Where are we?

6. Imperative programming

6.1. Commands

1. Preliminaries

2. Introduction

3. Values and types

4. Advanced typing

5. Storage

6. Imperative programming

6.1  Commands

6.1.1  Commands vs. expressions

6.1.2  Recursive definitions

6.1.3  Expressions’ evaluation order

6.1.4  Atomic commands

6.1.5  Block commands

6.1.6  Conditional commands

6.1.7  Iterative commands
Where are we?

6. Imperative programming

6.1. Commands

6.1.1. Commands vs. expressions
Where are we?

6. Imperative programming

6.1. Commands

6.1.1. Commands vs. expressions

5. Storage

6. Imperative programming

6.1. Commands

6.1.1. Commands vs. expressions

6.1.2. Recursive definitions

6.1.3. Expressions' evaluation order

6.1.4. Atomic commands

6.1.5. Block commands

6.1.6. Conditional commands

6.1.7. Iterative commands
Commands: what are they?
Commands are characteristic of imperative 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

---

1No commands in purely functional languages

J. Gil (Technion--IIT)
Commands vs. statements

- 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 vs. expressions

Ideally, they should be distinct

**Commands**
- Change state
- No value

**Expression**
- No state change
- Produce a value
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
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
Expressions changing the program’s state?

Nasty CS101 Exam Question

*You are given a seemingly innocent Pascal code,*
Expressions changing the program’s state?

Nasty CS101 Exam Question

*You are given a seemingly innocent* Pascal code,

```
Procedure Hamlet;
Begin
  ...  
      If toBe and not toBe
      WriteLn("The Answer!");
End;
```

In fact, function `toBe` returns the value of this global variable, just after flipping it! So, the answer is, … Yes!

J. Gil (Technion–IIT)
Expressions changing the program's state?

Nasty CS101 Exam Question

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

```pascal
Procedure Hamlet;
Begin
...
    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;
Begin
...
  If toBe and not toBe
    WriteLn("The Answer!");
End;

Could "The Answer" ever be written?
Expressions changing the program’s state?

Nasty CS101 Exam Question

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

Procedure Hamlet;
Begin
  ...
  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,
Expressions changing the program’s state?

Nasty CS101 Exam Question

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

**Procedure** Hamlet;
  **Function** toBe:Boolean;
  **Begin**
    ...
  **End**;
**Begin**
  ...
  **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,
Expressions changing the program’s state?

Nasty CS101 Exam Question

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

```pascal
Procedure Hamlet;
    Function toBe:Boolean;
    Begin
        ...
    End;
Begin
    ...
    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,
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
    ...
    End;
Begin
  ...
  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,
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
  ... 
  End;
Begin
  ... 
  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,
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
  ...
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*,
Expressions changing the program’s state?

Nasty CS101 Exam Question

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

Procedure Hamlet;

```pascal
VAR
  happy: Boolean;
Function toBe:Boolean;
Begin
...
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,
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
        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,
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
        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!
Expressions changing the program’s state?

Nasty CS101 Exam Question

*You are given a seemingly innocent* Pascal *code, and asked…*

**Procedure Hamlet;**

```pascal
VAR
  happy: Boolean;
Function toBe:Boolean;
Begin
  happy := not happy;
  toBe := happy
End;
```

```pascal
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!
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 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:

\[
\begin{align*}
\text{int } y &= \text{foo}(); \\
\text{int } z &= \text{foo}(); \\
\text{if } (y > 0) z &= y; \\
\text{else } z &= -y; \\
z; 
\end{align*}
\]

is a valid (though slightly more complex than necessary) expression for the absolute value of \text{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 1st operand...
  - and then discards it!
  - computes the 2nd operand...
  - and then returns it.

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

- The ancient BCPL
Reasonable realizations of command-expression II

- PostScript
Reasonable realizations of command-expression II

- PostScript
- Icon, in which every expression is a generator;
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;
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;
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
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
  - ...
Where are we?

