LINFO1104 – LSINC1104
Concepts, paradigms, and semantics of programming languages

Lecture 1
Basic concepts, functions, and invariants

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Context of this course

- LEPL1402 – LSINC1402: First semester programming course
  - Introduction to programming based on Java and IntelliJ
  - Java and object-oriented programming
  - Algorithms, data structures, and invariants
  - Complexity and software engineering
  - Introduction to functions and concurrency in Java
- LINFO1104 – LSINC1104: Second semester course
  - Programming paradigms based on Oz multiparadigm language
  - Formal semantics including lambda calculus
  - Procedural abstraction (higher-order programming)
  - Data abstraction (objects and abstract data types)
  - Symbolic programming
  - Concurrent programming
  - Multi-agent programming and Erlang

2

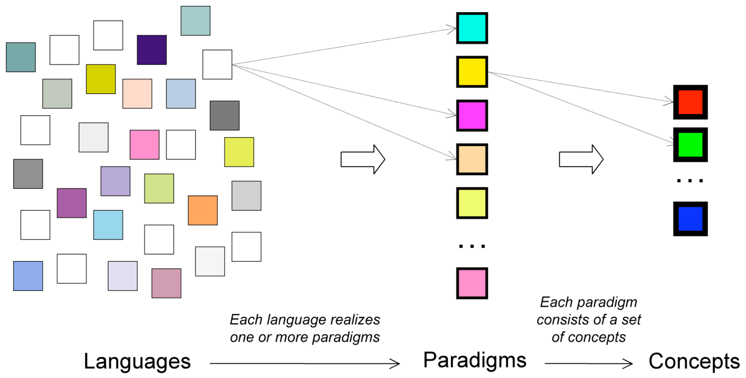
Programming paradigms

3

Hundreds of programming languages are in use...

4

So many, how can we understand them all?



- Key insight: languages are based on paradigms, and there are many fewer paradigms than languages
- We can understand *many* languages by learning *few* paradigms!

5

What is a paradigm?

- A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathematical theory
- A program is written to solve problems
  - Any realistic program needs to solve different kinds of problems
  - Each kind of problem needs its own paradigm
  - So we need multiple paradigms and we need to combine them in the same program

6

How can we study *multiple* paradigms?

- How can we study multiple paradigms without studying multiple languages (since most languages only support one, or sometimes two paradigms)?
- Each language has its own syntax, its own semantics, its own system, and its own quirks
  - Picking many languages, like Java, Scala, Erlang, Scheme, and Haskell, and structuring our course around them would be complicated
- Our pragmatic solution: two languages, Oz (a multiparadigm research language) and Erlang (a multi-agent industrial language)
  - Our textbook, *Concepts, Techniques, and Models of Computer Programming*, uses Oz to cover many paradigms

7

How can we *combine* paradigms in a program?

- Each paradigm is a different way of thinking
  - How can we combine different ways of thinking in one program?
- We can do it using the concept of a kernel language
  - Each paradigm has a simple core language, its kernel language, that contains its essential concepts
    - Every practical language, even complicated ones, can be translated easily into its kernel language
  - Even very different paradigms have kernel languages that have much in common; often there is only one concept difference

- We start with a simple kernel language that underlies our first paradigm, functional programming
  - We then add concepts one by one to give the other paradigms
  - Scientific method: understand a system in terms of its parts

8

Summary of the approach

- Hundreds of languages are used in practice: we cannot study them all in one course or in one lifetime
  - Solution: **focus on paradigms**, since each language is based on a paradigm and there there are many fewer paradigms than languages
- One language per paradigm is too much to study in a course, since each language is already complicated by itself
  - Solution: **use one research language**, Oz, that can express many paradigms (plus Erlang for multi-agent programming!)
- Realistic programs need to combine paradigms, but how can we do it since each paradigm is a different way of thinking?
  - Solution: **define paradigms using kernel languages**, since different paradigms have kernel languages that are almost the same
  - Kernel languages allow us to define many paradigms by focusing on their differences, which is much more economical in time and effort

