Rapidly implementing languages to compile as C++ without crafting a compiler

Anthony Savidis*

Institute of Computer Science, Foundation for Research and Technology—Hellas, Heraklion, Crete, GR-70013, Greece

SUMMARY

We present a heuristic implementation method for programming languages that is appropriate when the following requirements or conditions are met: (a) there is a need for very rapid development of a language with significant performance demands, while exploiting a comprehensive host language and/or library; (b) need of customized runtime execution environment supporting of execution tracing and visualization instruments; (c) the facilitation of hybrid code exists, such as mixing with the native language, and there is the potential for allowing multiple languages to be deployed concurrently in the same program. This may account for a wide range of domain-specific languages such as learning-oriented, scripting, assembly style, data manipulation, rule-based, or event languages. The proposed technique is presented for the C++ language, supporting the development of languages whose source programs compile as C++ code. The software architecture shifts from the tradition of lexical analysis, syntax-directed translation and code generation, and we propose a complementary, conditionally advantageous, heuristic development paradigm. The method has been applied to the development of a high-level imperative language, an assembly language and a functional language, which are all currently deployed for teaching purposes. Copyright © 2007 John Wiley & Sons, Ltd.

Received 24 July 2006; Revised 13 December 2006; Accepted 1 February 2007

KEY WORDS: rapid language development; domain-specific languages; embedded programming languages; syntax emulation

1. INTRODUCTION

The design and implementation of programming languages is well known to be a highly demanding task. Theoretically, although the notion of programming is related to languages that can support the description of algorithms, the term is loosely deployed to also refer to domain-specific or purpose-oriented languages not actually tied to algorithm implementation. The method reported here

*Correspondence to: Anthony Savidis, Institute of Computer Science, Foundation for Research and Technology—Hellas, Heraklion, Crete, GR-70013, Greece.
†E-mail: as@ics.forth.gr

Copyright © 2007 John Wiley & Sons, Ltd.
concerns the implementation of languages in general, when the following demands and conditions are met:

- there is a need for a very rapid implementation process;
- there are considerable runtime performance demands;
- the language is built on top of a comprehensive domain-specific library;
- there is a requirement for various execution tracing and visualization tools;
- modification of the language syntax is allowed for development needs (i.e. the language syntax design is quite flexible);
- there is a customized runtime execution environment (i.e. one not necessarily following the typical programming language traditions).

In this context, we will present our method by discussing the way it has been applied in the rapid implementation of two special-purpose programming languages, the implementations being publicly available (see the acknowledgements): an imperative programming language with a metaphoric computation model (named FLIP), and an assembly language for a virtual processor (named JET). Both languages are currently deployed for the introductory programming courses of computer-science students [1]. The introduction continues with an overview of the proposed method, followed by a brief discussion of the ‘language in a language’ concept. Then, we quickly elaborate on the two key subject scenarios, and enumerate the main contributions resulting from the reported work.

1.1. Overview of the methodology

The approach is based on the implementation of the language syntax in such a way that its programs pass as legal C++ source, without necessitating a pre-compilation phase. In practice, this is only feasible once a language-specific header file is provided, which, if included with the original source program, turns it into a directly legitimate C++ program. This concept is illustrated in Figure 1; such a header file essentially plays the role of a language front-end, internally linking to the implementation of the language semantics, runtime environment and runtime library.

At a first glance, one might assume that such a header file merely encompasses a series of preprocessor-based bindings (i.e. macros), so that the syntactic elements of our source language are effectively mapped to appropriate code snippets in the target language (hereafter always denoting C++). Indeed, preprocessing-based syntax emulation may work well for a number of very simple languages. However, owing to the fact that, via preprocessing, the syntactic mapping of the source to the target language relies on name matching and textual substitution, the technique by itself is severely limited as it makes source languages that syntactically and semantically deviate a great deal from the target language. In the proposed method, we exploit the following programming elements of C++, to accommodate a broader set of syntactic patterns, enabling the automatic transformation of source-language syntactic units to target-language eligible syntax:

- preprocessor techniques;
- template classes and functions;
- operator and function overloading;
- temporary instances in expressions;
- hidden instances within blocks and classes.
The idea is that we take every potential syntactic unit of the source language and we try to implement it in the target language through combinations of the above elements. In this iterative process, we exhaustively investigate the chances for implementing the syntactic unit as it is, so that it becomes a pure C++ code unit. Upon failure, i.e. after inconclusively applying multiple iterations, some particular grammatical properties of the source language are identified as being potentially inappropriate for this type of transformation, consequently leading to targeted syntax refinements. It is important to note that the failure condition of this heuristic process does not constitute any formal criterion, since it does not imply that ‘mapping is impossible’, but loosely that ‘mapping is improbable’. Naturally, a key factor for the success of this process is the programmer’s knowledge and capability to effectively manipulate the elements of the target programming language so as to accommodate (i.e. to program) the source code syntactic patterns. In this context, we will provide various types of such advanced syntactic transformations, which, collectively, realize an effective implementation toolbox to cope with the challenging issue of syntactic mapping. Finally, we also discuss the issue of grammar adaptation.

1.2. Language in a language concept

The method presented reflects a type of encapsulated or embedded development of one language within another language. Conceptually, the target language, on which the source language is actually mapped, constitutes a type of host language: the source language represents an infinite subset of legal subprograms of the target language. This relationship deserves particular attention, since it does not imply that the source language is implemented in the target language in a way typical for traditional compiler or interpreter development. Instead, it displays a different relationship, following which any source program, with its source language front-end, owing to the syntax and semantics of the target language, becomes ‘as it is’ a plausible target language program, meaning the source language is emulated in the target language. Consequently, the compiler, debugger and the overall development
environment of the target language are directly deployable for the new source language. The ‘language in a language’ concept has been exercised before [2], but has not been emphasized as a disciplined, prescriptive and integrated way for language delivery; in this paper we focus on the latter.