6. Imperative programming
   6.1. Commands
       6.1.2. Recursive definitions
Where are we?

6. Imperative programming

6.1. Commands

6.1.2. 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, the

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

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**
- the empty command
- ... ²
- “sequencers”³

**Command constructors**
- Block command constructor (huge variety)
- Conditional command constructor (huge variety)
- Iterative command constructor (huge variety)
- “try-catch-catch-⋯-finally” command constructor (huge variety)
- “With” command constructor (huge variety)
- ... ⁴

² each PL is different
³ WTF? sequencers will be discussed later
⁴ each PL is different
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!")
```
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,

\[
v := E
\]  

(6.1)

is an atomic command.

Assignment

Definition (Procedure call atomic Command)

If \( p \) is a procedure taking arguments of types \( \tau_1, \ldots, \tau_n \), where \( n \geq 0 \) 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., “;”
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., “;”

```
while (*s++ = *t++)
;
```
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., “;”

Empty command; no need for loop “body”; all work is carried out by the side-effects of the expression used in the loop condition

while (*s++ = *t++)
;

while (*s++ = *t++)
;
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.,

```c
while (*s++ = *t++)
;
```
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.,

```c
while (*s++ = *t++)
;
```

```c
0; i*1; i;
(i=i)==-i;
i++ + ++i;
```
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>$f() \ ? \ a + b : a - b;$</td>
</tr>
<tr>
<td><strong>Variable definition</strong></td>
<td>creates a variable and binds a name to it</td>
<td><code>int i;</code></td>
</tr>
<tr>
<td><strong>Variable declaration</strong></td>
<td>makes a binding; variable must be created elsewhere</td>
<td><code>extern int i;</code></td>
</tr>
<tr>
<td><strong>Definition + initializer</strong></td>
<td>creates a variable, binds a name to it, and initializes it</td>
<td><code>int i = 3;</code></td>
</tr>
</tbody>
</table>
More on atomic expressions in C

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;
}
```
Two kinds of atomic commands in JAVA\textsuperscript{5}

Just as in $C$, the empty command is a lonely semicolon;

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

\begin{itemize}
  \item Function call
  \item Operator with side effects:
    \begin{itemize}
      \item Assignment \texttt{e.g.}, $=, +=, <<=, \ldots$
      \item Increment/decrement \texttt{++} and \texttt{--}; either prefix or postfix.
      \item Object creation \texttt{e.g.}, \texttt{new Object()}
    \end{itemize}
  \item Nothing else!
\end{itemize}

\footnote{\textsuperscript{5}ignoring sequencers}
Not all Java expressions make commands

- ;

no comma operator in Java
Not all JAVA expressions make commands

- ; ✓

no comma operator in JAVA
Not all JAVA expressions make commands

- ;
- i++;

no comma operator in JAVA
Not all Java expressions make commands

- ; ✓
- i++; ✓

no comma operator in Java
Not all **JAVA** expressions make commands

- ; ✓
- i++; ✓
- ++i;

**no comma operator in JAVA**
Not all JAVA expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓

no comma operator in JAVA
Not all JAVA expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i

no comma operator in JAVA
Not all JAVA expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i ✗

no comma operator in JAVA
Not all JAVA expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i ✗
- i++, j++

no comma operator in JAVA
Not all **JAVA** expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i ✗
- i++, j++ ✗

no comma operator in **JAVA**
Not all Java expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i ✗
- i++, j++ ✗
- ;;

no comma operator in Java
Not all JAVA expressions make commands

- ; ✔
- i++; ✔
- ++i; ✔
- ++i ✗
- i++, j++ ✗
- ;; ✔(two commands)

no comma operator in JAVA
Not all Java expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i X
- i++, j++ X
- ;; ✓ (two commands)
- i = f();

no comma operator in Java
Not all **JAVA** expressions make commands

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

**no comma operator in JAVA**
Not all JAVA expressions make commands

- ; ✓
- i++; ✓
- ++i; ✓
- ++i X
- i++, j++ X
- ;; ✓ (two commands)
- i = f(); ✓
- f();

no comma operator in JAVA
Not all **Java** expressions make commands

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

no comma operator in **Java**
Not all JAVA expressions make commands

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

no comma operator in JAVA
Not all JAVA expressions make commands

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

no comma operator in JAVA
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
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
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. Gil (Technion–IIT)
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
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
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
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
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
Not all JAVA expressions make commands