9

Let's get started

- You should already know an object-oriented language!
  - Object-oriented programming, as used in Java, is clearly an important paradigm (as seen in LEPL1402)
  - But what about the other paradigms?
- Isn't object-oriented programming by far the most important and useful paradigm?
  - Actually, no, it's not!
  - Many other paradigms are extremely useful, often more so than OOP! For example, to make robust and efficient distributed programs on the Internet, OOP does not solve the right problems. Multi-agent programming is much better for that, which is why we will give an introduction to Erlang.
  - LINFO1104 covers five paradigms that solve many problems

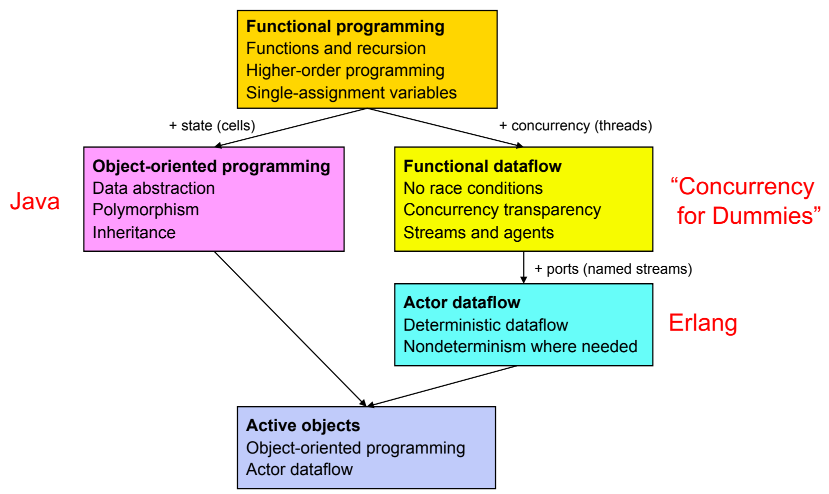
10

Five paradigms

- LINFO1104/LSINC1104 covers five paradigms:
  - Functional programming
  - Object-oriented programming (used in Java)
  - Functional dataflow programming
  - Actor dataflow programming (multi-agent) (used in Erlang)
  - Active objects
- These are probably the most important programming paradigms for general use
  - But there are many other paradigms, made for other problems: LINFO1104/LSINC1104 gives you a good foundation for studying them later if you wish

11

Five paradigms



12

Our first paradigm

- Functional programming
  - It is the simplest paradigm
  - It is the foundation of all other paradigms
  - It is a form of *declarative programming*: say what, not how
- Functional programming is the right way to start
  - It is our first introduction to programming concepts
  - It is our first introduction to a kernel language
  - We use it to explain invariant programming (how to do loops)
  - We use it to explain symbolic programming
  - We use it to explain higher-order programming
  - We give a formal semantics based on the kernel language
- Everything we learn in functional programming stays true later!

13

Practical organization

14

Course organization LINFO1104

- Weekly lecture
  - Tuesday 14h00-16h00 (A.02 SCES)
- Weekly lab session
  - Tuesday 16h15-18h15 (MERC 11)
  - Wednesday 8h30-10h30 (MERC 02)
  - Wednesday 10h45-12h45 (MERC 02)
  - Wednesday 14h00-16h00 (MERC 12)
  - Monday 14h00-16h00 (MERC 02, starting S2)
- Midterm exam (around week 7)
  - Optional, counts for 5 points on final grade (max. with 1st question of final exam)
- Course project (second half of semester)
  - Practical programming project in groups of two students
  - Mandatory, counts for 5 points on final grade
- Final exam (15 points; 5 points correspond to midterm)
- Bonus (1 point extra if you attend all lab sessions)

15

Course organization LSINC1104

- Weekly lecture
  - Thursday 10h45-12h45 (E6K Salle 12)
- Weekly lab session
  - Thursday 13h45-15h45 (E6K Salle 12)
- Midterm exam
  - Optional, counts for 5 points on final grade
- Course project
  - Practical programming project in groups of 2 students
  - Mandatory, counts for 5 points on final grade
- Final exam (15 points, 5 points correspond to midterm)
- Bonus (1 point extra if you attend all lab sessions)

