6 Commands

Contents [113 frames]
6.1 Expressions vs. commands [10 frames] ............... 1
6.2 Recursive definitions [11 frames] ................. 3
6.3 Atomic commands [4 frames] .................. 5
Exercises ............................................. 5
6.4 Block commands [6 frames] ....................... 5
6.5 Conditional commands [13 frames] ............ 7
6.6 Iterative commands [11 frames] .................. 9
Exercises ........................................... 10
6.7 Structured programming [11 frames] .......... 10
6.8 Sequencers [11 frames] ........................ 12
Exercises ........................................... 14
6.9 Exceptions [35 frames] .......................... 14
6.9.1 Robustness [7 frames] ....................... 14
6.9.2 Policy I: resumption [3 frames] ............ 16
6.9.3 Policy II: error rolling [5 frames] ........... 16
6.9.4 Policy III: setjmp/longjmp of C [6 frames] 17
6.9.5 Policy IV: exceptions [1 frame] ............. 18
6.9.6 Kinds of exceptions [4 frames] ............ 18
6.9.7 Resource acquisition is (or isn’t) initial-
ization [8 frames] ............................... 19
List of tables ....................................... 20
List of figures ..................................... 20
List of equations .................................. 21

1. Commands: a visual mindmap

Figure 6.0.1: Commands: a visual mindmap

6.1 Expressions vs. commands

10 Frames: □ Commands: what are they? □ Commands vs. state-
ments □ Commands vs. expressions □ Expressions changing the
program’s state? □ Expressions without side-effects? □ “Statement-
expressions” in GNU-C □ and in MOCK — □ “Command-
expression” □ Reasonable realizations of command-expression I
□ Reasonable realizations of command-expression II

2. Commands: what are they?

Commands are characteristic of imperative languages

Definition 6.1.1 (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”
• “;”

3. 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 “;”

4. 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 Expressions

1No commands in purely functional languages
5. Expressions changing the program’s state?

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

```pascal
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,…

6. 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

7. “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

8. 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?

9. “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?

10. 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.

11. Reasonable realizations of command-expression II

- The ancient BCPL
- 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
  - …
6.2 Recursive definitions

The set of atomic expressions and the constructors' set are PL dependent, but the variety is not huge.

12. 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 advent of “expression oriented languages”

In Pascal:

Empty Can you figure out where it hides?

Assignment As in the above,

Procedure call As in the above,

15. Three atomic commands in Pascal

(Ignoring goto, the only sequencer of the language)

Empty Can you figure out where it hides?

Assignment As in the above,

Procedure call As in the above,

16. More on Pascal’s atomic commands

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

Definition 6.2.3 (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,

\[ v := E \]  (6.2.1)

is an atomic command.

Definition 6.2.4 (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

\[ p(E_1, \ldots, E_n) \]  (6.2.2)

is an atomic command.

17. 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.

-------------

12. Expressions are recursively defined

13. Function call expression constructor

Definition 6.2.1 (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 6.2.2 (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 \( \tau \).

14. Commands are also recursively defined!

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

Atomic commands 3

• the empty command
• assignment
• “sequencers”4

Command constructors 5

• Block command constructor
• Conditional command constructor
• Iterative command constructor
• “try...catch...finally” command constructor
• “with” command constructor

2Ignoring sequencers
3Each PL is different
4WTF? sequencers will be discussed later
5huge variety
18. Two kinds of atomic commands in C++

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.,

