HIT3315
Languages in Software Development
Semester 2, 2008
HIT3315

Convener: Dr. Markus Lumpe
EN 508c
http://www.ict.swin.edu.au/personal/mlumpe/hit3315

Lecture: Wednesday 11:30, BA801

Labs: Wednesday 13:30, BA601

Grading: problem sets, midterm, final exam

Assignments: on a weekly basis
Subject Aims

- What are the expressive ways of specifying computational processes?
- How can a given problem be effectively expressed?
- What are suitable formal frameworks for the definition and the implementation of a programming language?
Selected Objectives

- Solve problems in different paradigms.
- Assess strengths and weaknesses of compiled and interpreter-based languages.
- Explain and answer questions about specific languages.
- Evaluate programming language features and designs.
Overview

Tentative subject program:

- Introduction - Basic Concepts
- Inductive Sets of Data
- Introduction to Lambda Calculus
- Environment-Passing Interpreters:
  - Functional Concepts
  - Imperative Concepts
  - Object-Oriented Concepts
  - Hypertext Concepts
- Typing and type inference
Why Do We Study Programming Languages?

- Some have suggested that there is no need to develop new computer languages nor even to teach language design and compiler theory.

Correct/Wrong
What Is a Programming Language?

- A formal notation for describing computation
- A “user interface” to a computer
- A more precise tool than any natural language

- Programming paradigms – different expressive power
- Syntax + semantics
- Compiler, or interpreter, or translator
Core Properties of Programming Languages

- Languages provide the framework for the way we organize complexity in our own minds.

- Languages are the means by which we communicate our understanding.
Reasons for Studying Concepts of Programming Languages

The potential benefits of studying language concepts are:

- **Increased capacity to express ideas.**
  - The way we think is greatly influenced by the expressive power of the language in which we communicate our thoughts.

- **Improved background for choosing appropriate languages.**
  - Many programmers, when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to the new project.

- **Increased ability to learn, to design, and to implement a new language.**
How Do Programming Languages Differ?

Generations:
- **1GL**: machine code
- **2GL**: symbolic assemblers
- **3GL**: (machine independent) languages
  - Fortran, Algol, Pascal, Smalltalk, C++, Java, Lisp, Haskell, Scheme, Prolog
- **4GL**: domain specific application generators

Common Constructs:
- basic data types (numbers, etc.); variables; expressions; statements;
- keywords; control constructs; procedures; comments ...

Uncommon Constructs:
- type declarations; special types (strings, arrays, matrices, ...);
- concurrency constructs; packages/modules; generics; exceptions; ...
Key Theses

Thesis 1: *Speak the programming language that you need to work with.*
- Every programming language meets some specialized goal.

Thesis 2: *Programming languages are invented while you sleep, and spread before you wake up.*
- Many languages already address your problem; the user only needs to find the appropriate language.

Thesis 3: *Understanding programming languages is the key to your job.*
Programming Domains

All programming languages have been developed with different goals in mind. Every language has its designated application domain, which, in general, requires a specific set of programming abstractions or/and runtime models.

- Scientific applications: floating-point arithmetic (Fortran, Algol)
- Business applications: reports, decimal arithmetic (Cobol)
- Artificial intelligence: symbolic computation (Lisp, Prolog)
- System programming: operating systems (C, Pascal)
- Scripting languages: system configuration (sh, awk, Perl, Tcl)
Programming Paradigms

- Imperative style:
  \[ \text{program} = \text{algorithms} + \text{data} \]

- Functional style:
  \[ \text{program} = \text{function} \cdot \text{function} \]

- Logic programming style:
  \[ \text{program} = \text{facts} + \text{rules} \]

- Object-oriented style:
  \[ \text{program} = \text{objects} + \text{messages} \]

Other styles and paradigms:
blackboard, events, pipes and filters, constraints, lists, ...
Imperative Programming

- This is the oldest style of programming, in which the algorithm for the computation is expressed explicitly in terms of instructions such as assignments, tests, branching and so on.

- Execution of the algorithm requires data values to be held in variables which the program can access and modify.

- Languages so classified include assembly languages, Fortran, Algol, Pascal, C, and Ada.

- Imperative programming corresponds naturally to the earliest, basic and still used model for the architecture of the computer, the von Neumann model.
Functional Programming

- Functional programming takes a much more mathematical approach, based on the lambda calculus.