16

Software support

- LINFO1104 Moodle
  - All practical information is given here
  - Register yourself!
- Mozart Programming System
  - For practical exercises; www.mozart2.org
  - Download and install Mozart 2.0.1
- Erlang OTP
  - For multi-agent programming; www.erlang.org
  - Download and install OTP 27.2 (in second half of course)

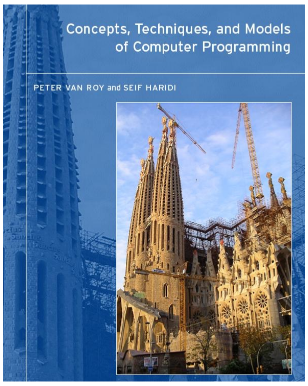
17

Educational team

- Professor
  - Peter Van Roy (peter.vanroy@uclouvain.be)
- Teaching assistants
  - Donatien Schmitz (donatien.schmitz@uclouvain.be)
  - Alix Temmerman (alix.temmerman@uclouvain.be)
  - Emilie Deprez (emilie.deprez@uclouvain.be)
- Tutors
  - Sébastien Amelinckx (sebastien.amelinckx@student.uclouvain.be)
  - Nicolas Daube (nicolas.daube@student.uclouvain.be)
  - Simon Defrenne (simon.defrenne@student.uclouvain.be)
  - Sacha Defrère (sache.defrere@student.uclouvain.be)
  - Ernesto Mendoza Gomez (ernesto.mendoza@student.uclouvain.be)
  - Charles Packo (charles.packo@student.uclouvain.be)
  - Robert Séverin (severin.robert@student.uclouvain.be)
  - Mateo-Francisco Willaime (mateo-francisco.willaime@student.uclouvain.be)

18

Course textbook and slides



- "Concepts, Techniques, and Models of Computer Programming" by Peter Van Roy and Seif Haridi, MIT Press
  - Copies available to borrow in the BST
- Slides for each week's lecture
  - The slides contain most of the course material, but please look at the book if anything is unclear or if you want more examples
  - Some material will be given that is not in the book, in particular regarding semantics (lambda calculus), for that part you should refer to the slides
  - It is up to you to read the book if anything is unclear! (or ask your tutor, a teaching assistant, or me)

19

Basic concepts

20

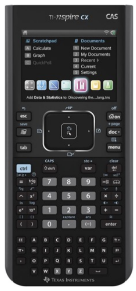
Mozart Programming System: interactive Emacs interface

declare

X = 1234 * 5678

{Browse X}

- Select a region in the Emacs buffer
- Feed the region to the system
  - The text is compiled and executed
- Interactive system can be used as a powerful calculator



21

Creating variables

declare

X = 1234 * 5678

{Browse X}

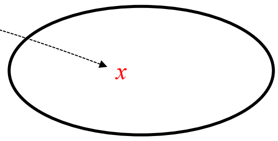
- **Declare** (create) a variable **designated** by X
- **Assign** to the variable the value 7006652
  - Result of the calculation 1234*5678
- **Call** the procedure Browse with the argument designated by X
  - Opens a window that displays 7006652

22

Identifiers and variables

Program text *System memory*

declare X
X=11*11
{Browse X}



- There are two concepts hiding in plain view here
  - Identifier X : what you type (character sequence starting with uppercase)
    Var, A, X123, FirstCapitalBank
  - Variable x : what is in memory (used to store the value)
- Variables are short-cuts for values (= constants)
  - Can only be assigned to one value (like mathematical variables)
  - Multiple assignment is another concept! We will see it later in the course.
  - The type of the variable is only known when it is assigned (dynamic typing)