```
0; i++;
```

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.

19. Command expressions in C

Definition 6.2.5 (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>Expression</td>
<td>evaluates to a value</td>
<td>$f() \ ? a + b : a - b$</td>
</tr>
<tr>
<td>Command</td>
<td>change program state (even vacuously)</td>
<td>$i = 0;$</td>
</tr>
<tr>
<td>Variable definition</td>
<td>creates a variable and binds a name to it</td>
<td>$int i;$</td>
</tr>
<tr>
<td>Variable declaration</td>
<td>makes a binding; variable must be created elsewhere</td>
<td>$extern int i;$</td>
</tr>
<tr>
<td>Definition &amp; initializer</td>
<td>creates a variable, binds a name to it, and initializes it</td>
<td>$int i = 3;$</td>
</tr>
<tr>
<td>Declaration &amp; initializer</td>
<td></td>
<td>$x$</td>
</tr>
</tbody>
</table>

Table 6.2.1: C program elements

20. More on atomic expressions in C

All C’s atomic commands (including sequencers) 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?

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

void main() {
    double x, y;
    x = 0.; y = 0.;
    printf("%f, %f\n", x, y);
    ...
}
```

• Not every semicolon makes a command
• Not every lonely semicolon makes an empty command
• Not every expression followed by a semicolon makes a command

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!

21. Two kinds of atomic commands in Java

Just ignoring sequencers

- Not all Java expressions make commands

22. Not all Java expressions make commands

<table>
<thead>
<tr>
<th>Expression</th>
<th>Command</th>
<th>Variable definition</th>
<th>Variable declaration</th>
<th>Definition &amp; initializer</th>
<th>Declaration &amp; initializer</th>
</tr>
</thead>
<tbody>
<tr>
<td>• $; // $</td>
<td>• $i++; // $</td>
<td>• $;++i; // $</td>
<td>• $++i // $</td>
<td>• $i++, j++ // $</td>
<td>• $i++, j++; // $</td>
</tr>
</tbody>
</table>

6.3 Atomic commands

11 Frames: Vanilla assignment command · Two variation of vanilla assignment · Two more varieties of the assignment command · And, what about the “forgotten” atomic commands?
23. Vanilla assignment command

\[ v \leftarrow e \]  
(6.3.1)

- Expression \( e \) is evaluated
- Its value is assigned to variable \( v \)

24. Two variation of vanilla assignment

Multiple

\[ v_1, v_2, \ldots, v_n \leftarrow e \]  
(6.3.2)

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

Update

\[ v \leftarrow \varphi(\cdot, e_1, e_2, \ldots, e_n) \]  
(6.3.3)

- Syntactic sugar for
  
  \[ v \leftarrow \varphi(v, e_1, e_2, \ldots, e_n) \]
- As in COBOL's
  
  Add 1 to a
- As in C/JAVA
  
  \( i++ \)
- As in C/JAVA
  
  \( i *= 3 \)

25. Two more varieties of the assignment command

Collateral

\[ v_1, v_2 \leftarrow e_1, e_2 \]  
(6.3.4)

- \( e_1 \) is evaluated and assigned to \( v_1 \)
- \( e_2 \) is evaluated and assigned to \( v_2 \)
- The two actions take place simultaneously
- Cannot be used for swapping contents of variables
- Theoretically possible, but not very useful

Simultaneous

\[ \langle v_1, v_2 \rangle \leftarrow \langle e_1, e_2 \rangle \]  
(6.3.5)

- \( e_1 \) 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 simultaneously
- Can be used for swapping
- We had tuples of values; \( (v_1, v_2) \) can be thought of as a tuple of variables; simultaneous assignment can be thought of as “tuple assignment”

26. And, what about the “forgotten” atomic commands?

The SKIP command aka NOP, aka \( \texttt{\relax} \), 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 \texttt{Unit} type.

Exercises

1. What does this program do? What kinds of assignments do we find in it?

\[
\text{Algol}
\begin{align*}
x, y & \leftarrow 1; \\
to n \ do (if \ x < y \ then \ x else y) & \leftarrow x + y; \\
x & \leftarrow \max(x, y);
\end{align*}
\]

2. What does this program do? What kinds of assignments do we find in it?

\[
\text{Algol}
\begin{align*}
x, y & \leftarrow 1; \\
to n \ do \begin{align*} \\
x, y & \leftarrow y, x; \\
x & \leftarrow x \times y \ end;
\end{align*}
\end{align*}
\]

6.4 Block commands

4 Frames:

- Vanilla assignment command
- Two variation of vanilla assignment
- Two more varieties of the assignment command
- And, what about the “forgotten” atomic commands?

