.

**Pascal** (and other language which use the stack memory model)
Variables allocated on the stack are reclaimed

**C++** Variables allocated on the stack are *destructed*. This occurs in two cases:
- The block terminates normally
- An exception is thrown out of the block

---

**Definition (RAII)**

A *common programming idiom, by which resources are allocated by the constructor and deallocated by the destructor.*
RAII example: a simple file reader class

```cpp
struct Reader {
    FILE *f;
    Reader(const char *name): f(fopen(name, "r")) {
        if (!f) throw name;
    }
    ~Reader() { fclose(f); }
    int eof() { return feof(f); }
    int get() {
        if (ferror(f)) throw "read";
        return fgetc(f);
    }
};
```

The file is opened when a `Reader` object is created; the file is closed when this object is destructed.
RAII example: a simple file writer class

```cpp
struct Writer {
    FILE *f;
    Writer(const char *name): f(fopen(name,"w")) {
        if (!f) throw name;
    }
    ~Writer() { fclose(f); }
    int put(int c) {
        if (ferror(f)) throw "write";
        return fputc(c,f);
    }
};
```

The file is opened when a `Writer` object is created; the file is closed when this object is destructed.
RAII example: using the reader & writer classes

Robust copying of a file

```c++
int main() {
    try {
        // Opening two resources and using them
        Reader f1("from.txt"); // No need to close
        Writer f2("to.txt"); // No need to close
        while (!f1.eof())
            f2.put(f1.get());
    } catch (const char *s) {
        // File(s) are properly closed,
        // even in the case of exception
        perror(s);
        return 1;
    }
    return 0;
}
```
Life without RAII is tough!

Java has no destructors, so RAII is difficult

Robust “Hello, World” (version I)

```java
private void f() {
    FileOutputStream f = null;
    // May Throw An Exception = MTAN
    try {
        f = new FileOutputStream("hello.txt");
        byte[] bs = "Hello,
World\n".getBytes(); // MTAN
        f.write(contentInBytes); // MTAN
        f.flush(); // MTAN
        f.close(); // MTAN
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            // Are you certain that file f
            // will not be closed twice?
            if (f != null)
                f.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Tough, and error prone, because we do not know the file’s state when the exception is caught.
Automatic deallocation: new **Java 7** feature

A **try** clause may allocate a resource

- **Resource is managed by the try block**
- A call to **close** is silently added to **finally** block
- **If there is no finally block, such a block is automatically created**
- **The call to close may still throw an exception, which must be handled by the programmer**

---

**Robust “Hello, World” (version II)**

```java
private void f() {
    try (FileOutputStream f = new FileOutputStream("hello.txt");) {
        byte[] bs = "Hello, \nWorld\n".getBytes(); // MTAN
        f.write(contentInBytes); // MTAN
        f.flush(); // MTAN
    } catch (IOException e) {
        e.printStackTrace();
    } // File f will be closed here if it was opened
}
```
Multiple resource acquisition with new Java 7 feature

Multiple resource allocated in a `try` will be automatically deallocated, and in the right order; only the resources which were allocated are deallocated; no need for an explicit `close` in the `finally` section;

```java
private void f() {
    try (FileInputStream in =
         new FileInputStream("from.txt");
         FileOutputStream out =
             new FileOutputStream("to.txt");) {
        String line = null;
        while ((line = in.readLine()) != null) {
            out.writeLine(line);
            out.newLine();
        }
    } // Files in and out will be closed here
    // (if they were opened)
}
```