1.3. Subject language cases

The main subject cases concern two domain-specific languages, primarily intended for teaching purposes. The first language named FLIP (Front-end Language for Introduction to Programming), is an imperative procedural language with a computational model (i.e. runtime environment) that is essentially a real-world metaphor: a human processor using as memory a normal physical notebook organized into numbered pages and rows. Details of this metaphoric model and its appropriateness for introductory teaching of imperative programming may be found in [1]. The underlying environment and memory model of the FLIP language is well defined, made explicit to FLIP programmers by being part of the language semantics, clearly deviating from typical general-purpose imperative languages, owing to its learning-oriented nature.

- Memory model:
  - direct access is allowed to the language, indexing by page and row, while memory cells are dynamically typed (i.e. not committed to a single type);
  - all program memory, including temporaries for expressions, take notebook memory in a well-defined deterministic manner.

- Types: these are common built-in scalar types, all having as storage requirements a single notebook row, with range-validated arrays and address-validated pointers, as pairs of a page and a row number.

- Variables: these are dynamically typed as synonyms of a single memory cell or a contiguous cell region, or strongly-typed variables associated to built-in types.

- Control: this has a block structure, with loops, branch statements, subprograms (functions and procedures) and an enhanced case statement.

Apparently, many of the previous features have their counterparts in existing imperative languages. However, the key differentiation is in the fact that the operational semantics reflect the way the store (memory) is read/written by the expressions of a language. Hence, besides the necessary syntactic emulation, the implementation of the language semantics is also required, following the metaphoric language store model. In addition, in order to provide a gateway for the execution tracing and visualization tools, the encapsulation of extra code has to be appropriately carried out, triggering the corresponding execution events. Sample code in the FLIP language, with comments briefly explaining the associated semantics, is provided in Figure 2.

The second language is an assembly language named JET for a hypothetical (virtual) processor. The birth of this language was dictated by the need to introduce students to machine programming in a stepwise fashion, via a simple assembly language offered with successive incremental versions, allowing students to write, execute and trace their programs; from the development point of view, we wished to avoid the implementation of an interpreter/compiler and debugger from scratch. Typical characteristics of the JET language are:
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

Figure 2. Sample code in the FLIP language.

```
USEPOSITION(x) (0,0); // Synonym for notebook cell (0,0)
INTEGER y; // Takes first free cell, i.e. (0,1)
PROGRAM_BEGIN
  INPUT(x); OUTPUT(x); // x dynamically typed; value type may vary
  OUTPUT(POSITIONOF(y)); // Displays y's location in the notebook
  y = x; // Succeeds if x's value was an integer
PROGRAM_END
```

Figure 3. Sample code in the JET language.

```
CONSTANT(TEN, 10)
CONSTANT(ONE, 1)
VARIABLE(x, 2) // TEN, ONE and x are 32-bit integers
PROGRAM_BEGIN
  LABEL(Loop) // Next instruction is labeled as 'Loop'
  ADD R0, R1, R2
  BRNE R1, TEN, TO(Loop) // A "branch not equal" to 'Loop'
  SETVAR x, TEN // Variable assignment
  INPUT R1 // Registers referred as normal variables
PROGRAM_END
```

- referenceable registers, variables and constants (32 bits—signed integers);
- memory to/from register store/load, with direct and indirect addressing modes;
- arithmetic and branch operations exclusively via registers;
- pseudo input/output commands.

Similarly to the FLIP language, the encapsulation of hooks for the execution tracing and visualization tools was also a prominent requirement. Sample code in the JET language is provided in Figure 3; the genuine assembly-like syntax style is evident.

1.4. Main contributions

The method presented in this paper does not constitute a new theoretical approach to the implementation of programming languages, neither is it an alternative to general-purpose language development paradigm. In this sense, the main contributions do not fall within the foundations of programming languages and compiler construction technology, but on the practical aspects of rapidly accomplishing fully-fledged embedded language implementations, if the latter put forward particular software requirements, such as (a) the need for very rapid development of a language with significant performance demands, while exploiting a comprehensive host language and/or library; (b) a customized runtime execution environment and support of execution tracing and visualization instruments;
the facilitation of hybrid code, such as mixing with the native language and, the potential to allow multiple languages to be deployed concurrently in the same program. If such conditions are contextually met, then the reported approach can lead to the language-delivery milestone in an easy, quick and manageable way, contributing the following techniques to the language development field.

- **Syntax matching, emulation and grammar adaptation.** These prescribe the way specific patterns in the target language (C++) are programmed so that they match the sentences of grammar rules of the source language, realizing emulation of the source syntax analysis via sentences of the target language.

- **Embedded semantic mapping and runtime environment.** The source language runtime environment, including semantics, memory management (when applicable) or library functions, is entirely implemented in the target language as an embedded subsystem. In addition, the semantic translation of the source language is implemented in the form of semantic mapping, through appropriate calls to the runtime environment functions encapsulated within the syntactic patterns.

- **Tracing and visualization hooks.** By exploiting the syntactic patterns that carry lexical information and the semantic calls encapsulated in the syntactic patterns, we hook the generation of execution events grasping lexical context and semantic actions. To allow third-party tools to handle such events, we encapsulate an event infrastructure, so that execution tracing, activity monitoring and content inspection facilities can be easily accommodated.

- **Mixing multiple languages together.** There are cases where during application development multiple languages are combined, with representative examples being Web applications (HTML, scripts, XML, etc.) and video games (engine implementation, animation definitions, character scripting, etc.). Our technique supports a mixed-language style, even allowing mixed source files in a form of hybrid programs. This may be very advantageous in the case when every employed language is essentially designed to ‘optimize’ the programming of a particular aspect of the target application system (e.g. for games where we have animation definitions, state machines for game character intelligence, event handling kernels, etc.).

