Programming Language Concepts, CS2104
Lecture 1

Dr Chin Wei Ngan
chinwn@comp.nus.edu.sg
Office: S15 6-01 (old site)
Programming Language Concepts

- CS2104 is a 4 credit points module
  - Written final exam 50%
  - Midterm exam 25%
  - Lab/tutorial assignments 25%

- Module homepage
  - http://www.comp.nus.edu.sg/~cs2104
  - IVLE

- Teaching
  - Lectures: Friday, 12:00-14:00, COM1/206
  - Exam: 27 Nov 2007, morning (Tue)
The Team

- Lectures
  - Dr. Chin Wei-Ngan (Consultation: Wed 9-11am but other times OK too but email me first.)

- Lectures based of the book:
  - Slides from CS2104, 2003-2006

- Recommended books:
Lab Assignment Submissions

- Student submission through IVLE
- Please use CS2104, Workbin
Lecture Structure

- Reminder of last lecture
- Overview
- Content (new notions + examples)
- Summary
- Reading suggestions
Tutorials/ Labs

- Purposes
  - for self-assessment
  - use material from lectures
  - answer questions
  - help deeper understanding
  - prepare lab assignments
  - compulsory tutorial attendance + credits (up to 10%)

- Supervised lab session
  - First lab/assignment: to be announced
  - done by students (with help from teaching assistant)

- You can discuss tutorials/chapters on the IVLE discussion groups
Assignments

- There will be 4 or 5 lab assignments
- Deadline is strict! Don’t leave till last-minute.
- Mostly individual programming projects
- Code of conduct
  - no copying (grade penalty for those caught)
  - plagiarism is cheating and can lead to expulsion!
Useful Software

- http://www.mozart-oz.org/
  - programming language: Oz
  - system: Mozart (1.3.0, released on April 15, 2004)
  - interactive system
- Requires **emacs** on your computer
- Install yourself
- First lab/assignment will help on installation
Aim

- Knowledge and skills in
  - Programming languages concepts
  - Corresponding programming techniques

- Acquaintance with
  - Key programming concepts/techniques in computer science
  - Focus on concepts and not on a particular language
Overview

- Introduction of main concepts:
  - Computation model
  - Programming model
  - Reasoning model
Programming

- **Computation model**
  - formal system that defines a language and how sentences (expressions, statements) are executed by an abstract machine

- **Programming model**
  - a set of programming techniques and design principles used to write programs in the language of the computation model

- **Reasoning model**
  - a set of reasoning techniques to let you reason about programs, to increase confidence that they behave correctly, and to estimate their efficiency
Computation Models

- **Declarative** programming (stateless programming)
  - functions over partial data structures

- **Concurrent** programming
  - can interact with the environment
  - can do independent execution of program parts

- **Imperative** programming (stateful programming)
  - uses states (a state is a sequence of values in time that contains the intermediate results of a desired computation)

- **Object-oriented** programming
  - uses object data abstraction, explicit state, polymorphism, and inheritance
Programming Models

- Exception handling
  - Error management

- Concurrency
  - Dataflow, lazy execution, message passing, active objects, monitors, and transactions

- Components
  - Programming in the large, software reuse

- Capabilities
  - Encapsulation, security, distribution, fault tolerance

- State
  - Objects, classes
Reasoning Models

- **Syntax**
  - Extended Backus-Naur Form (EBNF)
  - Context-free and context-sensitive grammars

- **Semantics**
  - Operational: shows how a statement executes as an abstract machine
  - Axiomatic: defines a statement as a relation between input state and output state
  - Denotational: defines a statement as a function over an abstract domain
  - Logical: defines a statement as a model of a logical theory

- **Programming language**
  - Implements a programming model
  - Describes programs composed of **statements** which compute with **values** and **effects**
Examples of Programming Languages

- **CS1102: Java**
  - programming with explicit state
  - object-oriented programming
  - concurrent programming (threads, monitors)

- **CS2104: Oz (multi-paradigm)**
  - declarative programming
  - concurrent programming
  - programming with explicit state
  - object-oriented programming
The focus is on the programming model, techniques and concepts, but **not** the particular language!