- The concept of variables is not used in pure functional programming. Instead, the computation is described as a function, which is applied to the input data and which gives the result(s) as output data.

- This style is more abstract since it requires the algorithm to be described in a way that is independent of the data.

- Most prominent languages of this style are Lisp, ML, Scheme, and Haskell.
Logic Programming

- Logic programming is like functional programming, it also takes a mathematical approach, but this time through formal logic.

- A program is described in terms of predicates, which are the rules that govern the problem. At run-time the use of logical inference enables new formulae to be deduced from those given, or the truth or falsehood of a formula to be deduced from the predicates (full unification).

- Logical inference is very much like the process of proving a theorem in mathematics, starting from the axioms and theorems already proved.

- The best-known logic language is Prolog.
Object-oriented Programming

- In general, object-oriented languages are based on the concepts of class and inheritance, which may be compared to those of type and variable respectively in a language like Pascal.

- A class describes the characteristics common to all its instances, in a form similar to the record of Pascal, and thus defines a set of fields.

- In object-oriented programming, instead of applying global procedures or functions to variables, we invoke the methods associated with the instances, an action called “message passing”.

- The basic concept inheritance is used to derive new classes from exiting ones by modifying or extending the inherited class(es).

- The most prominent object-oriented languages are Smalltalk, C++, Eiffel, Java, and ObjectPascal.
Sequential Languages

- Instructions are executed one after another in an order that is deduced from the text of the program.

- These are the most widely used languages, since they correspond to the classic von Neumann architecture.

- Pascal, Haskell, Smalltalk, and Java, for example, are members of the class of sequential languages.
Parallel Languages

- In contrast to sequential languages, several program instructions can be executed simultaneously.

- These languages are designed to develop programs for multi-processor (distributed memory) architectures.

- Parallel languages demand special constructs for communication and synchronization.

- The general model for programming in terms of objects can easily be made parallel - actor languages.

- The most prominent parallel languages are Occam and Actor.
Special-purpose Languages

- **Sh, Awk, Perl, Python, JavaScript:**
  - Rapid prototyping
  - System administration
  - Program configuration

- **Postscript, HPGL, Tex, RTF:**
  - Text setting
  - Description of text, graphical shapes, and images

- **HTML, XML:**
  - Markup languages
A Brief History

- Fortran, 1957
- Algol-60
- Cobol, 1960
- PL/I, 1965
- Lisp, 1960
- Basic, 1964
- Prolog, 1970
- Scheme, 1975
- Simula, 1962
- Algol-68
- Pascal, 1975
- C, 1972
- Prolog, 1970
- Ada, 1983
- C++, 1986
- Java, 1993
- Scheme, 1975
- Haskell, 1990
Syntax

- The syntax of a programming language is concerned with the **form of programs**, i.e., how expressions, commands, declarations, etc. are put together to form programs.

- A well-designed programming language will have a well-designed syntax. However, the syntax definition given for a specific language is not powerful enough to define a programming language completely. The purpose of a well-defined syntax is to guide the programmer to understand the language's semantics.
Semantics

- The semantics of a programming language is concerned with the **meaning of programs**, i.e., how they behave when executed on computers.

- The semantics of a programming language assigns a precise meaning to every sentence of the language that can be formed using the given syntax definition. There are three approaches to define the semantics of a programming language:
  - Axiomatic semantics,
  - Operational semantics,
  - Denotational semantics.
The Hilbert-style Proof System

- A Hilbert-style proof system consists of axioms and proof rules:
  - An axiom of a proof system is a formula that is provable by definition.
  - An inference rule asserts that if some list of formulas is provable, then so is another formula.
  - A proof is a structured object built from formulas according to constraints established by a set of axioms and inference rules.
Proofs

- The rule format:

\[
\begin{array}{c}
\text{Premise}_1 & \text{Premise}_2 & \ldots & \text{Premise}_n \\
\hline
\text{Conclusion}
\end{array}
\]

- We construct a proof from proofs:

\[
\begin{array}{c}
\text{Premise}_1 & \text{Premise}_2 & \ldots & \text{Premise}_n \\
\hline
\text{Conclusion}_1 & \text{Conclusion}_2 & \ldots & \text{Conclusion}_n \\
\hline
\text{Conclusion}
\end{array}
\]
**Axiomatic Semantics**

- The axiomatic semantics is a formal (proof) system for deriving equations between expressions.

