Proceedings of the Fourth Symposium on Programming Languages and Software Tools
Visegrád, Hungary, June 9-10, 1995.

Remarks on Language Extensibility

Jaanus Pöial

Department of Computer Science, University of Tartu
e-mail: jaanus@cs.ut.ee

Abstract

Programming languages are often characterised proceeding from the paradigm they are intended to support – imperative, functional, logic, object-oriented, parallel, real-time, distributed, event-driven etc. Independently from the paradigm there exist some other important aspects of the language design and implementation like user interface of the system, choice of integrated tools, support to incremental development, whether the system is interpretive or compiles into some external representation (machine code, assembly language, higher level language), support to standalone application development (incl. GUI development), portability, extensibility, compatibility with different standards and platforms etc. These additional qualities may appear in concrete implementation but may also be a part of the language itself.

This paper focuses on **language extensibility** that is the notion with wide range of interpretations. Authors opinions are based on experience of using and teaching the "minor languages" Forth [2] and Tcl/Tk [6] as well as "traditional" ones [4].

The first part gives a short overview about the evolution of different types of extensibility.

The second part is devoted to the problems of implementation of extensible languages.

The third part contains some remarks on programming in extensible languages.

As a conclusion some similarities between extensible languages and the language of mathematics are outlined:

- Freedom to create abstractions and notations to express these abstractions.
- The language itself is not "high level" or "low level" – it is always on the level of the Creator.

Categories and Subject Descriptors:
D.3.2 [Programming Languages]: Language Classifications;
D.3.0 [Programming Languages]: General ;
D.1.0 [Programming Techniques]: General;

Key Words and Phrases: Programming languages

1 Evolution of Language Extensibility

Let us start from the history of programming languages and have a look at different types of language extensibility.

Probably the first idea was to introduce subroutines to perform common tasks by calling the same code from different places (like GOSUB in Basic). To individualise each call the parameters were introduced (like functions in FORTRAN). Through many languages the concept of **procedural extensibility** has been deposited (procedures, functions, passing parameters, ...) that allows a programmer to extend the language with new operations.

Evolution of data types has a long history also. It started from predetermined data structures (FORTRAN, Basic), led to the idea of programmer-defined data structures (like in PL/I) and became stable in early typed languages like Pascal, Ada. Let us call this concept the **extensibility with data types** that allows a programmer to extend the language with new (passive) data types. Unfortunately, the possibilities of creating totally new abstract objects are restricted using this approach. A programmer may describe his/her own record structure in Pascal, but cannot introduce an associative array, for example. It is natural to expect a way of creating the **abstract data types** in an extensible language.

The next group of tools supporting the language extensibility is **macroprocessors** that allow to introduce substitutions to the program text. More sophisticated macroprocessors have their own powerful command languages comparable to the programming languages. The borderline between the programming languages and macro languages is sometimes quite fuzzy – in case of the Forth language, for example, the whole language may be considered as a macro language (performing actions during compilation phase, not changing the program text).

An important feature of any programming language is the way it handles the control flow. The first choice is to provide a fixed set of possibilities determined by the syntax of the language. We may distinguish between unstructured approach (GOTO-based) and structured approach (control structures). Even if the set of control structures is "rich" enough such an approach does not allow to introduce implementation-specific abstractions related to the control flow. The second choice is not to give direct access to the control flow at all – programmer works on more abstract level and it is up to the compiler to generate corresponding program (declarative style – functional and logic programming). Unfortunately, there are many cases when such an approach is inconsistent. Problems of exception handling (e.g. ON-statement in PL/I) and processing interrupts have led to the dynamic style of programming the control flow – event driven model (like in Tcl/Tk or in object oriented systems).

All these different concepts of handling the control flow ("procedural", "functional", "object oriented", ...) are usually not available in a particular language except languages which provide the **extensibility with new control structures** (in more general sense).

Speaking of abstractions related to the modularisation and control we cannot bypass **higher order functions**. This concept became natural in functional languages. In more general it is the question of processing code as data (e.g. passing functions as parameters). Extensible language should support introduction of higher order functions.

The same piece of text (or even code in dynamic case) may have different meanings when put into different context. This feature (sometimes called **context switching**) becomes a powerful tool when a programmer has some mechanisms in a language to manipulate it. Examples: static – Forth vocabularies, dynamic – `send`-command in Tcl.