- `;` ✓
- `i++;` ✓
- `++i;` ✓
- `++i` ✗
- `i++, j++;` (two commands)
- `j <<= g();` ✓
- `0;` ✗
- `f;` ✗
- `;f();`

- `i++ + j++;` (two commands)
- `i++ && j++;` ✓
- `f() + 0;` ✓
- `a ? f() : g();` ✓
- `1 << f();` ✓

no comma operator in JAVA
Not all Java expressions make commands

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

no comma operator in Java
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++;`
Not all Java expressions make commands

- `i++` ✓
- `++i` ✓
- `j <<= g(); ✓`
- `0; X`
- `f(); X`
- `;f(); ✓ (two commands)`
- `i++ + j++; X`

- `i++; ✓`
- `i += 1; ✓`
- `i++; X`
- `i++, j++ X`
- `i++, j++ X`
- `i += 1, j += 1; ✓ (two commands)`
- `i = f(); ✓`
- `f(); ✓`
- `new String(); ✓`
- `new String(); ✓`
- `no comma operator in Java`
- `j <<= g(); ✓`
- `0; X`
- `f(); X`
- `;f(); ✓ (two commands)`
- `i++ + j++; X`

Java does not have a comma operator.

J. Gil (Technion–IIT)
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++;
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++;  


Not all JAVA expressions make commands

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

No comma operator in JAVA

- j <<= g(); ✓
- 0; X
- f; X
- ;f(); ✓ (two commands)
- i++ + j++; X
- i++ && j++; X
- f() + 0;
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(); ✗ (two commands)
- i++ + j++; ✗
- i++ && j++; ✗
- f() + 0; ✗
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
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() ✗`

J. Gil (Technion–IIT)
CS 234319: Programming Languages
January 31, 2015
Not all JAVA expressions make commands

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

No comma operator in JAVA

- j <<= g(); ✓
- 0; X
- f; X
- ;f(); ✓ (two commands)
- i++ + j++; X
- i++ && j++; X
- f() + 0; X
- a ? f() : g() X
- 1 << f();
# Not all JAVA expressions make commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Verdict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>;</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>i++;</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>++i;</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>++i</code></td>
<td>✗</td>
<td>no comma operator in JAVA</td>
</tr>
<tr>
<td><code>i++, j++</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>;;</code></td>
<td>✓ (two commands)</td>
<td></td>
</tr>
<tr>
<td><code>i = f();</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>f()</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>new String();</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>new String()</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>j &lt;&lt;= g();</code></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>0;</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>f;</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>;f();</code></td>
<td>✓ (two commands)</td>
<td></td>
</tr>
<tr>
<td><code>i++ + j++;</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>i++ &amp;&amp; j++;</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>f() + 0;</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>a ? f() : g()</code></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><code>1 &lt;&lt; f();</code></td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
Where are we?

6. Imperative programming
6.1. Commands
6.1.3. Expressions’ evaluation order
Where are we?

6. Imperative programming

6.1. Commands

6.1.3. Expressions’ evaluation order
Evaluation order (revisiting expression & commands)

Is this program “correct”?

```c
#include <stdio.h>

int main() {
    return printf("Hello,\n") - printf("World!\n");
}
```

Function printf returns the number of characters it prints.

C sizeof "Hello,\n" == 7

C sizeof "World!\n" == 7

So, main returns 0, which means normal termination.