- The basic idea of the axiomatic method is to define the meaning of language elements indirectly using logical assertions. For example, we can write \( \{E_1\} C \{E_2\} \), called a **Hoare triple**, to state that if the boolean expression \( E_1 \) holds prior the computation of \( C \), and if \( C \) terminates, then the boolean expression \( E_2 \) must hold as well.

- Examples:

  \[
  \{ \ a > 0 \ \} \ a := a + 1 \ \{ \ a > 1 \ \}
  \]

  \[
  \{E_1\} \ C_1 \ {E_2}\ \quad \{E_2\} \ C_2 \ {E_3}\ \\
  \{E_1\} \ C_1;C_2 \ {E_3}\]
## Example Rules

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Proof Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C = \text{def} )</td>
<td>( C.\text{Target} := C.\text{Source} )</td>
</tr>
<tr>
<td></td>
<td>( {Q[C.\text{Source} \setminus C.\text{Target}]}C{Q} )</td>
</tr>
</tbody>
</table>
| \( C = \text{def} \) | \( \begin{align*} \text{if } C.\text{Test} \\
|                      | \quad \text{then } C.\text{Then} \\
|                      | \quad \text{else } C.\text{Else} \end{align*}                                                                                           |
|                      | \( \{C.\text{Test} \land P\}C.\text{Then}\{Q\} \quad \{\neg C.\text{Test} \land P\}C.\text{Else}\{Q\} \quad \{P\}C\{Q\} \)       |
| Rule of Consequence  | \( P \Rightarrow P' \quad \{P'\}C\{Q'\} \quad Q' \Rightarrow Q \quad \{P\}C\{Q\} \)                                               |
Using axiomatic semantics, we need to prove the validity of a given Hoare triple.

Example:

\[
\{ \text{true} \} \quad \text{if} (a \geq b) \quad \text{then} \quad m = a; \quad \text{else} \quad m = b; \quad \{ m = \max(a, b) \}
\]
Proof

Premise I:

\[ a \geq b \Rightarrow a = \max(a, b) \quad \text{true} \]

\[
\begin{align*}
\{a = \max(a, b)\} & m = a;\{m = \max(a, b)\} \\
\{a \geq b\} & m = a;\{m = \max(a, b)\}
\end{align*}
\]

Premise II:

\[ a < b \Rightarrow b = \max(a, b) \quad \text{true} \]

\[
\begin{align*}
\{b = \max(a, b)\} & m = b;\{m = \max(a, b)\} \\
\{a < b\} & m = b;\{m = \max(a, b)\}
\end{align*}
\]

\[
\begin{align*}
\{a \geq b \land \text{true}\} & m = a;\{m = \max(a, b)\} \\
\{a < b \land \text{true}\} & m = b;\{m = \max(a, b)\} \\
\{\text{true}\} & C\{m = \max(a, b)\}
\end{align*}
\]
Operational Semantics

- The operational semantics is based on a directed form of equational reasoning called "reduction". Reduction may be regarded as a form of symbolic evaluation.

- The basic idea of the operational method is to define the meaning of the language elements by means of a (labeled) transition system.

- The operational semantics definition provides means to display the computation steps undertaken when a program is evaluated to its output.

- Some forms of operational semantics are interpreted-based, with instruction counters, data structures, and the like, and others are inference rule-based, with proof trees that show control flows and data dependencies.
# Example Transition Rules

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Transition Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \overset{\text{def}}{=} def$</td>
<td>$\sigma(C.\text{Source}) = v$</td>
</tr>
<tr>
<td>$C.\text{Target} := C.\text{Source}$</td>
<td>$\sigma(C.\text{Target} := C.\text{Source}) \rightarrow \sigma \cup {(C.\text{Target}, v)}$</td>
</tr>
<tr>
<td>$C \overset{\text{def}}{=} def$</td>
<td></td>
</tr>
<tr>
<td>if $C.\text{Test}$ then $C.\text{Then}$ else $C.\text{Else}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sigma(C.\text{Test} = \text{true}) \sigma(C.\text{Then}) \rightarrow \sigma'$</td>
</tr>
<tr>
<td></td>
<td>$\sigma(\text{if } C.\text{Test then } C.\text{Then else } C.\text{Else}) \rightarrow \sigma'$</td>
</tr>
<tr>
<td></td>
<td>$\sigma(C.\text{Test} = \text{false}) \sigma(C.\text{Else}) \rightarrow \sigma'$</td>
</tr>
<tr>
<td></td>
<td>$\sigma(\text{if } C.\text{Test then } C.\text{Then else } C.\text{Else}) \rightarrow \sigma'$</td>
</tr>
</tbody>
</table>
Denotational Semantics

- The denotational semantics, or model theory, is defined in the spirit of equational logic or first-order logic. A denotational semantics definition (model) consists of a family of sets, one for each type, with the property that each well-typed expression may be interpreted as a specific element of the appropriate set.