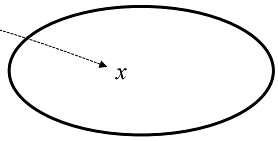
23

Environment

Program text *System memory*

declare X
X=11*11
{Browse X}

E



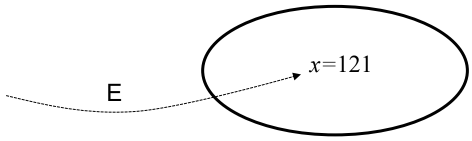
- **declare** is an interactive instruction
  - Creates a new variable in memory
  - Links the identifier and its corresponding variable
- Third concept: environment E={X→ x}
  - A function that takes an identifier and returns a variable: E(X) = x
  - Links identifiers and their corresponding variables
    (with the values they are bound to)

24

Assignment

Program text *System memory*

declare X
X=11*11
{Browse X}



- The assignment instruction X=121 binds the variable x to the value 121

25

Single assignment

- A variable can only be bound to one value
  - It is called a single-assignment variable
  - Why? Because we are doing functional programming!
- Incompatible assignment: signals an error
  X = 122
- Compatible assignment: accepted
  X = 121

26

Why single assignment?

- Single assignment is part of functional programming
  - It means that variables are *mathematical variables*, like in an equation
- Programming with mathematical variables is easy
  - It may seem strange for Java programmers, but it is actually quite easy
- Advantages
  - Programs are much easier to prove, analyze, test, and debug
- Disadvantages
  - Functional programming is a different way of thinking and takes some getting used to
  - Functional programming is a programming paradigm, and each programming paradigm is best for certain kinds of problems
    - You will understand when we start using other paradigms!

27

The functional style can be done in all languages

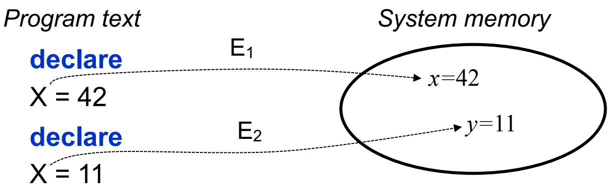
- "A program that works today will work tomorrow"
  - Functions and variables don't change
  - All changes are in the arguments, not in the functions
- This programming style is encouraged in all languages (incl. Java)
  - "Stateless server" for a client/server application
  - "Stateless component" for a service application

"Cookies" on the Web

- Learning functional programming helps us think in this style
  - As well as helping us understand all programming languages, today's and tomorrow's

28

Redeclaring an identifier



- An identifier can be redeclared
  - The same identifier refers to a different value
  - There is no conflict with single assignment.
    Each occurrence of X corresponds to a different variable.
- The interactive environment always has the last declaration
  - **declare** keeps the same correspondance until redeclared (if ever)
  - In this example X will refer to 11

29

Scope of an identifier occurrence

```
local
   X
in
   X = 42  {Browse X}
   local
      X
   in
      X = 11  {Browse X}
   end
   {Browse X}
end
```

- The instruction
  local X **in** <stmt> **end**
  declares X between **in** and **end**
- The scope of an identifier occurrence is that part of the program text for which the occurrence corresponds to the same variable declaration
- The scope can be determined by inspecting the program text; no execution is needed.
  This is called lexical scoping or static scoping.
- Why is there no conflict between X=42 and X=11, even though variables are single assignment?
- What will the third Browse display?

30

Scope of an identifier occurrence

Scope of X
(red declaration)

Scope of X
(green declaration)

```
local
   X
in
   X = 42  {Browse X}
   local
      X
   in
      X = 11  {Browse X}
   end
   {Browse X}
end
```

31

Tips on Oz syntax

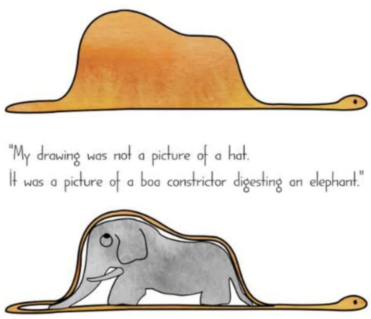
32

Tips on Oz syntax

- You can see that Oz syntax is not like Java syntax
- The most popular syntax in mainstream languages (Java and C++) is « C-like », where identifiers are statically typed « int i; » and start with lowercase, and code blocks are delimited by braces { … }
  - Oz syntax is definitely not C-like!
- Oz syntax is designed for multiparadigm programming
  - Oz syntax is inspired by many languages: Prolog (logic programming), Scheme and ML (functional programming), C++ and Smalltalk (object-oriented programming), and so forth
- (Later on we will see Erlang, with even another syntax)

33

Syntax is just the surface



"It is only with the heart that one can see rightly; what is essential is invisible to the eye."

