Languages and Compilers
(SProg og Oversættere)

Lecture 14-1
Programming Language Design – Subprograms

Bent Thomsen

Department of Computer Science

Aalborg University

Learning Goals

- Gain insigt into abstractions in programming languages
  - The principle of Abstraction

- Programming language design evaluation methods

Syntactic Elements

- Declarations and Definitions
  - Scopes and visibility
  - always before use or not, initialization or not,
- Expressions
- Statements
- **Subprograms**

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Subprograms

1. A subprogram has a single entry point

2. The caller is suspended during execution of the called subprogram

3. Control always returns to the caller when the called subprogram's execution terminates

Functions or Procedures?

- Procedures provide user-defined statements
  - Abstractions over statements
- Functions provide user-defined operators
  - Abstractions over expressions
- Methods used for both functions and procedures

Subprograms

- Specification: name, signature, actions
  - C/C++: `typ0 f(typ1 para1, typ2 para2, ...) { ... }`
  - SML: `fun f para1 para2 = ...`
  - Pascal:
```
function f(para1 : typ1, para2 : typ2, ...) : retval;
        var retval : typ0;
        begin  ...  end
```
- Signature: number and types of input arguments, number and types of output results
  - Sometimes this is called the subprogram protocol
- Actions: direct function relating input values to output values; side effects on global state and subprogram internal state
- May depend on implicit arguments in form of non-local variables

Local Referencing Environments

- Local variables can be stack-dynamic
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Subprogram As Abstraction

- Subprograms encapsulate local variables and specifics of algorithm applied
  - Once compiled, programmer cannot access these details in other programs
  - In most languages subprogram definitions are not executables, but e.g. in Python a function definition is executed to bind the function name in the current local namespace to a function object
- Application of subprogram does not require user to know details of input data layout (just its type)
  - Form of information hiding

Basic Definitions

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not necessarily the body, of the subprogram
- A *formal parameter* is a (dummy) variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement
- A *subprogram definition* provides the body, of the subprogram and may provide the protocol

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
  - E.g. in C# PrintOrderDetails("Gift Shop", 31, "Red Mug");
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names
  - E.g. in C# PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated

- Variable numbers of parameters
  - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
  - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
  - In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Subprogram Parameters

- Formal parameters: names (and types) of arguments to the subprogram used in defining the subprogram body
- Actual parameters: arguments supplied for formal parameters when subprogram is called
- *Actual/Formal Parameter Correspondence:*
  - attributes of variables are used to exchange information
    - Name – Call-by-name
    - Memory Location – Call-by reference
    - Value
      - Call-by-value (one way from actual to formal parameter)
      - Call-by-value-result (two ways between actual and formal parameter)
      - Call-by-result (one way from formal to actual parameter)

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
  - Require extra storage location and copy operation
- Potential problem: `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Name (Inout Mode)

- By textual substitution
  - (or thunking – i.e. passing a function)
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way

- These two are in conflict with one another!
  - Good programming ➔ limited access to variables, which means one-way whenever possible
  - Efficiency ➔ pass by reference is fastest way to pass structures of significant size
  - Also, functions should not allow reference parameters

Parameters that are Subprograms

- It is sometimes convenient to pass subprogram names or even subprograms as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

- Note this is first class functions or lambdas which is now becoming part of mainstream languages!!

Parameters that are Subprogram Names: Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility
- Java until Java 8 did not allow method names to be passed as parameters
- C# supports functions a parameters through delegates
  - Delegates can now be anonymous or lambda expression
  - We talk about first class functions
- Functional languages supports functions as first class functions

Criteria in a good language design

- The criterias from Sebesta's book are well established "rules of thumb"
- But until recently they had litlle or no research backing.
- Since 2009 a new directions in programming language design research has emerged
  - could be called Evidence based Programming Language Design
  - Use of social science methods

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

What is orthognality?

- “The number of independent primitive concepts has been minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied “orthogonally” in order to maximize the expressive power of the language while trying to avoid deleterious superfluities”
  - Adriaan van Wijngaarden et al., Revised Report on the Algorithmic Language ALGOL 68, section 0.1.2, Orthogonal design

What is orthogonality?

- "A precise definition is difficult to produce, but languages that are called orthogonal tend to have a small number of core concepts and a set of ways of uniformly combining these concepts. The semantics of the combinations are uniform; no special restrictions exist for specific instances of combinations." – David Schmidt
  - Ex:
    - A[4+(F(X)-1)] OK in Algol but not in Fortran IV
    - Pascal, only values from the scalar types can be results from function procedures. In contrast, ML allows a function to return a value from any legal type whatsoever.

What is lack of orthogonality?