**Approach**
- informal introduction to important concepts
- introducing the underlying kernel language
- formal semantics based on abstract machine
- in depth study of programming techniques
Declarative Programming Model Philosophy

- Ideal of declarative programming
  - say **what** you want to compute
  - let computer find **how** to compute it

- More pragmatically
  - let the computer provide more support
  - free the programmer from some burden
Properties of Declarative Models

- Focus on **functions** which compute when given data structures as inputs
- Widely used
  - functional languages: LISP, Scheme, ML, Haskell, …
  - logic languages: Prolog, Mercury, …
  - representation languages: XML, XSL, …
- Stateless programming
  - no update of data structures
  - Simple data transformer
The Mozart System

- Built by Mozart Consortium (Universität des Saarlandes, Swedish Institute of Computer Science, Université catholique de Louvain)

- Interactive interface (the `declare` statement)
  - Allows introducing program fragments incrementally and execute them
  - Has a tool (Browser), which allows looking into the store using the procedure `Browse`
    - `{Browse 21 * 10}  -> display 210`

- Standalone application
  - It consists of a main function, evaluated when the program starts
  - Oz source files can be compiled and linked
Concept of Single-Assignment Store

- It is a **set of variables** that are initially **unbound** and that can be **bound** to one value.

- A **value** is a mathematical constant that does not change.

  For e.g: $2, \sim 4, \text{true}, \text{'}a\text{'}, [1 \ 2 \ 3]$

- Examples:
  - $\{x_1, x_2, x_3\}$ has three unbound variables
  - $\{x_1=2, x_2=\text{true}, x_3\}$ has only one unbound variable
Concept of Single-Assignment Store

- A store where all variables are bound to values is called a value store:
  \[ \{x_1=2, x_2=true, x_3=[1 2 3]\} \]

- Once bound, a variable stays bound to that value

- So, a value store is a persistent mapping from variables to values

- A store entity is a store variable and its value (which can be unbound).
Concept of Single-Assignment Store

- Single-assignment store is set of (store) variables
- Initially variables are unbound
- Example: store with three variables, $x_1$, $x_2$, and $x_3$
Concept of Single-Assignment Store

- Variables in store may be bound to values
- Example: assume we allow values of type integers and lists of integers

```
Store

\( x_1 \) unbound
\( x_2 \) unbound
\( x_3 \) unbound
```
Concept of Single-Assignment Store

Examples:
- $x_1$ is bound to integer 314
- $x_2$ is bound to list [1 2 3]
- $x_3$ is still unbound
Concept of Declarative Variable

- It is a variable in the single-assginment store
- Created as being \textit{unbound}
- Can be \textit{bound} to exactly one value
- Once bound, stays bound
  - indistinguishable from its value

\[
\begin{array}{c|c|c|c}
\text{Store} & x_1 & 314 \\
\hline
x_2 & 1 & 2 & 3 | \text{nil} \\
\hline
x_3 & \text{unbound} \\
\end{array}
\]
**Concept of Value Store**

- Store where all variables are bound to values is called a *value store*

- Examples:
  - $x_1$ bound to integer 314
  - $x_2$ bound to list [1 2 3]
  - $x_3$ bound to record
    - person(name: george, age: 25)

- Functional programming computes functions on values
Concept of Single-Assignment Operation

\( x = value \)

- It is also called “value creation”
- Assumes that \( x \) is unbound
- Examples:
  - \( x_1 = 314 \)
  - \( x_2 = [1 \ 2 \ 3] \)
Concept of Single-Assignment Operation

\[ x = \text{value} \]
- \[ x_1 = 314 \]
- \[ x_2 = [1 \ 2 \ 3] \]
Concept of Single-Assignment Operation