Also, the proposed method is applicable within traditional compiler development processes, for implementing languages that do not actually require code generation for a real or virtual target processor. Typically, such languages relate to the general family of domain-specific languages (DSLs), although the latter do not constitute a strictly defined language category. Overall we consider as a DSL a language that incorporates domain-reflecting notations, constructs, abstractions and semantics, as part of the prominent design issues. Our method complements traditional approaches in crafting DSLs with an implementation that is embedded in C++. The appropriateness and capability of C++ to serve as the host language for building embedded DSLs has been demonstrated before and is well acknowledged [2]. The most representative example is the popular Blitz++ library [3], supporting array computations through a syntactic abstraction, known to be the first embedded ‘language in a language’ case.

The contribution of the reported work in this context is the delivery of a detailed implementation corpus including: the software architecture, numerous grammar patterns and generic rules for syntactic emulation, linkage to the underlying semantics, and support for execution tracing and visualization. This is practically complementary to traditional compiler construction methods. More specifically, compiler construction is well prescribed in terms of intermediate and target code generation,
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

Figure 4. Typical DSL development (left) with non-standard methods for code emission, and the enhanced approach (right) using our method to prescribe the implementation of code patterns, embedded runtimes and tracing/visualization tools.

optimization and runtime management for real or virtual processors. However, we currently miss any systematic recipes to craft DSLs, besides the traditional implementation approaches for lexical and syntactic analyzers using generator instruments. In such cases, language developers typically implement patterns of the source language in a selected target language, while performing syntax-directed code emission in a heuristic manner, the latter always being dependent on the particular DSL.

When seen in the perspective of traditional DSL development, the reported technique offers (see Figure 4) a systematic way of making code patterns in the source language directly eligible as code snippets of the target language, through grammar emulation of the source language in the target language, while encapsulating calls to the language runtime library (semantics) and supporting hooks to attach execution tracing and visualization tools. Since this addresses mostly the non-prescriptive aspects of DSL development, i.e. code patterns, runtime architecture and debugging support, in comparison to the remaining syntax analysis, the technique effectively complements traditional DSL development. In addition, the method prescribes in detail the way embedded DSLs can be implemented for varying language paradigms, while also supporting, as mentioned before, advanced features such as mixed-language programming (elaborated under Section 7.1), and debugging support. We analyze and discuss three specific language cases of different paradigms, all built with the reported method, by systematically categorizing the various implementation techniques for syntax emulation, semantics and execution tracing, so that programmers can deploy them to craft their own languages.

This paper is organized as follows. First, an account of related work is provided. Then, the implementation architecture of the method is presented and elaborated. The discussion follows with details of the implementation of the language syntax, based on the two language cases, and an example of a functional programming language. Then, the implementation of the semantic aspects is discussed, showing the way the semantic mapping is easily encapsulated within the syntactic patterns. This is followed by a discussion of the execution tracing and visualization support, presenting the
event system and the event generation hooks to support external tracing and visualization instruments. Then a discussion of key issues follows, such as mixing languages and performing grammar adaptations. Finally, key conclusions resulting from real practice and experience with the reported development technique are drawn. In Figure 5 an overview of the detailed techniques reported in this paper is provided.

2. RELATED WORK

The proposed technique of crafting language implementations entirely embedded within the C++ language, requiring a single included header file that provides syntax emulation and linkage to semantic mapping, is not known to be among the existing language development recipes. Although the method may display various generalization principles, we avoid its formulation as a generic strategy applicable to other languages, since, on the one hand, the programming patterns presented are heavily dependent on the C++ language, while, on the other hand, it is far from obvious how those patterns may be mapped to different languages. In this context, we do not emphasize the general aspects of the ‘language in language’ concept, but we focus on the way it is instantiated in a mainstream powerful language like C++. Other work related to the language in a language concept mostly relates to existing embedded
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

language cases, deployment of preprocessor tricks and pre-compilation methods. In this context we provide a brief account, showing the potential cross overs with the present work.

2.1. Embedded DSLs

A well-known example is the Boost Spirit [4] free software library. The method relies on EBNF language emulation in the C++ language, meaning EBNF-like rules of an LL language are defined as normal C++ statements that generate at runtime a recursive descent parser functor object hierarchy, essentially isomorphic to a typical hand-made RDP procedure hierarchy. Through this technique a compiler implementation is actually produced, which is then run to compile programs in the new source language. The implementation of Boost Spirit is based on EBNF-like grammar rules that are compiled as normal C++ source code, literally emulating the EBNF language syntax with a pure C++ implementation. Since EBNF-like source specifications are compiled as C++ legitimate code, this tool clearly reflects ‘a language in a language’ implementation discipline, assuming EBNF is the emulated language. Naturally, the EBNF language itself is quite easy to implement, while there is no support, as there was no need, for hooks to execution tracing tools, and there is a generalization of the method regarding grammar emulation and architecting of the language semantics.

Earlier work concerned the FC++ library [5], i.e. functional C++, revealing in an elegant way the expressive power of C++ for type abstraction, type inference and pattern matching. Although FC++ does not adopt textual substitution with preprocessor techniques, the library clearly establishes a syntactic and semantic transformation layer to allow functional elements in source programs so that they can be compiled directly as legitimate C++. Essentially FC++ is a C++ sublanguage as a form of a higher-level library which, when deployed from within C++ programs, facilitates the practicing of functional programming features, such as lazy evaluation, unnamed functions and polymorphic higher-order functions. In this sense, the technique clearly falls within the general context of this present work; however it does not emphasize larger-scale syntax transformation schemes, semantic variations of the runtime environment and methods to modularly support execution tracing and visualization.