- The C language is somewhat inconsistent in its treatment of concepts and thus not as orthogonal as it could be
- Examples of exceptions follow:
  - Structures (but not arrays) may be returned from a function.
  - An array can be returned if it is inside a structure.
  - A member of a structure can be any data type
    - (except void, or the structure of the same type).
  - An array element can be any data type (except void).
  - Everything is passed by value (except arrays).
  - Void can be used as a type in a structure, but a variable of this type cannot be declared in a function.

Tennent's Language Design principles

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions
- The Principle of Correspondence
  - Declarations ≈ Parameters
- The Principle of Data Type Completeness
  - All data types should be first class without arbitrary restriction on their use

–Originally defined by R.D.Tennent

Principle of correspondence

- Example in Pascal:

```
var i : integer;
begin
  i := -j;
  write(i)
end
```

and

```
procedure p(i : integer);
begin
  write(i)
end;
begin p(-j) end
```

- Are equivalent

Example of missing correspondence

In Pascal:

```
procedure inc(var i : integer);
  begin
    i := i + 1
  end;

var x : integer;
begin
  x := 1;
  inc(x);
  writeln(x);
end
```

No corresponding declaration

However C has correspondence

```
void inc(int *i) {
  *i = *i + 1;
}

int x = 1;
inc(&x);
printf("%d", x);

int x = 1;
{
  int *i = &x;
  *i = *i + 1;
}
printf("%d", x);
```

The Concept of Abstraction

- The concept of abstraction is fundamental in programming (and computer science)
- Tennents principle of abstraction
  - is based on identifying all of the semantically-meaningful syntactic categories of the language and then designing a coherent set of abstraction facilities for each of these.
- Nearly all programming languages support process (or command) abstraction with subprograms (procedures)
- Many programming languages support expression abstraction with functions
- Nearly all programming languages designed since 1980 have supported data abstraction:
  - Abstract data types
  - Objects
  - Modules

Cognitive Dimensions

- Developed by Thomas Green, Univ. of Leeds
- Used to analyze the *usability of information artifacts*
- Applied to discover useful things about usability problems that are not easily analyzed using conventional techniques
- Framework (as opposed to model or theory)

Cognitive Dimensions (2)

- Focused on ***notations***, such as
  - Music, Dance
  - Programming languages

- And on ***information handling devices***, such as
  - Spreadsheets
  - Database query systems
  - IDEs

- Gives descriptions of aspects, attributes, or ways that a user thinks about a system, called dimensions

- The 14 dimensions (and more have been added)

Dimensions

- Abstraction
  - types and availability of abstraction mechanisms
- Hidden dependencies
  - important links between entities are not visible
- Premature commitment
  - constraints on the order of doing things
- Secondary notation
  - extra information in means other than formal syntax
- Viscosity
  - resistance to change
- Visibility
  - ability to view components easily

Abstractions

- ***Types and availability of abstraction mechanisms***

- ***An abstraction is a class of entities or grouping of elements to be treated as one entity*** (thereby lowering viscosity).
- Abstraction barrier:
  - minimum number of new abstractions that must be mastered before using the system (e.g. Z)
- Abstraction hunger:
  - require user to create abstractions

Abstraction features

- Abstraction-tolerant systems:
  - permit but do not require user abstractions (e.g. word processor styles)
- Abstraction-hating systems:
  - do not allow definition of new abstractions (e.g. spreadsheets)
- Abstraction *changes the notation*.

Abstraction implications

- Abstractions are hard to create and use
- Abstractions must be maintained
  - useful for modification and transcription
  - increasingly used for personalisation
- Involve the introduction of an *abstraction manager* sub-device
  - including its own viscosity, hidden dependencies, juxtaposability etc.

Hidden Dependencies

- Important links between entities are not visible
- *Examples:*
  - class hierarchies
  - HTML links
  - spreadsheet cells

Secondary Notation

- Extra information in means other than formal syntax
- *Examples:*
  - Comments in programming languages
  - Pop-up boxes for icons
  - Well-designed icons

Viscosity

- Resistance to change
  - Fixed mental model
  - Hard-coded structure
- *Examples:*
  - Technical literature, with cross-references and section headings (because introducing a new section requires many changes to cross-references)