"On ne voit bien qu'avec le cœur. L'essentiel est invisible pour les yeux."

– The Little Prince, Antoine de Saint-Éxupéry

- Programmers are too much in love with their favorite syntax
  - Syntax is just the surface. Programmers must learn to look beyond syntax to the true reality, which is semantics.
  - In this course we will mainly use Oz syntax, but the semantics you learn applies to all the languages you will use in your whole career
  - Programmers should be polyglot and move easily from one syntax to another
    - I hope that this course helps you achieve that ability!

34

Why is Oz syntax different?

- It is different because Oz supports many programming paradigms
  - The syntax is carefully designed so that the paradigms don't interfere with each other
  - It's possible to program in just one paradigm. It's also possible to program in several paradigms that are cleanly separated in the program text.
- So it is important not to get confused by the differences between Oz syntax and other syntaxes you may know
- We show the main differences so that you will not be hindered by them

35

Four main differences in Oz syntax

- Identifiers in Oz always start with an uppercase letter
  - Examples: X, Y, Z, Min, Max, Sum, IntToFloat.
  - Why? Because lowercase is used for symbolic constants (atoms).
- Procedure and function calls in Oz are surrounded by braces { ... }
  - Examples: {Max 1 2}, {SumDigits 999}, {Fold L F U}.
  - Why? Because parentheses are used for record data structures.
- Local identifiers are introduced by **local** ... **end**
  - Example: **local** X **in** X=10+20 {Browse X} **end**.
  - Why? Because all compound instructions in Oz start with a keyword (such as « **local** ») and terminate with **end**.
- Variables in Oz are single assignment
  - Examples: **local** X Y **in** X=10 Y=X+20 {Browse Y} **end**.
  - Why? Because the first paradigm is functional programming. Multiple assignment is a concept that we will introduce later.

36

Oz syntax in the programming exercises

- Most programming bugs, at least early on, are due to syntax errors
  - Most common error: lowercase letter to start an identifier
- Please take into account the four main differences. Once you have assimilated them, reading and writing Oz will become straightforward.
- And now let’s introduce functions

37

Functional programming

38

Functions

- We would like to execute the same code many times, each time with different values for some of the identifiers
  - To avoid repeating the same program code, we can create a function
- A function defines program code to execute
  - A function is just another kind of value in memory, like a number (as we will see later)
  - Variables can be bound to functions just as easily as to numbers
- The function Sqr returns the square of its input:

```
declare
fun {Sqr X} X*X end
```

- The **fun** keyword identifies the function. The identifier Sqr refers to a variable that is bound to the function.

39

Numbers

- There are two kinds of numbers in Oz
  - Exact numbers: integers
  - Approximate numbers: floating point
- Integers are exact (arbitrary precision)
- Floats are approximations to real numbers (around 15 digits precision, using 64-bit representation)
- In Oz there is never any automatic conversion from exact to approximate and vice versa
  - To convert, we use functions IntToFloat or FloatToInt
  - Design principle: don't mix incompatible concepts

40

SumDigits3

- Function SumDigits3 calculates the sum of digits of a three-digit positive integer:

```
declare
fun {SumDigits3 N}
  (N mod 10) + ((N div 10) mod 10) +
  ((N div 100) mod 10)
end
```

- **mod** and **div** are integer functions
- / (division) is a float function
- + (addition) is a function on both floats and integers

41

SumDigits6

- Sum of digits of a six-digit positive integer

```
fun {SumDigits6 N}
  {SumDigits3 (N div 1000)} +
  {SumDigits3 (N mod 1000)}
end
```

- This is an example of function composition: defining a function in terms of other functions
  - This is a key ability for building large systems: we can build them in layers, where each layer is built by a different person
  - This is the first step towards data abstraction

42

SumDigitsR (first try)

- Sum of digits of any positive integer
- We use recursion: the function calls itself with a smaller argument

```
fun {SumDigitsR N}
   (N mod 10) + {SumDigitsR (N div 10)}
end
```

- This function calls itself with a smaller value
  - What is the problem with this definition?