2.2. Preprocessor techniques

While the use of language preprocessors was very popular for a long time and is applicable for a wide range of techniques, following the in-depth review regarding the C preprocessor provided in Ernst et al. [6], there is no reference on the employment of preprocessing for the delivery of severe syntactic variations of the source language. The capabilities merely via preprocessing, literally being text substitution, to implement the thorough syntax of programming languages like the subject cases of our discussion is severely restricted. Today preprocessing methods are considered as inadequate for applying sophisticated syntax transformations. In this context, while we still exploit the traditional ‘goodies’ of the C preprocessor, our technique heavily emphasizes the use of advanced facilities of the C++ language, in particular operator and function overloading for syntax matching, inheritance and templates to built the type system of the source language, inheritance and delegation to attach execution visualizers, and hidden or temporary local instances to trap execution events. Moreover, in contrast to preprocessing techniques typically applicable at a mini scale, the proposed approach is provided as an integrated recipe for entire language development, not only to cope with syntax emulation and transformation, but also to address the delivery of an embedded runtime environment,
and to accomplish the modular implementation of debugging and execution visualization tools. The latter is also justified by the fact that there are no approaches or propositions on the development of complete languages using the C preprocessor exclusively.

2.3. Meta-compilation techniques

Informally, meta-compilation methods are those where a source language is compiled to a high-level target language, thus functionally exploiting the mechanisms of the target language to build the runtime of the source language. The source language may either be embedded in the target language as a tagged code unit, like embedded SQL [7] or lambda-DB [8], or constitute the main user programming language, like early pre-compilers of C++ to C. When compared to our approach, the most apparent drawback is the reduced flexibility in supporting a mixed-language programming style, since interoperability among independent cross-language code units is accomplishable only once an all-in-one pre-compiler for all languages together is developed. However, the most significant contribution of our method lays in the fact that the practicing of meta-compilation methods is radically enhanced for the following key reasons.

- First, once the source language is implemented using the proposed method, there is no need to explicitly implement a code emitter (the component that assembles code patterns together in a syntax-directed fashion, so as to tailor the resulting target code), since the source program is directly eligible as a target language program (in our case it is C++). In addition, because of this there is no need to manage all intermediate representations that aim to play a role in the target code generation. This reduces severely the development overhead, since the meta-compiler development effort is reduced to trivially implementing syntactic and semantic checking via a parser generator, a symbol table and, as applicable, the semantic and type checking logic.

- Second, the proposed method prescribes the detailed software architecture and implementation approach, so as to craft the specific target code patterns and to implement the underlying language semantics, in a disciplined way that can be directly deployed for meta-compiler construction. This is a valuable asset since meta-compilers are not actually supported with explicit implementation protocols for code emission and runtime management (as is the case with traditional compilers for target virtual or real machines).

- Third, debugging and visualization tools can be easily and modularly built, over a modular and scalable notification architecture hooked into the code patterns and the runtime layer. Again, this is a significant enhancement, since meta-compilers are hardly delivered with tracing or visualization support.

- Finally, owing to our technique for integrating grammar emulation in the target code patterns, and the inherent support for independent language runtimes, language mixing is effectively enabled, which, if required, may even facilitate inter-operability among the distinct runtimes (more details on mixed-language programming support are provided under Section 7.1).

In conclusion, meta-compilation, essentially not being a formalized and prescriptive compiler development method, practically constitutes no substitute to the proposed method, but, on the contrary, is arguably enhanced through the adoption of the proposed technique becoming a more effective development strategy.
3. ARCHITECTURE

The proposed implementation method is based on a software architecture with three high-level logical layers (see Figure 6): (a) the language semantics; (b) the language syntax (including all lexical details); and (c) the execution tracing system. Following Figure 6 (above the dotted line), all implementation components that concern the language semantics may be deployed as they are, meaning that it is possible to write code pertaining to the semantics of the source language directly in the C++ language. Semantically, such code could constitute a typical client C++ application, essentially treating the implementation of the source language semantics as a supplier library, while, syntactically, the resulting C++ code would be complicated and quite unreadable as expected. The key implementation layer, making the proposed method feasible, is the implementation of the language syntax, in particular syntax matching, emulation and transformation. This layer reflects a syntactic paradigm shift by providing a source language front-end for programming with the elements of the semantics layer, the latter being implemented in the target language. The role of the semantic mapping is to implement the necessary adaptation on top of the language semantics, so that the latter can be called from within the syntactic patterns. Hence, we anticipate the presence of various calls or references to the semantic aspects inside the syntactic patterns of the source language. Finally, the support for execution tracing and visualization is facilitated through a built-in execution event system, with hooks for event generation appropriately embedded within the syntactic patterns and the semantic implementation. The event system is also accompanied with an extensible API enabling the linkage of external execution tracing and visualization instruments.

4. SYNTAX

We elaborate on the most important syntactic patterns of the two languages. Since FLIP is an imperative procedural language and JET is an assembly language, we mostly focus on some syntactic details that
differentiate between the common practices in their respective language families. The largest part of
the discussion is devoted to the details of the programming patterns to accommodate syntax matching
and direct transformation to the target C++ language syntax.

4.1. Basic grammar rules

The syntax of JET language instructions is illustrated with examples in Figure 7. Instructions do not
have a termination marker, may span across multiple source lines and accept at most three arguments.
Branch instructions accept as the last argument a label specifier, syntactically TO(label id). Uniquely
named labels are issued via LABEL(label id) at any point in the source, but before/after instructions.
The key challenging issue of the JET language syntax, regarding emulation in C++, is the absence of a
termination marker, like the semicolon, and the lack of parentheses surrounding instruction arguments.

The syntax of the FLIP language reflects a few special-purpose semantic characteristics, more
specifically, the support of a metaphoric memory model, accompanied by a memory allocation policy
for variables that is part of the documented language semantics (see Figure 8). In particular, the memory
is a notebook organized into numbered pages and rows. Programmers may directly refer to a notebook
cell via POSITION(page, row) or may alternatively issue symbolic names for direct memory cells
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

via \texttt{USEPOSITION(id)(page, row)}. The same holds for continuous regions within a page, that can be granted for use in the program via \texttt{USEREGION(id)(page, start row TO end row)}.

The content of a memory cell is always scalar, while it can be dynamically typed. Statically-typed variables are also supported with a deterministic automatic storage-allocation policy. In addition, \textit{FLIP} supports arrays, records and notebook addresses (i.e. pointers), while programs have a typical block structure with control flow statements and subprograms. The syntactic differentiations of the \textit{FLIP} language from similar languages mainly concern direct memory access, as shown at the bottom of Figure 8.