J. Gil (Technion–IIT) CS 234319: Programming Languages January 31, 2015 Section 6.1: p. 23/71
6. Imperative programming

6.1. Commands / 6.1.3. Expressions’ evaluation order

Expressions + side-effects = evaluation order dilemma

In evaluating, e.g., $X + Y$, the PL can decide which of $X$ and $Y$ is evaluated first, but, most PLs prefer to refrain from making a decision.

Definition (Collateral evaluation)

Let $X$ and $Y$ be two code fragments (expressions or commands). Then, all of the following are correct implementations of collateral execution of $X$ and $Y$

1. $X$ is executed before $Y$
Expressions $+$ side-effects $=$ evaluation order dilemma

In evaluating, e.g., $X + Y$, the PL can decide which of $X$ and $Y$ is evaluated first, but, most PLs prefer to refrain from making a decision.

Definition (Collateral evaluation)

Let $X$ and $Y$ be two code fragments (expressions or commands). Then, all of the following are correct implementations of collateral execution of $X$ and $Y$

1. $X$ is executed before $Y$
2. $X$ is executed after $Y$
Expressions + side-effects = evaluation order dilemma

In evaluating, e.g., $X + Y$, the PL can decide which of $X$ and $Y$ is evaluated first, but, most PLs prefer to refrain from making a decision.

Definition (Collateral evaluation)

Let $X$ and $Y$ be two code fragments (expressions or commands). Then, all of the following are correct implementations of collateral execution of $X$ and $Y$

1. $X$ is executed before $Y$
2. $X$ is executed after $Y$

1. “interleaved” execution
Expressions + side-effects = evaluation order dilemma

In evaluating, e.g., $X + Y$, the PL can decide which of $X$ and $Y$ is evaluated first, but, most PLs prefer to refrain from making a decision.

Definition (Collateral evaluation)

Let $X$ and $Y$ be two code fragments (expressions or commands). Then, all of the following are correct implementations of collateral execution of $X$ and $Y$

1. $X$ is executed before $Y$
2. $X$ is executed after $Y$

1. “interleaved” execution
2. simultaneous execution
What will be printed?

So, in compiling and executing

```c
#include <stdio.h>

int main() {
    return printf("Hello,\n")
        - printf("World!\n");
}
```

there is no telling what will be printed!
“Undefined behavior” in PLs

PL designers do not specify everything

- If certain patterns are “bad” programming practice...
- If many “legitimate” implementation make sense...
- If different compilers / different architectures may take a performance toll from over-specification...

Then, the PL designer will tend to consider specifying “undefined behavior”.

```c
messy() {
    int i, n, a[100];
    for (i = 0; i < n; i++)
        printf("a[%d]=%d\n", i, a[i]);
}
```

**C** Specifying that `auto` variables are zero-initialized may cause an unnecessary performance overhead.

**Java** Advances in compiler technology make it possible for the compiler to produce an error message if an uninitialized variable is used.
Side-effects \(\iff\) evaluation order question

- If expressions have side-effects then there is clearly an evaluation order question.

```c
printf("Hello,\n") - printf("World!\n");
```

- But, even if expressions have no side-effects, consider, e.g., the evaluation of the following pure-mathematical expression over \(\mathbb{R}\):

\[
\arcsin 2 + \frac{\sqrt{-1}}{0} \times \log_2 0
\]

whose evaluation tree is

Depending on the evaluation order, each of the red nodes may trigger a runtime error first.
More on expressions’ evaluation order

Example: Rotate 13 Algorithm

Star Wars Episode V: The Empire Strikes Back (1980)

Spoiler

Qnegu Inqre vf Yhxr Fxljnyxre’f sngure.

“ROT13” algorithm: add 13 or subtract 13 from all letters.

Spoiler… Revealed!

Darth Vader is Luke Skywalker’s father.
Rotate 13 implementation

Where’s the bug?