Up to now we have listed the features which support techniques of abstraction but extensibility may as well have an opposite direction – **extensibility with low level**

operations. This is a problem with "very high level" languages and even with traditional languages to enable an access to the hardware level. Adding a new driver, for example, is often an insurmountable task.

The following properties are not directly related to the notion of extensibility but are important to mention also:

– **Modularity**. Does the language support grouping of abstractions to well-defined and manageable units – "modules", "packages", "classes", etc.

– **Reusability** of resources. Does the language enable to "avoid programming".

Theoretically it is possible to implement nearly everything in most existing languages – in many cases this seems to be a matter of taste. However, the problems with different languages vary from "small things" (like procedures as parameters) to basic philosophical questions about balance between freedom and safety, influence of the language to the way of thinking etc. The main drawback of completed languages (typically having only few types of extensibility) is their categorical compulsion to make a programmer thinking "right way" predicted by the author(s). Instead of providing flexible tools to create problem oriented abstractions these languages often suppress reasonable solutions and insist on tricks to bypass the unpredictable problems.

2 Extensibility and Language Implementation

Often the following scheme for language implementation is chosen

$$\textit{Language} \Leftrightarrow \textit{Abstract machine} \Leftrightarrow \textit{Real machine}$$

Such an approach induces two stages of translation. Relatively portable stage is the translation from "sugared" language to the commands of an abstract machine. The implementation stage of the abstract machine is more hardware-dependent and may in turn contain several layers.

Extensibility of the language reflects on both translations. The first possibility is to have a fixed set of "commands" for the abstract machine. In that case the second translation is also fixed and all types of extensibility are handled during the first stage. The second possibility is to share responsibility – abstract machine itself is extensible and a programmer extends both the language and the machine. In that case the abstract machine must not be too abstract initially – much better is to let a programmer feel that he/she has an access to all "real" resources.

Let us have a look at advantages and drawbacks of these two methods.

1. Fixed abstract machine

   – Easier to compile.

   – Programmer has to think on the language level only.

   – Safer programming.

   – Less possibilities for extensions.

2. Extensible abstract machine

   – Harder to compile the programs, usually interpretive implementations that give short response time during development but cause problems with turnkey applications.

– Programmer has to think at least on two levels – how to design a suitable abstract machine and how to express the needed concepts in terms of this machine.

– More freedom and more responsibility.

– Programmer needs more time to learn to use this freedom and to think on different levels in a disciplined manner. On the other hand – once getting used to it there is no way back. Even if there are remarkable constraints on the environment, hardware or application there should be no constraints on programmers freedom to choose appropriate abstractions.

There are examples of both approaches in existing languages, e.g. Pascal and Forth. The question is whether the drawbacks of the second approach are important enough to resign. It depends on traditions, skills and habits of programmers, fields of application etc.

Let us list some important properties of the language from the viewpoint of extensibility to conclude the implementation-related issues.

1. Clean and "simple" concepts for both the language and an abstract machine. Simplicity here means that there are no exceptions in the language (maybe at most one or two – but this is already a sign of bad design), that the number of basic concepts is small (at most 5 – 7) and that the way of handling these concepts is consistent.

2. Minimal syntax is an advantage that makes it easier to find notations for extensions introduced. This is not a problem for the compiler to handle the "rich" and complicated syntax, but this is a problem for a programmer to orient oneself in it and to follow the rules of **syntactic extensibility**. Laconic style ("high density" languages) supports modularisation and makes language safer.

3. Giving a programmer possibility to extend the abstract machine actually means that a programmer is given full access to the compiler. It is important to decrease the probability of causing disaster through this access. To achieve certain safety the compiler interface (e.g. "access to dictionary") must be "simple" and well-defined.

4. Re-usability of resources is an important feature that should be supported when implementing a system. There is a problem with interpretive languages which usually do not provide precompiled or dynamic libraries or provide implementation-specific non-portable libraries. The only resource that is portable is the program text (in case of choosing the right level of abstractions).

5. Even if the system is interpretive there should be a possibility to compile turnkey applications using some kind of **post-compilation**.

6. Success of an extensible environment depends on attractive extensions provided with the system. It is not obvious how to gain the best results when nearly "everything is possible". Tcl/Tk, for example, owes its popularity to Tk toolkit that contains extensions for X-windows programming. Didactic example (from the field of text processing) is the Emacs editor – a powerful tool but not very well designed in providing "defaults".

Interpretive implementations are "cheaper", give quick response time during development and provide more flexibility. Compilers usually appear at the later stage when concepts are stabilised and there is a market for the language.