Contents

27. Sequential block constructor

Definition 6.4.1 (Sequential block constructor). If \( C_1, \ldots, C_n \) are commands, \( n \geq 0 \), then

\[ \{C_1; C_2; \ldots; C_n\} \]  
(6.4.1)

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 other PLs; does not match the above definition.

28. Collateral block constructor

Definition 6.4.2 (Collateral block constructor). If \( C_1, \ldots, C_n \) are commands, \( n \geq 0 \), then

\[
\{ C_1 \sim C_2 \sim \cdots \sim C_n \} \tag{6.4.2}
\]

is a composite command, whose semantics is that \( C_1, \ldots, C_n \) are executed collateral.

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

29. 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

\[
\begin{align*}
\text{grandsons} & \leftarrow 7; \\
\text{grandsons} & \leftarrow 4; \\
\text{grandsons} & \leftarrow 5;
\end{align*}
\]

Semantically Equivalent

\[
\begin{align*}
\text{humanity}.add("Adam"); \\
\text{humanity}.add("Eve");
\end{align*}
\]

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

30. Concurrent block constructor

Definition 6.4.3 (Concurrent block constructor). If \( C_1, \ldots, C_n \) are commands, \( n \geq 0 \), then

\[
\{ C_1 | C_2 | \cdots | C_n \} \tag{6.4.3}
\]

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

31. 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...”.

32. Concurrent execution in Occam

### The cow

```occam
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

```occam
PROC calf(CHAN INT nipple?)
    WHILE TRUE
        INT milk:
        SEQ
            nipple ? milk
            milk := milk + 1
    : -- end of PROC calf
```

### The cowshed

```occam
PROC cowshed()
    CHAN INT mammaryGland:
    PAR
        calf(mammaryGland?)
        calf(mammaryGland?)
        calf(mammaryGland?)
        calf(mammaryGland?)
        cow(mammaryGland!)
    : -- end of PROC cowshed
```
6.5 Conditional commands

Definition 6.5.1 (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.

34. Semantics of conditional commands

Semantics of

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

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 \( E_1, E_2, \ldots, E_n \) collateral. 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:

In any infinite run, there is no process which does not execute infinitely many times.

35. CSP: Communicating sequential processes

OCCAM features a concurrent conditional command:

Jacob and his four wives

| 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...

36. The "else" variants

Definition 6.5.2 (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

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

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

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

The "else" clause is sometimes denoted by:

- default
- otherwise

Almost all languages use "else"

If thouWiltTakeTheLeftHand then  
  iWillGoToTheLeft
else  
  iWillGoToTheRight

PASCAL uses “Otherwise”

\[
\text{case expression of Selector: Statement; … Selector: Statement otherwise Statement; … Statement and }
\]

(the Gnu-PASCAL’s EBNF) C uses “default”

37. Variant #1 / many: the "else" clause

38. Variant #2 + #3 / many: if-then-else & cases

- Special construct for the case \( n = 1 \) in the form of

\[
\text{if Condition then Statement [ else Statement ]}
\]

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

39. Variant #1 / many: the "else" clause

40. Variant #2 + #3 / many: if-then-else & cases
A selector of Pascal’s `case` statement may contain:

- Multiple entries
- Range entries

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* $n = 1$

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 if case `Expression` evaluates to 1
- $l_2$ is the label to go to if case `Expression` evaluates to 2
- $l_3$ is the label to go to if case `Expression` evaluates to 3

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

Early versions of FORTRAN had a “computed goto” instruction

```
GO TO ( $l_1, l_2, \ldots, l_n$ ) `Expression`
```

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
- $\ell_n$ is the label to go to in case `Expression` evaluates to $n$

```
elseif
```

What’s the big difference?

- There is no big difference!
- `elseif` many levels of nesting
- `elsif` 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 “}”.