- The denotational semantics is a recursive definition that maps well-typed derivation trees to their mathematical meanings. For example, the set Bool consists of two meanings: $\text{Bool} = \{\text{true, false}\}$ and an operation $\text{not} : \text{Bool} \to \text{Bool}$ with $\text{not}(\text{false}) = \text{true}$, $\text{not}(\text{true}) = \text{false}$.

- The denotational method does not maintain states, but the meaning of a program is given as a function that interprets all language elements of a given program as elements of a corresponding set of values.
# Example Meaning Functions

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Meaning Functions</th>
</tr>
</thead>
</table>
| $C = \text{def}$  
$\text{C.Target} := \text{C.Source}$ | $f(\langle \text{C.Target} := \text{C.Source} \rangle, \sigma) = \begin{array}{l} \text{if } (f(\langle \text{C.Target} \rangle, \sigma) \neq \text{nil}) \\
\text{then } \sigma \cup \{(f(\langle \text{C.Target} \rangle, \sigma), f(\langle \text{C.Source} \rangle, \sigma))\} \\
\text{else } \text{ERROR} \end{array}$ |
| $C = \text{def}$  
$\text{if } C.\text{Test} \text{ then } \text{C.Then} \text{ else } \text{C.Else}$ | $f(\langle \text{if } C.\text{Test} \text{ then } \text{C.Then} \text{ else } \text{C.Else} \rangle, \sigma) = \begin{array}{l} \text{if } f(\langle \text{C.Test} \rangle, \sigma) \\
\text{then } f(\langle \text{C.Then} \rangle, \sigma) \\
\text{else } f(\langle \text{C.Else} \rangle, \sigma) \end{array}$ |
Types and Type Systems

- Types are **collections of values** that share some common properties. When we say that \( v \) is a value of type \( T \), we mean that \( v \in T \).

- In some systems, there may be types with types as members. Types with types as members are usually called something else, such as *universes*, *orders* or *kinds*, to avoid the impression of circularity.

- In a type system, types provide a division or classification of some universe of possible values. A type system defines in a mathematical way (axioms and deduction-rules), which expressions are typable, i.e., which expressions can be assigned a valid type using the underlying type system.

- In most programming languages, types are “checked” in some way, either during program compilation, or during execution. The main purpose of type checking is the detection of errors, documentation, program optimization, etc.
Values

- In computer science we classify as a value everything that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.

- In computer science, as in mathematics, an “expression” is used (solely) to denote a value.

- Which kinds of values are supported by a specific programming language is heavily depended on the underlying paradigm and its application domain.

- Most programming languages share some basic sets of values like truth values, integers, real number, records, lists, etc.
Constants

- Constants are named abstractions of values.
- Constants are used to assign an user-defined meaning to a value.

Examples:
- EOF = -1
- TRUE = 1
- FALSE = 0
- PI = 3.1415927
- MESSAGE = “Welcome to HIT3315”

- Constants do not have an address, i.e., they do not have a location.

- At compile time, applications of constants are substituted by their corresponding definition.
Primitive Values

- Primitive values are these values that cannot further decomposed. Some of these values are implementation and platform dependent.

- Examples:
  - Truth values,
  - Integers,
  - Characters,
  - Strings,
  - Enumerands,
  - Real numbers.
Composite Values

- Composite values are built up using primitive values and composite values. The layout of composite values is in general implementation dependent.

- Examples:
  - Records,
  - Arrays,
  - Enumerations,
  - Sets,
  - Lists,
  - Tuples,
  - Files.
Pointers

- Pointers are references to values, i.e., they denote locations of a value.

- Pointers are used to store the address of a value (variable or function) – pointer to a value, and pointers are also used to store the address of another pointer – pointer to pointer.

- In general, it not necessary to define pointers with a greater reference level than pointer to pointer.

- In modern programming languages, we find pointers to variables, pointers to pointer, function pointers, and object pointers, but not all programming languages provide means to use pointers directly (e.g. Java, Scheme).