PROGRAM Rotate13(Input, Output);
VAR
  rot13: Array['A'..'z'] of Char;
  c: char;
Procedure fill;
Begin
  For c := 'a' to 'Z' do
    rot13[c] := chr(0);
  For c := 'a' to 'm' do
    rot13[c] := chr(ord(c)+13);
  For c := 'n' to 'z' do
    rot13[c] := chr(ord(c)-13);
  For c := 'A' to 'M' do
    rot13[c] := chr(ord(c)+13);
  For c := 'N' to 'Z' do
    rot13[c] := chr(ord(c)-13);
end;

Procedure Convert(Var c: Char);
Begin
  If c >= 'A'
    and
    c <= 'Z'
    and
    rot13[c] <> chr(0)
    then
      c := rot13[c]
  end;
Begin
  Fill;
  While not eof do begin
    Read(c);
    Convert(c);
    Write(c)
  end
end.

J. Gil (Technion–IIT)
Here is the bug!

Array bounds violation!

Procedure Convert(Var c: Char);
Begin
  If c >= 'A'
    and
    c <= 'Z'
    and
    rot13[c] <> chr(0) (* X *)
  then
    c := rot13[c]
end;

Another annoying (and typical) case...

If p <> null and p^.next <> null then ...
Eager evaluation

Definition (Eager evaluation order)

An eager evaluation order specifies that all arguments to functions are evaluated before the procedure is applied.

- Also called applicative order
- Eager order does not specify in which order the arguments are computed; it can be
  - unspecified (collateral)
  - left to right
  - right to left
  - ...
- Most PLs use eager evaluation for all functions, and the majority of operators
Eager vs. short-circuit logical operators

**Definition (Short-circuit evaluation)**

Short-circuit evaluation of logical operators $\land$ and $\lor$ prescribes that the second argument is evaluated only if the first argument does not suffice to determine the result.

- **Logical “and”** Evaluate the second argument only if the first argument is **true**.
- **Logical “or”** Evaluate the second argument only if the first argument is **false**.

<table>
<thead>
<tr>
<th>PL</th>
<th>Eager version</th>
<th>Short-circuit version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>and, or</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>andalso, orelse</td>
</tr>
<tr>
<td>ML</td>
<td></td>
<td>and, or, and then, or else</td>
</tr>
</tbody>
</table>
Eager vs. short-circuit evaluation

Comparing eager and short-circuit operators:

- Same result,
- Same computation,
Eager vs. short-circuit evaluation

Comparing eager and short-circuit operators:

- Same result, if there are no errors
- Same computation,
Eager vs. short-circuit evaluation

Comparing eager and short-circuit operators:

- Same result, if there are no errors
- Same computation, if there are no side-effects
Eager vs. short-circuit evaluation

Comparing eager and short-circuit operators:

- Same result, if there are no errors
- Same computation, if there are no side-effects

Beautiful programming idiom (originated by Perl, but applicable e.g., in C)

```c
f = fopen(fileName, "r") || die("Cannot open file %s\n", fileName);
```
Using short circuit evaluation in **Bash**

A **Bash** program to remove contents of current directory.

```bash
for f in *; do
    echo -n "$f: 
    [ -e $f ] || echo "already removed"
    [ -d $f ] &&
        echo -n "removing directory" &&
        rmdir $f && ( [ -e $f ] &&
        echo "...failed" || echo ""
    )
    [ -e $f ] &&
        echo -n "moving to /tmp" &&
        mv -f $f /tmp && ( [ -e $f ] &&
        echo "...failed" || echo ""
    )
done
```

- **Bash** commands may **succeed** or fail:
  - Success returns `true` (integer 0)
  - Failure returns `false` (error code ≠ 0)
- “[” is a command; it takes arguments; it may succeed or fail:
  - “[ -e f ]” 3 arguments to command “[”;
    succeeds if file f exists
  - “[ -d f ]” 3 arguments to command “[”;
    succeeds if directory f exists
Emulating short-circuit operators with conditional commands

*poor substitute, operators occur in expressions, and expressions are not commands*
Emulating short-circuit operators with conditional commands

*poor substitute, operators occur in expressions, and expressions are not commands*

Logical “And”

```pascal
If p^ <> null then
  If p^.next <> null
    (* some command *)
```

Logical “Or”

```pascal
If p^ = null then
  (* command, possibly SKIP *)
else if p^.next <> null then
  (* some command *)
```
Normal evaluation order

**Definition (Normal evaluation order)**

In normal evaluation order arguments to functions are evaluated whenever they are used by the function.