\( x = value \)

- **Single assignment operation** (‘=‘)
  - constructs \( value \) in store
  - binds variable \( x \) to this value

- If the variable is already bound, operation tests compatibility of values
  - if the value being bound is different from that already bound, an error is raised

![Diagram of variable bindings](attachment:image.png)
Concept of Variable Identifier

- Variable identifiers start with capital letter: X, Y2
- The environment is a mapping from variable identifiers to store entities

```declare X = <value>
- creates a new store variable x and binds it to <value>
- maps variable identifier X in environment to store variable x, e.g. {X→x}
```

```declare X = Y
Y = 2
```

- The environment: E={X→x, Y→y}
- The single-assignment store: σ={x=y, y=2}
Concept of Variable Identifier

- Refer to store entities
- Environment maps variable identifiers to store variables
  - `declare X`
  - `local X in ... end`
- X is variable identifier
- Corresponds to 'environment' \{X → x_1\}
Concept of Variable Identifier

- declare
  
  \[
  \begin{align*}
  x &= 21 \\
  x &= 22 \\
  \% \text{ raise an error} \\
  x &= 21 \\
  \% \text{ do nothing} \\
  \text{declare} \\
  x &= 22 \\
  \% \text{ from now on, } x \text{ will be bound to 22}
  \end{align*}
  \]
Partial Value

A partial value is a data structure that may contain unbound variables. For example, $x_2$ is unbound. Hence, $x_1$ is a partial value.
Variable-Variable Binding

- Variables can be bound to variables. They form an equivalence set of store variables after such binding.
- They throw exception if their values are different.

![Diagram showing variable binding and equivalence set](image)

After $X=Y$
Variable-Variable Binding

- After binding one of the variables.

After binding one of the variables.
Concept of Dataflow Variables

- Variable creation and binding can be separated. What happens if we use a variable before it is bound? Scenario is known as **variable use error**.

- Possible solutions:
  1. Create and bind variables in one step (use error cannot occur): functional programming languages
  2. Execution continues and no error message is given (variable’s content is “garbage”): C/C++
  3. Execution continues and no error message is given (variable’s content is initialized with a default value): Java
Concept of Dataflow Variables

- Execution stops with error message (or an exception is raised): Prolog
- Execution is not possible; the compiler detects that there is an execution path to the variable’s use that does not initialize it: Java – local variables
- Execution waits until the variable is bound and then continues (dataflow programming): Oz
Example of Dataflow Variables

```ozone
declare X Y
Y = X + 1
{Browse Y}
```

Running this Oz code, the Oz Browser does not display anything

```ozone
X = 2
```

Running the previous line, the Oz Browser displays 3
Dynamic Typing in Oz

- A variable type is known only after the variable is bound.
- For an unbound variable, its type checking is left for run time.
- An operation with values of wrong type will raise exceptions.
- This setting is dynamically typed.
- In contrast, Java is a static type language, as the types of all variables can be determined at compile time.
- Examples: Types of $x$ maybe Int, Float, ..
  - $x < 1$
  - $x < 1.0$
Concept of Cell

- **A cell** is a multiple-assignment variable
- A memory cell is also called **explicit state**
- Three functions operate on cells:
  - `NewCell` creates a new cell
  - `:=` (assignment) puts a new value in a cell
  - `@` (access) gets the current value stored in the cell

```plaintext
declare
C = {NewCell 0}
{Browse @C}
C := @C + 1
{Browse @C}
```
Concept of Function

- Function definition
  
  ```
  fun {<Identifier> <Arguments>}
  [<Declaration Part> in]
  [<Statement>]
  <Expression>
  end
  ```

- The value of the **last expression in the body** is the **returned value** of the function.

