1 GOALS AND PROCESS FOR DESIGNING A LANGUAGE

1.1 Simplicity

Useful especially eases parsing the input and the language implementation. Simplicity generally derives from regularity in the programming language.

The principle of regularity is nicely defined by Maciennan in his book, “Principles of programming languages: design, evaluation and implementation”, 3rd edition, Oxford University Press, 1999. Maciennan states that a programming language should have regular rules. His regularity principle is reproduced below.

Definition 1.1 (Regularity Principle) “Regular rules, without exceptions, are easier to learn, use, describe, and implement.”

1.2 Generality

Since it aids simplicity. Generality can be further subdivided into three principles: orthogonality, 0-1-infinity, and consistency. Each of these principles is defined by Maciennan as follows.

Definition 1.2 (Orthogonality Principle) “Independent functions should be controlled by independent mechanisms.”

Exercise 1.1 Are constructors orthogonal in the Java programming language?

Definition 1.3 (0-1-infinity Principle) “The only reasonable numbers are zero, one, and infinity.”

Exercise 1.2 Does inheritance relationship between classes in the Java programming language satisfies 0-1-infinity principle? What about that in the C++ programming language?

Definition 1.4 (Consistency Principle) “Similar things should look similar, different things different.”

Exercise 1.3 Is the use of keyword const consistent in C++?

1.3 Ability to manipulate programs as data

An “isolated program” is a myth commonly found only in introductory programming tales. Most programs in the real world interoperate with other programs that may be written in different programming languages (e.g. VB and C# on the .NET platform), may be running on different language runtime environment (e.g. a Java program calling native function written in C or C++), may be running in entirely different process on the same or on a different machine across the world (e.g. remote procedure calls, web service calls, etc). It is for these reasons one must be able to deal with programs as data so that one can deal with there interoperability issues. For example, in the Java programming language when calling a native method with certain arguments, your program must be able to process the data type of the argument to be able to convert it to a form understandable by the native environment. This is precisely what the Java Native Interface (JNI)
does and it uses the Java programming language’s facility to manipulate program as data called reflection. More details on this later.

Second common reason to have this property is to be able to create self-modifying code (also self-adaptive or self-healing software).

Finally, this property eases construction of interpreters and compilers to allow "meta programming". We want to be able to manipulate programs and data easily (meta-programming, generic programming).

Exercise 1.4 What is meta-programming?

The discipline of writing programs that represent and manipulate other programs or themselves.

Exercise 1.5 What are meta-programming features in Java and C#?

• reflection
• generics
• metadata annotations

1.4 Process

We will operate by consensus, if necessary we will vote, or I will select if there’s an arbitrary choice to be made.

2 PARTS OF A LANGUAGE

• means of computation
  – primitive expressions (data)
• means of combination
  – procedure calls, sequencing, loops, container data structures
• means of abstraction
  – procedures, abstract data types, macros

3 DATA OR MEANS OF COMPUTATION

The key question considered in this i.e., means of computation, primitive expressions

Exercise 3.6 What should be the types in the language?
First, there is a basic difference between primitive types and compound types. Primitive types cannot be broken down further, they are like atoms. Compound types can be thought of as molecules. Consider the following primitive types and compound types as examples.

<table>
<thead>
<tr>
<th>primitive types</th>
<th>compound types</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>vector (arrays)</td>
</tr>
<tr>
<td>char</td>
<td>(linked) lists</td>
</tr>
<tr>
<td></td>
<td>ports (files)</td>
</tr>
<tr>
<td>symbol</td>
<td>stream</td>
</tr>
</tbody>
</table>

We may also need: strings, boolean, procedures.

**Exercise 3.7** How do we know whether we have enough types?

(Time and space efficiency, completeness) sum and product types lists as able to simulate both

**Exercise 3.8** How does simplicity affect the list of types?

key test is simulation: one type be used to simulate another (or vice versa)?

### 3.1 Simulation, Generality

**Definition 3.1** A type $T$ can simulate a type $U$ if using $T$ one can get the same behavior as $U$.

Behavior includes: functionality (inputs, outputs) size (space used) time (complexity of running ops)

**Exercise 3.9** How do we know whether we have too many types?

(User-defined types, simplicity of numeric types)

**Exercise 3.10** What types should be built-in?

Depends on their utility, in efficiency, portability, and whether they need to be used in means of combination (such as Booleans).

### 4 Type Checking and Type Errors

**Exercise 4.11** What is a type error?

**Exercise 4.12** Should there be a static type system, or should there be run-time type checking?

**Definition 4.1** Dynamic type checking means type errors are detected (in general) at run-time.

**Definition 4.2** Static type checking means type errors are detected before the program is run.
Exercise 4.13 Which is better for users?

Exercise 4.14 Which is more flexible?

(+ 3 (if (zero? 0) 4 'done)) ; illegal in typed languages

Exercise 4.15 Which suits interpreters better?

Exercise 4.16 Which gives better performance?

4.1 Operations

Exercise 4.17 What literals, constructors, and observers are needed?

Consider the following types and the operations allowed on them as an example.

<table>
<thead>
<tr>
<th>operations</th>
<th>literals</th>
<th>special forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>+, -, *, /</td>
<td>1, 1.2, 5.7e4</td>
</tr>
<tr>
<td>char</td>
<td>&lt;, &gt;, &lt;=,...</td>
<td>'a', 'b'</td>
</tr>
<tr>
<td>vector</td>
<td>1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>symbol</td>
<td>#asym</td>
<td></td>
</tr>
</tbody>
</table>

Exercise 4.18 Are any special forms (e.g., &&, || in Java) needed?

5 MEANS OF COMBINATION SYNTAX

5.1 Syntax of expressions

Exercise 5.19 What should the syntax be for expressions?

- consider parsing things like 3 + 4
- parsing, parse trees, operator precedence (high, low)
- recall that an additional goal is to be able to manipulate programs as data

Exercise 5.20 How can we represent parse trees as linear text?
Exercise 5.21  What data structures do we have that would work?

5.2  Commands and combinations of commands

Exercise 5.22  What commands do we need?

Assignment, sequencing, if-then-else, perhaps other special forms

Exercise 5.23  What is an expression language?

Exercise 5.24  What is a statement language?

6  MEANS OF ABSTRACTION

6.1  Naming

Exercise 6.25  How should we name expressions and procedures?

Exercise 6.26  Do we want to have local declarations within expressions?

6.2  Parameterization and function declarations

Exercise 6.27  How should we declare parameters for procedures?

Exercise 6.28  Can we unify the two kinds of declarations?