4.2. Syntax matching patterns

Now we elaborate on the coding of the syntax matching patterns, which, on the one hand, manage to emulate the syntax of the source language by code snippets of the target language, while, on the other hand, encapsulate: (a) the extraction of lexical information (such as identifiers and source lines); (b) placeholders to hook execution event notifications; and (c) the necessary semantic mapping calls. As will be shown, such syntactic patterns are made possible with a mixture of macros, small utility classes, template functions, partially defined classes, overloading semantics and temporary instances. We first introduce the syntactic patterns for the \textit{JET} language followed by those of the \textit{FLIP} language.

4.2.1. \textit{JET} syntactic patterns

Program instructions without semicolons. In the \textit{JET} language, we had to accomplish an assembly-like syntax, with no parentheses and semicolons. Every instruction is implemented as a normal function, with an internal name. The external instruction names are actually macros defined to behave as follows (see Figure 9, instruction \texttt{INPUT}): (a) they syntactically close the previous instruction,
putting a parenthesis and a trailing semicolon; (b) they open a call for the requested instruction, putting as a standard first argument the source line. To ensure this algorithm works for all programs, it has to work for \( N = 1 \), i.e. the first instruction, and \( N = K \), i.e. the last instruction. This is accomplished via the \texttt{PROGRAM\_BEGIN} and \texttt{PROGRAM\_END} macros, which ensure graceful syntactic termination of the first and last commands, respectively (see Figure 9, upper part, last two lines).

The example at the bottom of Figure 9 shows a very simple source program (bottom left) illustrating the way the syntactic patterns manage to transform assembly-style instructions to typical C++ function calls (bottom right). Notably, this pattern will generally apply for all source language cases if function calls need to become more intuitive with a less-cluttered syntax, stripping away parenthesized argument lists and trailing semicolons. Branch instructions are a little more complicated, mainly because, being targeted towards the easiest implementation, we wish to directly exploit the availability of statement labels and the \texttt{goto} command of the C++ language. The problem is that label definitions in the C++ language are not expressions, but a form of \texttt{nop} statement, and so cannot be matched with an expression pattern (e.g. via a template parameter in a template expression pattern). In this case, the only solution is to apply a syntactic adaptation, so as to ensure issued labels comply with the previous closing/opening instruction syntax pattern. This is achieved easily via the \texttt{LABEL(id)}, as shown in Figure 10 (upper part, last source line).

The next step to complete the syntactic pattern of branch instructions concerns their syntactic transformation to encapsulate calls to the corresponding C++ \texttt{goto} statements. Again, since \texttt{goto} is
a statement while label identifiers are typeless, they are not allowed to mix with normal expressions. This implies that the part of the transformed branch instruction which issues a `goto` command should be a separate statement in between syntactically closed instructions.

This is accommodated by turning the last argument of the branch statement into a macro named `TO(label)` (see Figure 10), gracefully closing the branch-condition test (function call) and encompassing a `goto` statement. Hence, branch instructions, with their target instruction label, are transformed to `if (cond ()) goto label` target code snippets. Following this pattern, we may also note that the unconditional `JUMP` instruction is implemented in exactly the same way, using a condition function constantly returning `true`. Finally, as shown in Figure 10, two types of execution notifications are posted for branch statement, one before the branch test is performed, and one depending on whether the branch target is taken or not.

Declarations of instruction syntax. The rest of the JET instructions adopt a common syntactic pattern, as shown at the top of Figure 11 (e.g. `BREQ, BRGE,` etc.). Apart from these, there are also pseudo instructions, i.e. declarations with an instruction syntactic style, for defining variables and constants, as shown at the bottom of Figure 11. In this case, the choice of the syntax directly depends on the need for runtime tracing. In particular, if name information is required during tracing, then it is mandatory to construct a syntactic pattern that is capable of transparently extracting an identifier string upon compilation, something that is accomplished only if the identifier is provided as a macro parameter (using #), as shown in our pattern of Figure 11. Otherwise, an alternative, very simple syntactic pattern is possible, as shown in Figure 11; again, to avoid explicit semicolons, we use the same trick as before, requiring an explicit `DEFINITIONS` keyword (see the top part of Figure 12) and minor modifications to `PROGRAM_BEGIN` to produce a preceding semicolon.

Although the implementation of the latter syntactic pattern is very simple, it has to be backed up with the necessary semantic support. In particular, let us consider the possibility of defining `ARRAY name(size)` in the traditional way as `name[size]`. To make this syntax transformation plausible, it is mandatory that `size` is a C++ compile-time integer constant. However, since in the JET language programmers may define symbolic names for constant integer memory cells, it seems unnatural to forbid the use of such constants as array sizes. However, such constants are semantically mapped to constant `IntObject` instances (see the example of Figure 12), meaning they are inappropriate for C++ array sizes. The latter imposes the emulation of augmented single-dimension arrays as classes...
accepting different types of sizes as constructor arguments, thus transforming the original $P[N]$ declaration syntax to $P(N)$.

*Name extraction from non-parenthesized declarations.* In case name information needs to be extracted from declarations, like the pseudo instructions of the *JET* language, but without parenthesized definitions, single-pass preprocessing does not suffice. We provide an implementation pattern that requires two preprocessing passes, the first of which is essentially needed to transform the non-parenthesized form to a parenthesized one, calling typical name extraction macros, such as those provided in the syntactic patterns of Figure 11. The output from the first phase is compliant to the parenthesized forms of Figure 11, being also the input to the second preprocessing phase. In this sense, two-phase preprocessing with non-parenthesized forms is equivalent to one-phase preprocessing with parenthesized forms.