- Function application (call)
  
  ```
  X = {<Identifier> <Arguments>}
  ```
Concept of Function. Examples

declare
fun {Minus X}
   ~X
end
{Browse {Minus 15}}
declare
fun {Max X Y}
   if X>Y then X else Y end
end
declare
X = {Max 22 18}
Y = {Max X 43}
{Browse Y}
Recursive Functions

- Direct recursion: the function is calling itself
- Indirect (or mutual) recursion: e.g. $F$ is calling $G$, and $G$ is calling $F$

General structure
- base case
- recursive case

Typically, for a natural number $n$
- base case: $n$ is zero
- recursive case:
  - $n$ is different from zero
  - $n$ is greater than zero
Inductive Function Definition

- Factorial function: \( n! = 1 \times 2 \times 3 \times \ldots \times n \)
  - inductively defined as
    \[
    \begin{align*}
    0! & = 1 \\
    n! & = n \times ((n-1)!) \\
    \end{align*}
    \]
  - program as function \texttt{Fact}
Inductive Function Definition

- Factorial function definition in Oz

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

{Browse {Fact 5}}
```
Correctness

- The most popular reasoning techniques is mathematical induction:
  - Show that for the simplest (initial) case the program is correct
  - Show that, if the program is correct for a given case, then it is correct for the next case

- \{Fact 0\} returns the correct answer, namely 1

- Assume \{Fact N-1\} is correct. Suppose N>0, then Fact N returns N*\{Fact N-1\}, which is correct according to the Oz inductive hypothesis!

- Fact N for negative N goes into an infinite number of recursive calls, so it is wrong!
Complexity

- The execution time of a program as a function of input size, up to a constant factor, is called the program’s **time complexity**.

```declare
fun {Fibo N}
  case N of
    1 then 1
    [] 2 then 1
    [] M then {Fibo (M-1)} + {Fibo (M-2)}
  end
end

{Browse {Fibo 100}}
```

- The time complexity of \{Fibo N\} is proportional to \(2^N\).
Complexity

```haskell
declare
fun {FiboTwo N A1 A2}
    case N of
        1 then A1
        [] 2 then A2
        [] M then {FiboTwo (M-1) A2 (A1+A2)}
    end
end
{Browse {FiboTwo 100 1 1}}
```

- The time complexity of `FiboTwo N` is proportional to `N`.

```
Concept of Lazy Evaluation

- **Eager** (supply-driven, or data-driven) evaluation: calculations are done as soon as they are called
- **Lazy** (demand-driven) evaluation: a calculation is done only when the result is needed


declare

fun lazy {F1 X} X*X end

fun lazy {Ints N} N|{Ints N+1} end

A = {F1 5}

{Browse A}

% it will display: A

Note that {F1 5} does not execute until it is demanded!
Concept of Lazy Evaluation

- F1 and Ints created “stopped executions” that continue when their results are needed.
- After demanding value of A (function * is not lazy!), we get:
  
  B = {Ints 3}
  C = 2 * A  // A={F1 5}
  {Browse A}
  % it will display: 25
  {Browse B}
  % it will display: B
  case B of X|Y|Z|_ then {Browse X+Y+Z} end
  % it will cause only first three elements of B to be evaluated and then display: 12
  % previous B is also refined to: 3 | 4 | 5 |_
Concept of Higher-Order Programming

- Ability to pass functions as arguments or results
- We want to write a function for $1+2+...+n$ (**GaussSum**)
- It is similar to **Fact**, except that:
  - “$\times$” is “$+$”
  - the initial case value is not “0” but “1”
- The two operators are written as functions; they will be arguments for the generic function

```
fun {Add X Y} X+Y end
fun {Mul X Y} X*Y end
```
Concept of Higher-Order Programming

The generic function is:

```pascal
fun {GenericFact Op InitVal N} 
  if N == 0 then InitVal 
  else {Op N {GenericFact Op 
                InitVal (N-1)}}
  end
end
```
Concept of Higher-Order Programming