43

SumDigitsR (correct)

- Sum of digits of any positive integer

```
fun {SumDigitsR N}
   if (N==0) then 0
   else
      (N mod 10) + {SumDigitsR (N div 10)}
   end
end
```

- This introduces the conditional (**if**) statement
- This is a correct example of function recursion: defining a function that calls itself
  - This is a key ability for building complex algorithms: we divide a complex problem into simpler subproblems (divide and conquer)

44

We can now do functional programming

- We're done! We can now do functional programming.
  - Calculating with numbers, functions, and conditionals can do everything!
- This language is Turing complete
  - We can compute anything that any computer can compute
- We can extend it with more concepts and techniques
  - Invariant programming, symbolic programming, higher-order programming, data abstraction (including objects), concurrent programming
  - How is this better, since the original language is already Turing complete?
- Is there an even smaller language for functional programming?
  - Yes, the lambda calculus (λ calculus) is the smallest!
  - We will see the lambda calculus later in the course
  - All programming languages are based on the lambda calculus

45

Invariant programming

46

Invariant programming: loops!

- A loop is a part of a program that is repeated until a condition is satisfied
  - Loops are an important technique in all paradigms
  - In functional programming, loops are done with recursion
- We give a general technique, invariant programming, to program correct and efficient loops
  - Loops are often hard to get exactly right, and invariant programming is an excellent way to make them right
  - Invariant programming works for all paradigms, including functional programming and Java's object-oriented programming
- Each loop has one invariant
  - An invariant is a formula that is true at the beginning of each loop

47

Doing loops with recursion…

- Let's take another look at SumDigitsR:

```
fun {SumDigitsR N}
    if (N==0) then 0
    else (N mod 10) + {SumDigitsR (N div 10)} end
end
```

- The recursive call and the condition together act like a loop: a calculation that is repeated to achieve a result
  - Each execution of the function body is one iteration of the loop
- We see that recursion can make a loop
  - Let's show a simpler example to understand better how it works!

48

Naïve factorial function

- We can define factorial mathematically:
  - $0! = 1$
  - $n! = n \times (n-1)!$ when $n>0$
- We can define it as a program:

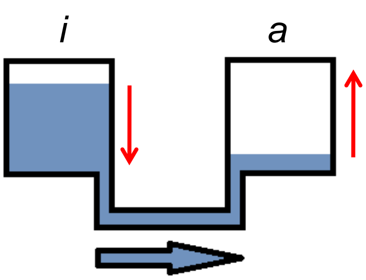
```
fun {Fact1 N}
    if N==0 then 1
    else N*{Fact1 N-1} end
end
```

- This looks ok, right? It's actually very bad!

49

Better factorial function

- There is a much better way to define factorial!
- We start with an invariant, which is a logical formula that splits the work into two parts
  - We want to calculate n!
  - Split it into two parts: $n! = i! \times a$
- We do communicating vases
  - *n* is constant
  - *i* and *a* are varying
  - Start with $i=n$ and $a=1$
  - Decrease *i* and increase *a*, while keeping the formula true
  - When $i=0$ then *a* is the answer



50

Better factorial function

- Using the invariant gives another factorial function
  - The varying parts (*i* and *a*) are arguments of Fact2
  - The constant part (*n*) is the first call to Fact2

```
fun {Fact2 I A}
  if I==0 then A
  else {Fact2 I-1 I*A} end
end
{Browse {Fact2 N 1}}
```