3 Programming with Extensible Languages

The author has some experience in teamwork at large applications using the Forth language – compiler compiler ([10]), compilers for Fortran and Modula-2, database applications etc. The nature of compiler construction field determines the need for rich variety of abstractions one can find in programming languages as well as tools for writing compilers (see also [3]). Programming a compiler sometimes introduces five levels of thinking:

– run-time behaviour of an application on the target machine
– mapping $host \rightarrow target$ (cross-compilation)
– resulting program on the host machine ("abstract code")
– compile-time behaviour on the host machine (context checking and code generation)
– translation of the initial program text (translation scheme).

This example demonstrates the complexity of abstractions in a particular area where it is not easy to use traditional methods. Always when the problem is complex and experimenting is needed to find innovative solutions an extensible approach is preferred (we do not solve a problem but create abstractions to solve a group of problems). For concrete problems the "safe" language seems to be a better choice.

The main drawback of an extensible approach is putting all the responsibility on a programmer who needs to be an expert. Even if this is the case some additional tools are needed to gain safety in managing all levels of "internals". For the compiler compiler and the Forth language some research is carried out in [8, 9]. There is a danger to "freeze" and fix too many things when designing supporting tools for extensible languages. In general this is a problematic issue to fix right things.

Another topic of large discussions is the standardisation that is more complicated in case of extensible languages because even the object of standardisation is not obvious. This work is nearly finished for the Forth language for now. Unfortunately, this standard is far from being ideal (e.g. the model of extensibility). To write a portable application in an extensible language a programmer should first design good layers of (abstract) operations. Following the standard comes second (but is important, nevertheless). Sometimes standards change faster than an application and these changes should touch as few layers as possible.

The main advantage of extensible languages from the viewpoint of a programmer is correspondence between his/her skills and the level of the language. As a rule extensible languages are not "high-level" or "low-level" – they are on the level of their users. Having a look at the Forth language again we can find an object-oriented approach ([7]), logic and functional approaches ([1, 5]), an abstract data type approach etc. but also the "assembler" approach, for example. On the other hand – bad style in extensible languages is much worse than bad style in traditional languages.

If the language is too high-level for a programmer initially, he/she may have difficulties to use it efficiently. It is better if the programmer "grows up" together with the language and does not bump against restrictions when the level of abstraction increases.

The way of thinking propagated in this article is the mathematicians way of solving problems that is not always applicable. Nevertheless, there are similarities between extensible languages and the language of mathematics: both provide freedom to create abstractions and notations to express these abstractions.

References

[1] Belinfante, J.G.F.: *S/K/ID: Combinators in Forth*, Journal of Forth Application and Research, Vol.4, No.4, 1987.

[2] Brodie, L.: *Starting Forth*, Prentice-Hall, 1987.

[3] Dixon, R.D.: *Embeddings of Languages in Forth*, Journal of Forth Application and Research, Vol.4, No.4, 1987.

[4] Fischer, A. E., Grodzinsky, F. S.: *The Anatomy of Programming Languages*, Prentice-Hall, 1993.

[5] Odette, L.L.: *Compiling Prolog to Forth*, Journal of Forth Application and Research, Vol.4, No.4, 1987.

[6] Ousterhout, J. K.: *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[7] Pountain, D.: *Object-Oriented Forth*, Academic Press, 1987.

[8] Pöial, J.: *Formal Semantics of Parameter Passing for Forth-programs with Control Structures*, Proceedings of the Second Symposium on Programming Languages and Software Tools, Aug. 21 – 23, 1991, Pirkkala, Finland, University of Tampere, Department of Computer Science, A-1991-5, 1991, August, 48 – 54.

[9] Pöial, J.: *Some Ideas on Formal Specification of Forth Programs*, 9th euroFORTH conference on the FORTH programming language and FORTH processors, Oct. 15 – 18, 1993, Marianske Lazne, Czech Republic, 1993, 4 pp.

[10] Tombak, M., Soo, V., Pöial, J.: *A Forth-Oriented Compiler Compiler and its Applications*, FORTH Dimensions, Vol XVI No 5, Jan.-Feb. 1995, (ISSN 0884-0822), Forth Interest Group, Oakland, USA, 21 – 22.

Postal address

Jaanus Pöial
Department of Computer Science
University of Tartu
J. Liivi St., 2 – 310, Tartu, EE2400
Estonia