- The instances of this generic function may be:
  
  ```haskell
  fun {FactUsingGeneric N}
      {GenericFact Mul 1 N}
  end

  fun {GaussSumUsingGeneric N}
      {GenericFact Add 0 N}
  end
  
  They can be called as:
  
  {Browse {FactUsingGeneric 5}}
  {Browse {GaussSumUsingGeneric 5}}
  ```
Concept of Concurrency

- Is the ability of a program to run independent activities (not necessarily to communicate)
- A **thread** is an executing program
- **Concurrency** is introduced by creating threads

```
thread P1 in
  P1 = {FactUsingGeneric 5}
  {Browse P1}
end

thread P2 in
  P2 = {GaussSumUsingGeneric 5}
  {Browse P2}
end
```
Concept of Dataflow

- Is the ability of an operation to wait until all its variables become bounded

```plaintext
declare X in
thread {Delay 5000} X = 10 end
thread {Browse X * X} end
thread {Browse 'start'} end
```

- The second `Browse` waits for `X` to become bound
- `X = 10` and `X * X` can be done in any order, so dataflow execution will always give the same result

```plaintext
declare X in
thread {Delay 5000} {Browse X * X} end
thread X = 10 end
thread {Browse 'start'} end
```

- Dataflow concurrency (Chapter 4)
Concept of Object

- It is a function with internal memory (cell)

```plaintext
declare
local C in
  C = {NewCell 0}
  fun {Incr}
    C := @C + 1
    @C
  end
  fun {Read} @C end
end
```

- C is a counter object, Incr and Read are its interface
- The declare statement makes the variables Incr and Read globally available. Incr and Read are bounded to functions
Concept of Object-Oriented Programming

- Encapsulation
  - Variable $C$ is visible only between `local` and `end`
  - User can modify $C$ only through `Incr` function (the counter will work correctly)
  - User can call only the functions (methods) from the interface
    `{Browse {Incr}}`
    `{Browse {Read}}`

- Data abstraction (Section 6.4)
  - Separation between interface and implementation
  - User program does not need to know the implementation

- Inheritance (Chapter 7)
Concept of Class

- It is a “factory” which creates objects

```
declare
fun {ClassCounter} C Incr Read in
  C = {NewCell 0}
  fun {Incr}
    C := @C + 1
    @C
  end
  fun {Read}
    @C
  end
end
counter(incr:Incr read:Read)
end
```
Concept of Class

- **ClassCounter** is a function that creates a new cell and returns new functions: **Incr** and **Read** (recall higher-order programming)
- The record result groups the methods so that they can be accessed by its fields.

```
declare
Counter1 = {ClassCounter}
Counter2 = {ClassCounter}
```

- The methods can be accessed by “.” (dot) operator
  ```
  {Browse {Counter1.incr}}
  {Browse {Counter2.read}}
  ```
Concept of Nondeterminism

- It is concurrency + state
- The order in which threads access the state can change from one execution to the next
- The time when operations are executed is not known
- Interleaving (mixed order of threads statements) is dangerous (one of most famous concurrent programming error:
- Solution: An operation is atomic if no intermediate states can be observed (Chapter 8)
Summary

- Oz, Mozart
- Variable, Type, Cell
- Function, Recursion, Induction
- Correctness, Complexity
- Lazy Evaluation
- Higher-Order Programming
- Concurrency, Dataflow
- Object, Classes
- Nondeterminism
Reading suggestions

- From [van Roy, Haridi; 2004]
  - Chapter 1
  - Appendix A
  - Exercises 1.18.1-1.18.10

- From [Tucker, Noonan; 2002]
  - Chapter 1
  - Exercises 1.1-1.7 from [Tucker, Noonan; 2002]

- First lab/assignment: Fri 24 Aug 2007 (15:00-18:00 Venue: ?) Compulsory attendance. Choose a 1-hr session.