- This one is much better than Fact1!
  - Let's see why by looking at the execution of both

51

Comparing Fact1 and Fact2

- 10 * {Fact1 10-1} ⇒
  10 * (9 * {Fact 9-1}) ⇒
  10 * (9 * (8 * {Fact 8-1})) ⇒
  ...
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * {Fact 0})...) ⇒
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * 1)...) ⇒
  ...
  3628800

Each line does one computation step.
The expression gets bigger and bigger!
All the multiplications are done at the end

- {Fact2 10-1 10*1} ⇒
  {Fact2 9-1 9*10} ⇒
  {Fact2 8-1 8*90} ⇒
  ...
  {Fact2 1-1 1*3628800}
  3628800

? [It seems that Fact2 is better than Fact1, because the expression does not grow. How can we make this intuition precise? We need to introduce a formal semantics of the execution.]
→ we will see it in lectures 2 & 3

52

Fact2 is tail-recursive

- Tail recursion is when the recursive call is the last operation in the function body
- N * {Fact1 N-1}   % No tail recursion

  After Fact1 is done, we must come back for the multiply.
  Where is the multiplication stored? On a stack!

- {Fact2 I-1 I*A}   % Tail recursion
  The recursive call does not come back!
  All calculations are done *before* Fact2 is called.
  No stack is needed (memory usage is constant).

53

Sum of digits using invariant programming

- Each recursive call handles one digit
- So we divide the initial number *n* into its digits:
  - $n = (d_{k-1}d_{k-2}\cdots d_2 d_1 d_0)$ *(where d_i is a digit)*
- Let's call the sum of digits function *s(n)*
- Then we can split the work in two parts:
  - $s(n) = \underbrace{s(d_{k-1}d_{k-2}\cdots d_i)}_{s_i} + \underbrace{(d_{i-1} + d_{i-2} + \cdots + d_0)}_{a}$
  - s_i is the work not yet done and *a* is the work already done
  - To keep the formula true, we set $i' = i+1$ and $a' = a+d_i$
  - When $i=k$ then $s_k=s(0)=0$ and therefore *a* is the answer

54

Example execution

- Example with *n=314159*:

  $s(n) = s(d_{k-1}d_{k-2}\cdots d_i) + (d_{i-1} + d_{i-2} + \cdots + d_0)$

  - s(314159) = s(314159) + 0
  - s(314159) = s(31415) + 9
  - s(314159) = s(3141) + 14
  - s(314159) = s(314) + 15
  - s(314159) = s(31) + 19
  - s(314159) = s(3) + 20
  - s(314159) = s(0) + 23 = 0 + 23 = 23

55

Final SumDigits2 program

- S = $(d_{k-1}d_{k-2}\cdots d_i)$
  A = $(d_{i-1} + d_{i-2} + \cdots + d_0)$

```
fun {SumDigits2 S A}
   if S==0 then A
   else
      {SumDigits2 (S div 10) A+(S mod 10)}
   end
end
{Browse {SumDigits2 314159 0}}
```

56

Invariant programming is best

- We have now programmed two problems
  - Factorial
  - Sum of digits
- For each problem we have defined two functions
  - First version based on a simple mathematical definition
  - Second version designed with invariant programming BEST
- The second version has three interesting properties
  - It uses constant memory space, unlike the first version
  - It has two arguments: the growing one is called an accumulator
  - The recursive call is the last operation: it is tail-recursive

Tail recursion = while loop

- A while loop in functional programming:

```
fun {While S}
   if {IsDone S} then S
   else {While {Transform S}} end /* tail recursion */
end
```

- A while loop in imperative programming:
  (i.e., in languages with multiple assignment like Java and C++)

```
state whileLoop(state s) {
   while (!isDone(s))
      s=transform(s); /* assignment */
   return s;
}
```

- In *both* cases, invariant programming is the right way to do loops

Summary

59

Summary

- Overview of course
- Practical organization
- Basic concepts
- Tips on Oz syntax
- Functional programming
- Invariant programming (loops)

- Next week:
  symbolic programming

60