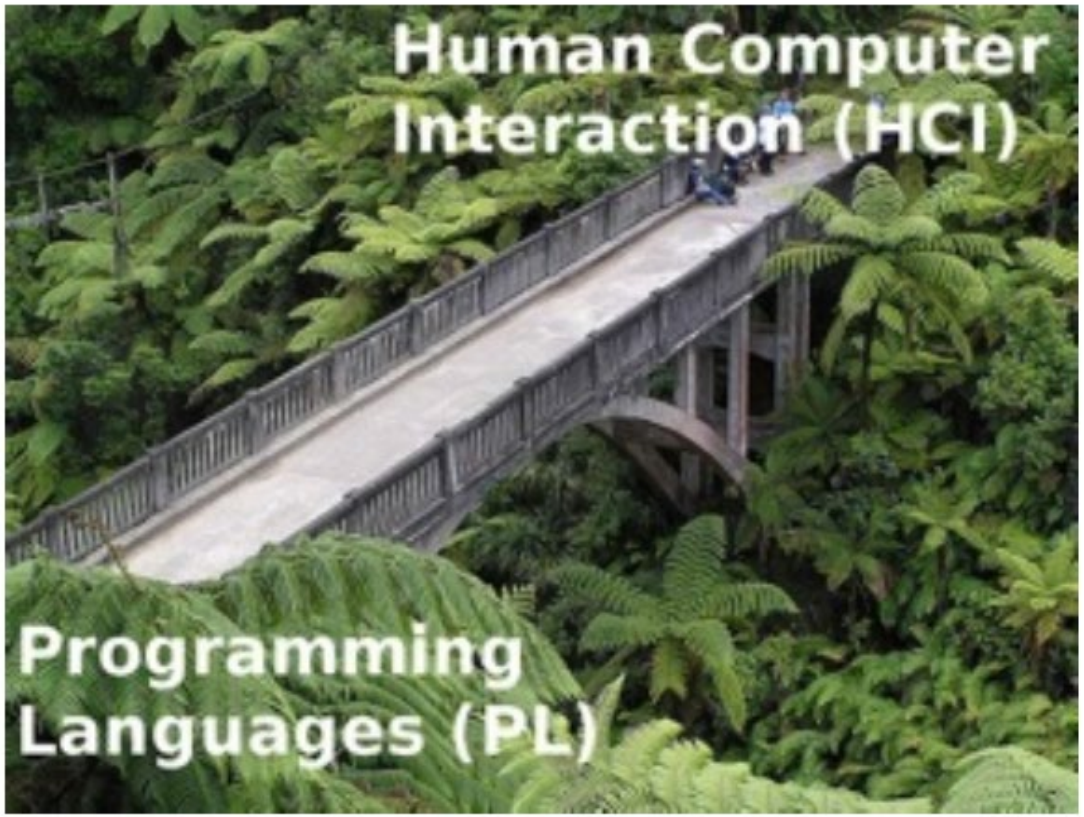
Further Dimmensions

- Closeness of mapping
  - closeness of representation to domain
- Consistency
  - similar semantics expressed in similar forms
- Diffuseness
  - verbosity of language
- Error-proneness
  - notation invites mistakes
- Hard mental operations
  - high demand on cognitive resources
- Progressive evaluation
  - work-to-date checkable any time
- Provisionality
  - degree of commitment to actions or marks
- Role-expressiveness
  - component purpose is readily inferred
- And more …
  - several new dimensions still under discussion

Supplementary Material

- Cognitive Dimensions of Notations website www.cl.cam.ac.uk/~afb21/CognitiveDimensions
- 10th Anniversary CD of Notations Workshop www.cl.cam.ac.uk/~afb21/CognitiveDimensions/workshop2005/index.html

PLATEAU - ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools



Programming Language design

- Designing a new programming language or extending an existing programming language usually follows an iterative approach:

1. Create ideas for the programming language or extensions
2. Describe/define the programming language or extensions
3. Implement the programming language or extensions
4. Evaluate the programming language or extensions
5. If not satisfied, goto 1

Discount Method for Evaluating Programming Languages

1. Create tasks specific to the language being tested - tasks that the participants of the experiment should solve. Estimate the time needed for each task (max 1 hour)
2. Create a short sample sheet of code examples in the language being tested, which the participants can use as a guideline for solving the tasks.
3. Prepare setup (e.g. use of NotePad++ and recorder) and do a sample test with 1 person.
   - Adjust tasks if needed
4. Perform the test on each participant, i.e. make them solve the tasks defined in step 1. (Use approx. 5 test persons)
5. Each participant should be interviewed briefly after the test, where the language and the tasks can be discussed.
6. Analyze the resulting data to produce a list of problems
   - Cosmetic problems, Serious problems, Critical problems

Discount Method for Evaluating Programming Languages

- Method inspired by the Discount Usability Evaluation (DUE) method and Instant Data Analysis (IDA) method
- Reference:
  - Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen.
  - Discount method for programming language evaluation.
  - In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016). ACM, New York, NY, USA, 1-8. DOI: https://doi.org/10.1145/3001878.3001879

What can you do in your project now?

- Design abstractions
  - Functions and/or Procedures or ..

- Evaluate your language design
  - Revisit Sebesta's design criteria
  - Tennent's principles
  - Cognitive dimmensions
  - Discount Method for Evaluating Programming Languages