- Is a generalization of “short circuit evaluation”
- The terms
  - “normal evaluation order” and
  - “normal order evaluation”
  are synonymous.
- Can be used to encapsulate any of the following C operators in functions:
  - `&&`
  - `||`
  - `,`
  - `? :`
Expressive order of normal order evaluation

Useful (but missing) for time efficient generalization of the “die” programming idiom:

**Definition of function “unless”**

```java
static <T> // exists for each type T
    T unless(boolean b, T t, Exception e) {
        if (b)
            throw e;
        return t;
    }
```

which can then be used to write

**Using function “unless”**

```java
final Integer a = readInteger();
final Integer b = readInteger();
final Integer c = unless(
    b == 0, // always evaluated
    a / b, // evaluated only if b != 0
    new ArithmeticException( // evaluated only if b == 0
        "Dividing \( a \) by \( b \) is invalid!"
    )
);
```
Lazy evaluation order

**Definition (Lazy evaluation order)**

In *lazy evaluation order* arguments to functions are evaluated at the first time they are used by the function.

- Used in **Haskell** (the main feature which distinguishes it from ML)
- Only makes sense in PLs in which there is no "program state"
- Makes it possible to cache results, e.g., the following requires $O(n)$ time to compute the $n^{th}$ Fibonacci number

```haskell
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```
Where are we?

6. Imperative programming

6.1. Commands

6.1.4. Atomic commands
Where are we?

6. Imperative programming

6.1. Commands

6.1.4. Atomic commands
Vanilla assignment command

\[ v \leftarrow e \] (6.3)

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

Multiple

\[ v_1, v_2, \ldots, v_n \leftarrow e \quad (6.4) \]

- Expression \( e \) is evaluated
- Its value is assigned to variables \( v_1, \ldots, v_n \)

Update

\[ v \leftarrow \phi(\cdot, e_1, e_2, \ldots, e_n) \quad (6.5) \]

- syntactic sugar for
- as in COBOL’s
  
  \textbf{Add 1 to a}

- as in C/JAVA
  
  \texttt{i++}

- as in C/JAVA
  
  \texttt{i *= 3}
Two more varieties of the assignment command

Collateral

\[ v_1, v_2 \leftarrow e_1, e_2 \] (6.6)

- \( e_1 \) 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

\[ \langle v_1, v_2 \rangle \leftarrow \langle e_1, e_2 \rangle \] (6.7)

- \( e_1 \) 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 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 **void** aka the **Unit** type.
Where are we?

6. Imperative programming

6.1. Commands

6.1.5. Block commands
Where are we?

6. Imperative programming

6.1. Commands

6.1.5. Block commands
Sequential block constructor

**Definition (Sequential block constructor)**

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

$$\{ C_1; C_2; \ldots; C_n \}$$

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 other PLs; does not match the definition.
Collateral block constructor

Definition (Collateral block constructor)

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

$$\{ C_1 \sim C_2 \sim \cdots \sim C_n \}$$

(6.9)

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

\[
\{ C_1 | C_2 | \cdots | C_n \}\tag{6.10}
\]

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, *programmaticall 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
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
```
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
```
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 cowshed

PROC cowshed()

CHAN INT mammaryGland:

PAR

    calf(mammaryGland?)

    calf(mammaryGland?)

    calf(mammaryGland?)

    calf(mammaryGland?)

    cow(mammaryGland!)

: -- end of PROC cowshed

The calf

PROC calf(CHAN INT nipple?)

    WHILE TRUE

    INT milk:

    SEQ

        nipple ? milk

: -- end of PROC calf
Where are we?

6. Imperative programming

6.1. Commands

6.1.6. Conditional commands
Where are we?

6. Imperative programming

6.1. Commands

6.1.6. 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

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

(6.11)

is a conditional command, whose semantics 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 $E_1$, $E_2$, ..., $E_n$ 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...*
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”
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
CSP: Communicating sequential processes

**Occam** features a concurrent conditional command:

```
INT kisses:
ALT -- a list of guarded commands
  rachel ? kisses
    out ! kisses
  leah ? kisses
    out ! kisses
  bilhah ? kisses
    out ! kisses
  zilpah ? kisses
    out ! kisses
```

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

- *Deep theory of* “communicating sequential processes”
- **ALT** is a 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

\[
\{ E_1?C_1 : E_2?C_2 : \cdots : E_n?C_n : C_{n+1} \} \quad (6.12)
\]

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

\[
E_n = \neg E_1 \land \neg E_2 \land \cdots \land \neg E_{n-1} \quad (6.13)
\]

The “else” clause is sometimes denoted by:

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

Almost all languages use “else”
Variant #1 / many: the “else” clause

Almost all languages use “else”

If thouWiltTakeTheLeftHand
then
  iWillGoToTheRight
else
  iWillGoToTheLeft
Variant #1 / many: the “else” clause

Almost all languages use “else"

\[
\text{If} \ \text{thouWiltTakeTheLeftHand} \\
\text{then} \\
\quad \text{iWillGoToTheRight} \\
\text{else} \\
\quad \text{iWillGoToTheLeft}
\]
Variant #1 / many: the “else” clause

Almost all languages use “else”

If thouWiltTakeTheLeftHand
then
  iWillGoToTheRight
else
  iWillGoToTheLeft

Pascal uses “Otherwise”
case expression of
  Selector:
    Statement;
  …
  Selector:
    Statement
otherwise {"else" instead of "otherwise" is allowed}
  Statement;
  …
  Statement
end (the Gnu-Pascal’s EBNF)

C uses “default”
int isPrime(unsigned c) {
  switch (c) {
    case 0:
    case 1: return 0;
    case 2:
    case 3: return 1;
    default:
      return isPrime(c);
  }
}
Variant #1 / many: the “else” clause

Almost all languages use “else”

Pascal uses “Otherwise”

If thouWiltTakeTheLeftHand
then
  iWillGoToTheRight
else
  iWillGoToTheLeft
Almost all languages use "else"

\[
\text{If thouWiltTakeTheLeftHand then } \begin{align*}
\text{iWillGoToTheRight} \\
\text{else } \text{iWillGoToTheLeft}
\end{align*}
\]

\text{PASCAL uses "Otherwise"}

```
case expression of
  Selector: Statement;
  ...
  Selector: Statement
otherwise
  Statement;
  ...
  Statement
end
```

(\text{the Gnu-PASCAL's EBNF})
Variant #1 / many: the “else” clause

Almost all languages use “else”

If thouWiltTakeTheLeftHand then
   iWillGoToTheRight
else
   iWillGoToTheLeft

PASCAL uses “Otherwise”

case expression of
   Selector: Statement;
   ...;
   Selector: Statement
otherwise
   Statement;
   ...;
   Statement
end

(the Gnu-PASCAL’s EBNF)
Almost all languages use “else”

If thouWiltTakeTheLeftHand then
    iWillGoToTheRight
else
    iWillGoToTheLeft

PASCAL uses “Otherwise”

case expression of
    Selector: Statement;
...
    Selector: Statement
otherwise
    Statement;
...
    Statement
end

(the Gnu-PASCAL’s EBNF)
Variant #1 / many: the “else” clause

Almost all languages use “else”

If thouWiltTakeTheLeftHand then
  iWillGoToTheRight
else
  iWillGoToTheLeft

**Pascal** uses “Otherwise”