The resulting syntactic pattern (see Figure 13) is unavoidably more complicated in comparison to the previous ones, since it has to perform ‘code generation’ from the first pass, and certify that the resulting code will perform syntactic pattern matching in a way identical to our previous examples. Care should be taken since in the C-preprocessor there is no way of having macros which produce text spanning across multiple lines. As shown at the example of Figure 13, the first pass transforms the program to one in which parentheses are put around arguments, with calls to the normal macros of the syntactic patterns inserted as needed. The second pass results in the pure C++ program. In this sense, the first set of macros can be considered as meta-macros. A more complicated and sophisticated method is to have macros with recursive file inclusion, like in the Boost Spirit Meta Programming Library (MPL).

*Handling variable number of arguments.* In the *JET* language, all instructions pertain to an invariant and small number of arguments, from a very limited set of data types. In such cases, syntactic emulation with the ‘terminating $(N − 1)$th command, opening $N$th command’ style becomes straightforward. However, let us assume that the syntax of the language allows a variable number of actual arguments, where the order of an actual argument does not statically commit its type (i.e. we allow flexible signatures). In this case, we have to perform argument matching, collection and propagation to the internal semantic function, by appropriately overloading the comma operator. The technique, outlined in Figure 14, is based on the implementation of semantic functions to accept an *Arguments* reference, the latter being a list of *Argument* values. The *Argument* class is a unified type for all sorts of argument

---

**Figure 12. Alternative syntactic patterns for definitions.**

<table>
<thead>
<tr>
<th>DEFINITIONS</th>
<th>int _dummy</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM_BEGIN</td>
<td>int main (int argc, char **argv) { (0</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>; const IntObject</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>; IntObject name</td>
</tr>
<tr>
<td>ARRAY</td>
<td>; Array</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEFINITIONS</th>
<th>int _dummy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT</td>
<td>N = 10</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>x</td>
</tr>
<tr>
<td>ARRAY</td>
<td>arr(N)</td>
</tr>
</tbody>
</table>
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

MetaMacros.b

```c
#define DEFINITIONS #include "NormalMacros.h"
#define CONSTANT ) _CONSTANT(
#define PROGRAM_BEGIN ) _PROGRAM_BEGIN

NormalMacros.b

```c
#define _CONSTANT(name, value) ); IntObject name(value, #name
#define _PROGRAM_BEGIN ); int main (int argc, char **argv) { (0
#define DEFINITIONS int _dummy

DEFINITIONS((0

CONSTANT N , 10
PROGRAM_BEGIN

After 1st pre-processing pass
```c
#include "NormalMacros.h"
) _CONSTANT (N, 10 ) _PROGRAM_BEGIN

After 2nd pre-processing pass
```c
int _dummy ((0 )); IntObject N(10, "N"
); int main (int argc, char **argv) { (0

Figure 13. Two-pass preprocessing to extract name information without parenthesized declaration forms.

```c
struct Argument {
    Argument (int i);
    Argument (const char*);
    Argument (bool b);
    Argument (const Argument&);
};
```

```c
struct Arguments {
    std::list<Argument> args;
    template <class T>
    Arguments& operator,(const T& arg) {
        args.push_back(Argument(arg));
        return *this;
    }
};
```

For each instruction or function F we define the following pattern:
```c
#define F )); _F(_LINE_, (Arguments()));
```
The semantic implementation of F has the form:
```c
void _F (unsigned line, Argument& args);
```
Example:
```c
OUTPUT 12, 'hello', true, 3.14
); _OUTPUT(1,(Arguments()),12, 'hello', true, 3.14
```

Figure 14. Implementing the syntactic pattern for instructions allowing a variable number of arguments.

types acceptable to semantic functions, while Arguments is a class overloading the comma operator; thus, its instances are capable of consuming and collecting all expression values in a comma separated list supplied to their right side.

Following the technique shown in Figure 14, a call f(a1, ..., an) has to be transformed to the form _f((Arguments()), a1, ..., an)), where _f is equivalent to f but with a single Argument& argument; the inner pair of parentheses is mandatory, otherwise the compiler will complain that while _f is
Figure 15. The syntactic pattern for declarations with a parenthesized initialization expression list (e.g. USEPOSITION and USEREGION).

4.2.2. FLIP syntactic patterns

Name and line extraction for declarations with parenthesized initializers. In the syntax for the declaration of symbolic names for memory cells and regions we need: (a) to extract name as well as declaration line information, since those are needed by the semantic layers to associate any variable with its external name and source line of declaration (for execution tracing reasons); and (b) to accommodate the previous point for initializations that are syntactically parenthesized expression lists. The two related declaration categories concern memory positions and regions, through the USEPOSITION and USEREGION definitions, respectively (see Figure 15). As shown in Figure 15, the technique relies on a syntactic pattern in which the symbolic names are defined via pseudo commands (macros) where the variable name is always surrounded by parentheses. Consequently, by macro substitution, line and name are extractable in a straightforward manner via _LINE_ and # preprocessor facilities.

To make declarations syntactically cleaner for programmers, we chose to separate the rest of the declaration specifications, which are actually initialization expressions, as a trailing

---

parenthesized group. Apparently, if a declaration style similar to the JET pseudo instructions previously discussed is chosen, the previous respective implementation pattern would suffice. In this particular case, since the syntax produces two groups of parentheses next to each other, the second being the initialization list, it is not possible to consume those with only a single declaration.

The implementation technique of Figure 15 adopts the generation of a name and line storing implicit instance, i.e. LineNameRecorder, followed by the generation of a partial (variable or region) declaration, i.e. Object\& x = VerifyUnusedPosition for variables or Region in place of Object for regions, syntactically completed by the trailing parentheses group to constitute an instance declaration. As shown in the definition of the LineNameRecorder utility class, line and name information is stored in static members. Those are read and reset by variable or region constructors, which, owing to our coding pattern, are always certified to be called at runtime after their corresponding line and name recording call.

Line extraction in calling overloaded functions. This situation concerns scenarios in which we have to extract the source code line when calling overloaded functions, the latter varying with respect to argument types as well as the number of arguments. In our case such a scenario concerns the CONTENTOF function, accepting either an address variable, or a pair of expressions denoting the memory page and row. Semantically, ADDRESS types are mapped to special-purpose classes emulating pointers, while, syntactically, the declaration of ADDRESS requires trivial lexical mapping of the type name (i.e. as seen in Figure 16, from ADDRESS to AddressObject of the semantic implementation).