```
elsif ( $a > b$ ) {
  echo "a is bigger than b";
} elsif ( $a == b$ ) {
  echo "a is equal to b";
} else {
  echo "a is smaller than b";
}
```
6.6 Iterative commands

11 Frames: □ Iterative command constructor □ State generator? answer #1/5 □ State generator? answer #2/5 □ State generator? answer #3/5 □ State generator? answer #4/5 □ State generator? answer #5/5 □ Minor varieties of iterative commands □ The iteration variable □ Subtleties of the iteration variable □ □ Definite vs. Indefinite Iteration □ So, let’s make our guesses...

Contents

46. Iterative command constructor

A very general pattern of iterative command constructor

Definition 6.6.1 (Iterative command constructor). If $S$ is a “program state generator” and $C$ is a command, then $forall\ S do\ C$ 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!

47. 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);
```

48. State generator? answer #2/5

The state generator $S$ may be... Any arithmetical progression, e.g., in FORTRAN

```
Comment WHAT IS BEING COMPUTED??
INTEGER SQUARE11
SQUARE11=0
DO 1000 I = 1, 22, 2
  SQUARE11 = SQUARE11 + I
1000 CONTINUE
```

49. 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,

50. 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);
```

51. State generator? answer #5/5

The state generator $S$ may be... Cells in an array, e.g., in JAVA

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

52. Minor varieties of iterative commands

Minimal number of iterations?

<table>
<thead>
<tr>
<th>Minimal # Iterations = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (s &lt; 100)</td>
</tr>
<tr>
<td>z++;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimal # Iterations = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>do {</td>
</tr>
<tr>
<td>z++;</td>
</tr>
<tr>
<td>} while (s &lt; 100);</td>
</tr>
</tbody>
</table>

Truth value for maintaining the iteration

Iteration continues with true

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

Iteration continues with false

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

none of these is too interesting...

53. The iteration variable

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

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

```
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);
```

54. 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?

55. 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
```

---

56. 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.

Exercises

1. Revisit the example of using references in ML. Is r an L-value? An R-value? Both? None of these? Explain.

2. AWK designers chose the iteration variable to range over the indices of an array, instead of the values. Why was this the only sensible decision?

3. What’s the iteration variable of a while loop?

4. What does the acronym “CSP” stand for?

5. Write the most feature-rich class in Java, without using the semicolon character, “;”, even once.

6. How come C does not offer any rules regarding the iteration variable?

7. How come JAVA, despite being similar to C, recognize the notion of and iteration variable.

8. Explain why it is impossible to use the PERL die(...) programming idiom in JAVA.

9. JAVA designers chose the iteration variable to range over the array cells, instead of the indices. Why was this the only sensible decision?

10. Revisit the example of using references in ML. Is !r an L-value? An R-value? Both? None of these? Explain.


12. Write a C function with at least three commands in it, without using the semicolon character, “;”, even once.

13. What are the circumstances in which

  ```pascal
  if false then
    writeln(true < false);
  ```

  prints true; concretely, add some Pascal code around to make this happen.

---

6.7 Structured programming

11 Frames: □ What are sequencers? □ Labels □ Label “literal” & first class labels □ Declared vs. ad-hoc labels □ Ad-hoc labels may generate subtle bugs □ The persecuted goto □ Goto to a nesting function □ Problems of structured programming □ Escapes □ Varieties of escape □ Continue

Contents

6.9.1 Robustness . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
6.9.2 Policy I: resumption . . . . . . . . . . . . . . . . . . . . . 16
6.9.3 Policy II: error rolling . . . . . . . . . . . . . . . . . . . . 16
6.9.4 Policy III: setjmp/longjmp of C . . . . . . . . . . . . . 17
6.9.5 Policy IV: exceptions . . . . . . . . . . . . . . . . . . . . 18
6.9.6 Kinds of exceptions . . . . . . . . . . . . . . . . . . . . . 18
6.9.7 Resource acquisition is (or isn’t) initialization 19

---

57. Flowcharts: another method for command composition

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

- Edges: goto

Nodes are in fact atomic commands
60. Challenge of understanding spaghetti code

- The program on the right does something useful!
- Many intersecting spaghetti edges
- No obvious meaningful partitioning of the chart
- Only a few nodes with one entry and one exit
- All decision nodes have two (or more) outgoing edges
- Some nodes with two incoming edges, even three!