```pascal
case expression of
  Selector: Statement;
  ...
  Selector: Statement
otherwise
  Statement;
  ...
  Statement
end
```

(\textit{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 #1 / many: the “else” clause

Almost all languages use “else”

If thouWiltTakeTheLeftHand
  then
    iWillGoToTheRight
else
  iWillGoToTheLeft

PASCAL uses “Otherwise”

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
  \[
  \text{if} \ \text{Condition} \ \text{then} \ \text{Statement} \\
  \quad [ \ \text{else} \ \text{Statement} \ ]
  \]
  
  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 $i = 1, 2, \ldots$

  case Expression of
  
  \[
  \{ \ \text{constantExpression} \ \text{Statement} \ }^{+} \\
  \quad [ \ \text{otherwise} \ \text{Statement} \ ]
  \]
  
  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, probably explain many features of PL.

This is probably the reason 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*

```
IF ( Expression ) l_1, l_2, l_3
```

where

- $l_1$ is the label to go to in case $Expression$ is negative
- $l_2$ is the label to go to in case $Expression$ is zero
- $l_3$ 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

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

where

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

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

- Range of consecutive integer values (in **Pascal**)
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
- Regular expressions in selectors
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in **Bash**
Cases variants?

- Range of consecutive integer values (in Pascal)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of Java after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in Bash
  - Seems natural for the problem domain
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in **Bash**
  - Seem natural for the problem domain
- General patterns in selectors
Cases variants?

- Range of consecutive integer values (in *Pascal*)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of *Java* after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in *Bash*
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in *ML* and other functional PLs
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in **Bash**
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in **ML** and other functional PLs
  - In the spirit of the PL type system
Cases variants?

- Range of consecutive integer values (in Pascal)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of Java after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in Bash
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in ML and other functional PLs
  - In the spirit of the PL type system
- No cases statement
Cases variants?

- Range of consecutive integer values (in **Pascal**)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of **Java** after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in **Bash**
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in **ML** and other functional PLs
  - In the spirit of the PL type system
- No cases statement
  - In **Eiffel** a pure OO language
Cases variants?

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

- Exists in many languages, in the form of a special keyword
Vanilla multi-way conditional?

- Exists in many languages, in the form of a special keyword
- `elseif`, or `elsif` or `ELIF`,
Vanilla multi-way 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? elseif? what’s the big difference?

- There is no big difference!

  else if  many levels of nesting
else if? elseif? what’s the big difference?

- There is no big difference!
  - else if many levels of nesting
  - elseif one nesting level
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!
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 "}".
Where are we?

6. Imperative programming

6.1. Commands

6.1.7. Iterative commands

5. Storage

6. Imperative programming

6.1. Commands

6.1.1. Commands vs. expressions

6.1.2. Recursive definitions

6.1.3. Expressions’ evaluation order

6.1.4. Atomic commands

6.1.5. Block commands

6.1.6. Conditional commands

6.1.7. Iterative commands
Where are we?

6. Imperative programming

6.1. Commands

6.1.7. Iterative commands

5. Storage

6. Imperative programming

6.1. Commands

6.1.1. Commands vs. expressions
6.1.2. Recursive definitions
6.1.3. Expressions’ evaluation order
6.1.4. Atomic commands
6.1.5. Block commands
6.1.6. Conditional commands
6.1.7. Iterative commands
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 \texttt{break} and \texttt{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);
```
State generator? answer #2/5

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
```

Comment WHAT IS BEING COMPUTED???
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

```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
With the “Expression” state generator

**Minimal number of Iterations?**

<table>
<thead>
<tr>
<th>Minimal # Iterations = 0</th>
<th>Minimal # Iterations = 1</th>
</tr>
</thead>
</table>
| `while (s < 100)`
  `s++;`          | `while (s < 100);`       |

**Truth value for maintaining the iteration**

<table>
<thead>
<tr>
<th>Iteration continues with <strong>true</strong></th>
<th>Iteration continues with <strong>false</strong></th>
</tr>
</thead>
</table>
| `While not eof do`
  `Begin`
  `…`
  `end`                     | `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.,

```awk
#!/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/arithmetic 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