The implementation of CONTENTOF as a macro with a variable number of arguments is not portable across all platforms. In this context, the following solution generally resolves this particular scenario.

- For overloaded function _F, for example _CONTENTOF of Figure 16, a corresponding functor class FF is implemented, for example ContentOf of Figure 16. In FF, for each version of _F we implement a corresponding version of the function call operator with the same signature as a delegate call to _F. In addition, FF is derived from LineTrapper (see Figure 16), so it requires a single line construction argument that is also supplied to the delegate calls of the _F function versions.
- A macro F is defined, corresponding to the external name of _F, for example CONTENTOF of Figure 16, as follows: #define F FF(_LINE__). Now, F is the required overloaded version of _F, supporting effectively call line extraction.

The previous programming pattern will work generally once there is a need to extract the values of preprocessor variables, such as source line, source file or encompassing function, for built-in overloaded functions that support multiple alternative signatures with possibly a varying number of arguments. In this case, distinct versions of functor classes, like the ContentOf of Figure 16, will have to be defined accordingly. Finally, the generation of execution events is better encapsulated within the _CONTENTOF implementation; that is why they do not become part of the previous syntactic pattern.

Trapping condition pre/post-evaluation and value for control flow statements. This scenario concerns branch and loop statements typically involving the evaluation of a condition expression. In the FLIP language, we support if, while and for statements. The syntactic pattern itself is trivial, since the only differentiations from the C++ counterparts concern minimal lexical variations (e.g. IF instead of if, WHILE instead of while, and no parentheses around conditions). However, in FLIP we require: (a) to trap at runtime the point exactly before and after the condition is evaluated; and (b) to trap the value of the condition to post an execution event depending on whether the ‘if’ statement is entered or skipped.
```cpp
#define CONTENTOF ContentOf (_LINE_)
Object& _CONTENTOF (const Object& x, unsigned line);
Object& _CONTENTOF (unsigned page, unsigned row, unsigned line);
class LineTrapper {
    protected:  unsigned line;
    public:
        LineTrapper(unsigned _line) :line(_line){}
        virtual ~LineTrapper(){}
    }
    class ContentOf : public LineTrapper {
        public:
            ContentOf (unsigned _line) : LineTrapper(_line){}
            Object& operator() (const Object& x) const
                { return(_CONTENTOF(x, line)); }
            Object& operator() (unsigned page, unsigned row) const
                { return (_CONTENTOF(page, row, line)); }
        }
ADDRESS p(0,0);          AddressObject p(0,0);
CONTENTOF(p) = CONTENTOF(0,1);      ContentOf(2)(p) = ContentOf(2)(0,1);
```

Figure 16. The syntactic pattern to accomplish line extraction in the call of overloaded functions (e.g. `CONTENTOF`).

The implementation method, applied for the ‘if else’ statement, is provided under Figure 17. Following Figure 17, the utility of the `IfTrapper` class relies on two facts: (a) in an expression list, when the comma operator is not overloaded, expressions are evaluated left to right; and (b) temporary instances as part of an expression are destroyed with the reverse order of their instantiation, exactly after the expression evaluation completes. Consequently, the `IfTrapper(I)` temporary instance not overloading the comma operator is constructed before `x>y` is evaluated, and is destroyed after the evaluation of the whole augmented condition concludes.

Based on this remark, the constructor and destructor of the `IfTrapper` class constitute the entry points to post-notifications regarding pre/post-evaluation of the condition expression, respectively. The same technique is applied for the `While` statement (For statements require special treatment).

Our pattern is completed through the method to trap the condition expression value. We use a temporary instance of class `CondValueTrapper`, overloading the comma operator to collect all comma separated expressions supplied or the right side (typically, one is supplied, such as `x>y` of our example, but we want to support an expression list as in C++). This instance stores the boolean value of the last condition met in `val`, while supporting auto conversion to boolean via `operator bool()` the latter is automatically called by the compiler to convert from `CondValueTrapper` to `bool` as the last step of condition evaluation; hence, in `operator bool()` we post a notification regarding the expression value.

The comma operator implementation provided within Figure 17 handles both `bool` values (C++) and `BOOLEAN` values (`FLIP`).

Extracting lexical information for context-dependent statements. Context-dependent statements are those whose semantics restrict their presence within their original definition context, meaning they cannot be wrapped within delegate function calls, while the capability to encapsulate them
class IfTrapper {
    unsigned line;
public:
    IfTrapper (unsigned _line) : line (_line) { NOTIFY_IFENTER(line);} 
    ~IfTrapper() { NOTIFY_IFEXIT(line);} 
};
class CondValueTrapper {
    std::string stmt;
    bool val;
public:
    operator bool(){
        std::string s;
        if (val) s = "TRUE condition, entering ";
        else s = "FALSE condition, skipping ";
        s.append(stmt);
        NOTIFY_CONDITIONVALUE(s.c_str(), val);
        return val;
    }
    CondValueTrapper& operator,(bool b) { val = b; return *this; }
    CondValueTrapper& operator,(BOOLEAN& val)
    { this->operator,(val == true); return *this; }
    CondValueTrapper(const char* stmt) : stmt(stmt) {} 
};
#define IF if (IfTrapper(LINE), CondValueTrapper("IF"),
#define THEN )
#define ELSE else
IF x>y THEN OUTPUT(x);  
ELSE Output("x", x);

Figure 17. Trapping pre/post-evaluation and value of the condition expression of IF ELSE statements.