61. Structured programming

...is a programming paradigm characterized by

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

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

Theorem 6.7.1 (The structured programming theorem (Böhm-Jacopini, 1966)). Every flowchart graph \( G \), can be converted into an equivalent structured program, \( P(G) \).

62. Nassi-Shneiderman diagram

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

63. Compound commands in Nassi-Shneiderman diagrams

- Compound commands are rectangles which have smaller rectangles in them
- Each rectangle may contain in it one, two, or more rectangles
- Correspond to our familiar command constructors
- Color is not part of the diagram
- But we can add it anyway...

64. Matrix multiplication with Nassi-Shneiderman diagram

65. Factorial with Nassi-Shneiderman diagram

66. More Nassi-Shneiderman notations

67. 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...

6.8 Sequencers

7 Frames: Resumption Pros & cons of resumption Emulating resumption in Heron’s program
68. What are sequencers?

**Definition 6.8.1** (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

69. Labels

To denote where **goto** will go to, one needs a **label**

**Definition 6.8.2** (Label). A **label** is 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

Labels are deliberately disprivileged entities.

70. 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.

```
11: label v = a > b?
    11: 12;
    
12: goto v;
```

71. 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++

```
int main() {
    http://www.cs.technion.ac.il/yogi;
    printf("Page loaded successfully\n");
}
```

72. Ad-hoc labels may generate subtle bugs

```
int isPerect(unsigned n) {
    switch (n) {
    default:
        return 0;
    case 6:
    case 28:
    case 496:
    case 8128:
    case 0x1FFF000u:  
        return 1;
    } 
```

spelling error in “default”

73. 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!

74. 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.

75. 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 some, a problem of a different kind was found then begin
                do something about it
                end else begin
                continue to work
                end
            end
        end
end
```
76. Escapes

Definition 6.8.3 (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

77. Varieties of escape

- Escape any enclosing Loop. **exit** \( \ell \) in ADA and **break** \( \ell \) in PERL/JAVA, where \( \ell \) 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

78. Continue

Definition 6.8.4 (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**.

Exercises

1. How can you emulate **break** using **goto** in PASCAL?
2. Give an example in which **continue** can be used to simplify nesting structure.
3. Why did PASCAL’s designer decide to identify labels with integers.
4. Give an example in which **return** can be used to simplify nesting structure.
5. Why did PASCAL’s designer decide to require that labels are pre-declared.
6. Give an example in which **break** out of a loop can be used to simplify nesting structure.
7. **C** does not make it possible to **break** out of more than one loop. How can you do something similar with **return** and auxiliary functions?
8. Compare stored labels with continuations. What’s common and how are they different?

6.9 Exceptions

5 Frames

- How does it work?
- What’s stored in a jump buffer?
- Step I: low-level functions call **longjmp**
- More low-level functions calling **longjmp**
- Step II: catching the **longjmp**
- Long jump in C & PASCAL

79. Sometimes... it does not make sense to proceed as usual!

Two kinds of abnormal and unusual situations

**Bugs** An error in the program code, makes it try to execute an invalid instruction.

- division by zero
- array bounds overflow
- stack overflow
- dereferencing null pointer
- runtime type error
- (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

6.9.1 Robustness

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.

81. Robust PL vs. robust programs

Definition 6.9.1 (Robust PL). The PL’s runtime system recovers gracefully from all program bugs

“the programmer cannot make the runtime system crash”

Definition 6.9.2 (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 6.9.3 (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.

83. 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.

84. Corollaries of the robust programs theorem

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

Corollary 6.9.4 (Adequate support for exception handling). No PL provides adequate support for exception handling.

Humanity concentrates in what it does well

Corollary 6.9.5 (Graceful termination). Many PLs offer graceful termination in case of bugs.

Humanity cannot do what it does not know how to do

Corollary 6.9.6 (Rarestness of robust programs). Very few programs are truly robust

And since programmers are lazy,

Corollary 6.9.7 (Input errors considered bugs). Most programs convert input errors into bugs

85. Example: Heron’s formula for the area of the triangle

Figure 6.9.1: Hero of Alexandria (10–70 AD); also known as Heron

Figure 6.9.2: A triangle with edges \(a\), \(b\), and \(c\)

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

\[
s = \frac{a+b+c}{2}
\]

Let’s start implementing it. First, essential include files

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

Then, a function to prompt for and read real numbers:

```c
double get(const char *s) {
    double r;
    printf("Please enter \%s: \n", s);
    scanf("%lg", &r);
    return r;
}
```
Invalid input? End of file? We do not care, so let’s carry on…

Square root of a negative number? We don’t care! Proceeding…

Is either of \( s - a \), \( s - b \), or \( s - c \) negative? We proceed…

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.

---

### 6.9.2 Policy I: resumption

**4 Frames:**
- The multiple resource allocation problem
- Resource acquisition is initialization (RAII)
- RAI example: a simple file reader class
- RAI example: a simple file writer class
- RAI example: using the reader & writer classes
- Life without RAII is tough!
- Automatic deallocation: new JAVA 7 feature

**Motivating example:** memory exhausted exception

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

**Corrective action?**
- free some non-essential memory
- invoke the garbage collector
but, these efforts are not guaranteed to be sufficient

---

### 6.9.3 Policy II: error rolling

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

**88. Pros & cons of resumption**

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

**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

---

### 89. Emulating resumption in Heron’s program

**Original: naive read**

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

**Retrying the Read**

```cpp
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;
}
```

---

### 90. Explicit error handling: returning error code

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

---

### 91. More error code returned

**Step I:** another low-level function…

---

**Issues:**

- Unexpected result in case of invalid input
- Repeated unpredictable result
- 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!
92. Error handling

Step II: dealing with the propagated errors

```c
int main() {
    double a = get("A");
    double b = get("B");
    double c = get("C");
    double area;
    if (a < 0 || b < 0 || c < 0) {
        fprintf(stderr, "Bad input");
        return 1;
    }
    area = A(a, b, c);
    if (area < 0) {
        fprintf(stderr, "Bad triangle");
        return 1;
    }
    printf("Area is: %g\n", area);
    return 0;
}
```

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?

93. 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

94. 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

6.9.4 Policy III: setjmp/longjmp of C

95. How does it work?

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

static jmp_buf b;

void g(void) {
    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;
}
```

96. 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
#include <stdio.h>

typedef struct jmp_buf jmp_buf[1];
```

97. Step I: low-level functions call longjmp

```c
double get(const char *s) {
    double r;
    int n;
    printf("Please enter %s: ", s);
    n = scanf("%lg", &r);
    if (n < 0) {
        perror(s);
        longjmp(b, 1); // exception code
    }
    if (r < 0) {
        perror(s);
        longjmp(b, 1); // exception code
    }
    return r;
}
```
98. More low-level functions calling longjmp

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?

99. Step II: catching the longjmp

void v(double v, const char *error) {
    if (v <= 0) throw error;
}

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);
}

100. 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

6.9.5 Policy IV: exceptions

101. Exception language construct

6.9.6 Kinds of exceptions

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.

102. Summary: policies for exception handling

What can be thrown?

Pascal The Unit type: goto to a nesting block has no associated value
Consider an application that needs to open two files, f₁ and f₂. If it opens f₁ and discovers an error in f₂ is must remember to close f₁ 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");
            }
        }
    } catch (IOException e) {
        System.out.println(f1 + ":I/O error");
    }
}
```

7. 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 model) Variables allocated on the stack are reclaimed.

C++ Variables allocated on the stack are destroyed.

This occurs in two cases:

- The block terminates normally
- An exception is thrown out of the block.

Definition 6.9.8 (RAII). A common programming idiom, by which resources are allocated by the constructor and deallocated by the destructor.

106. The multiple resource allocation problem

Consider an application that needs to open two files, f₁ and f₂. If it opens f₁ and discovers an error in f₂ is must remember to close f₁ 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");
            }
        }
    } catch (IOException e) {
        System.out.println(f1 + ":I/O error");
    }
}
```

107. 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 destroyed.

This occurs in two cases:

- The block terminates normally
- An exception is thrown out of the block.

Definition 6.9.8 (RAII). A common programming idiom, by which resources are allocated by the constructor and deallocated by the destructor.

108. RAII example: a simple file reader class

```
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 destroyed.

109. RAII example: a simple file writer class

```
```
The file is opened when a `Writer` object is created; the file is closed when this object is destructed.

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;

Robust copying of a file

Java has no destructors, so RAII is difficult

Tough, and error prone, because we do not know the file’s state when the exception is caught.

Robust “Hello, World” (version I)

Tough, and error prone, because we do not know the file’s state when the exception is caught.

List of tables

6.2.1 C program elements

List of figures

6.0.1 Commands: a visual mindmap
6.1 An intriguing finite automaton computing (in a non-sensible manner) a function that does make sense
6.2 Flowchart for “getting things done (GTD)”
6.3 Challenge of understanding spaghetti code
6.4 Compound commands in Nassi-Shneiderman diagrams
6.5 Matrix multiplication with Nassi-Shneiderman diagram
6.6 Factorial with Nassi-Shneiderman diagram
6.7 More Nassi-Shneiderman notations
6.8 Even more Nassi-Shneiderman notations
6.9.1 Robustness

6.9.1 Hero of Alexandria (10–70 AD); also known as Heron
6.9.2 A triangle with edges $a$, $b$, and $c$
List of equations

(6.2.1) Assignment atomic command of PASCAL .... 3
\[ v := E \]

(6.2.2) Procedure call atomic command of PASCAL . 3
\[ p(E_1, \ldots, E_n) \]

(6.3.1) Vanilla assignment ....................... 5
\[ v \leftarrow e \]

(6.3.2) Multiple assignment ....................... 5
\[ v_1, v_2, \ldots, v_n \leftarrow e \]

(6.3.3) Update assignment ....................... 5
\[ v \leftarrow \varphi(\cdot, e_1, e_2, \ldots, e_n) \]

(6.3.4) Collateral assignment ..................... 5
\[ v_1, v_2 \leftarrow e_1, e_2 \]

(6.3.5) Simultaneous assignment ................... 5
\[ \langle v_1, v_2 \rangle \leftarrow \langle e_1, e_2 \rangle \]

(6.4.1) Sequential block command constructor ...... 5
\[ \{C_1; C_2; \ldots; C_n\} \]

(6.4.2) Collateral block command constructor ...... 6
\[ \{C_1 \sim C_2 \sim \cdots \sim C_n\} \]

(6.4.3) Concurrent block command constructor ...... 6
\[ \{C_1 | C_2 | \cdots | C_n\} \]

(6.5.1) Conditional command constructor .......... 7
\[ \{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n\} \]

(6.5.2) Conditional command constructor with else clause ................. 7
\[ \{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n : C_{n+1}\} \]

(6.5.3) A boolean expression for the “else” clause ... 7
\[ E_n = \neg E_1 \land \neg E_2 \land \cdots \land \neg E_{n-1} \]

(6.5.4) Computer goto in FORTRAN ................. 8
GO TO (\ell_1, \ell_2, \ldots, \ell_n) Expression

6.9.1 Robustness

(6.9.1) Heron’s formula for the area of the triangle 15
\[ A = \sqrt{s(s-a)(s-b)(s-c)} \]
\[ s = \frac{a+b+c}{2} \]