within artificially introduced compound statements has limitations dependent on the type of statement. Examples of such statements in C++ are the return, break, continue and goto statements. While syntactic matching is very easy, since lexical mapping is only needed (i.e. renaming), the extraction of lexical information, such as source line, source file and encompassing function (for preprocessors supporting the latter) is not that obvious. In the FLIP language, the statement FINISH() within functions is semantically equivalent to return of C++. Apparently, to allow the FINISH() command to encompass notifications for function return, it has to be defined as a macro, otherwise the original semantics cannot be preserved. The solution to our problem is the syntactic pattern provided in Figure 18. Presumably, the defined coding pattern may look a little awkward at a first glance. For instance, one might consider that an apparent easier alternative would be a macro like {NOTIFY_FINISH(...) ; return ;}. However, in order to make our example at the bottom left of Figure 18 syntactically correct, the trailing semicolon after FINISH() should be necessarily removed: the block and a semicolon constitute two statements, meaning the ELSE of the next line is turned to syntax error. However, because of this particular case, the simpler alternative syntactic pattern would turn the FINISH() command into an exception to the rule ‘all statements terminate with a semicolon’.
For the general case, we needed a syntactic pattern which was: (a) capable of encapsulating a context-dependent statement together with a notification call; and (b) able to form syntactically a single statement, closing gracefully with a trailing semicolon.

While the previous syntactic pattern worked gracefully as a ‘decorated’ substitute of the return statement, it cannot work in the same way for break and continue as well. The reason is inherent in the semantics of those two loop control-flow commands that always match with their inner enclosing loop. In this context, we had to seek for an alternative pattern, with the same qualities as before, but without syntactically introducing a wrapper loop. The solution is provided in Figure 19, introducing an artificial if else statement. Everything is put inside the if block, which is always executed, while the else is empty, syntactically completing with the trailing semicolon. As is also shown in Figure 19, the same syntactic pattern is employed for the implementation of the GOTO statement.

**Trapping the lifetime and storage location of aggregate data.** We start our discussion with records, since those require the prior definition of a user-defined data type. In this context, there are two key requirements owing to the semantics of the FLIP language that had to be effectively met: (a) the need for posting of notifications regarding the instantiation and destruction of record variables in the notebook memory; and (b) the ability to support the query of the starting page and row in the notebook memory, via the POSITIONOF built-in function, for record variables.

In this implementation case, shown in Figure 20, although the syntactic emulation is trivial once again, the accommodation of the semantic aspects, without ‘polluting’ the original syntax, is a little more demanding. The wraparound is needed so as to ensure that record instances are aware of their starting location in the notebook memory, while also certifying that record types implement the basic GetRow() and GetPage() interface, similarly to the rest of the data types, as those functions are required for the POSITIONOF built-in function. For this purpose, the RecordSuper class is defined, which
Figure 20. Syntactic pattern for records trapping instantiation/destruction and recording the starting memory cell.

encompasses notifications for instantiation and destruction, while recording the starting memory page and row as the first free memory position. Following the C++ semantics, the construction of the superclass instance is certified to take place before the derived class members, so that the starting memory cell will be correctly recorded. Finally, since the constructor of the base instance is called prior to the one for the derived instance, while the destructors are called in the reverse order, the notifications encapsulated within `RecordSuper` of Figure 20 are verified to behave correctly. The same technique is directly applicable to arrays. Their syntax is different from the C++ style, since arrays are internally defined as template types, so as to facilitate array indexing validation.

The basic array template class is derived from an `ArraySuper` template class, offering functionality similar to `RecordSuper`, i.e. book-keeping of first memory cell and posting of notifications. The syntactic pattern is provided in Figure 21. The variations when compared to records concern mainly the different style for notifications: ‘allocation’ before the array elements are constructed; ‘declaration’ after they are constructed; and eventually ‘destruction’ before the array elements are destroyed.

### 4.3. More advanced scenarios

We now study more demanding scenarios, starting with an enhanced `case` statement for the `FLIP` language, which is significantly different from that supplied in the target language (i.e. C++ `switch`), and continuing with an entirely different language exercise, having a functional programming style, named `LEAF` (Language Emulation Applied for a Functional language). Then, we try to generalize commonly appearing grammar patterns, showing the transition from the grammar rules to the respective target code patterns, preserving the syntax and operator precedence regulations.

#### 4.3.1. Enhancing/extending statement semantics

For the `FLIP` language this concerned the ‘CASE’ statement, enhanced to have the following semantics: (a) the basic case expression should be of a scalar type (including strings); (b) a list of multiple
```c++
#define ARRAY(T,N) Array<T,N>

template <const unsigned N> class ArraySuper {
    ... as in RecordSuper...
    ArraySuper (void) { ... NOTIFY ARRAYALLOC(N) ; }
};
template <class T, const unsigned N> class Array : public ArraySuper<N> {
    T array[N];

    public:
        Array (void)
            { NOTIFY ARRAYDECL(GetPage(), GetRow(), N); }
        ~Array (void)
            { NOTIFY ARRAYDESTR(GetPage(), GetRow(), N); }
        T & operator[] (int i)
            { NOTIFY ARRAYINDEX(GetPage(), GetRow(), N, i);
              VerifyIndex(i);
              return array[i];
            }
    ...;
};
```

Figure 21. The syntactic pattern for arrays, with various categories of notifications and verifiable array indexing.

Matching expressions may be provided per case statement, which may not be constant, evaluated only during matching in a left to right fashion; (c) the case statement may optionally incorporate code to be executed all the time, even if no case expression is matched; and (d) although there are no control fall-through semantics as in C/C++, a break statement is still applicable to instantly exit the whole case statement, dismissing also the optional code provided at the end. The structure of a FLIP case statement is provided within Figure 22 at the top, while the syntactic pattern together with the necessary utility classes are shown in the middle. Finally, a generic example showing the actual code transformation from FLIP to C++ is provided at the bottom of Figure 22.

Following the implementation pattern of Figure 22, the case statement is transformed to a 'do while' statement, playing a two-fold role: (a) the whole case statement becomes syntactically a single statement gracefully terminating with a trailing semicolon; and (b) the break statement is directly applicable, causing immediate exit from the 'do while', i.e. essentially from the case statement. As shown from the implementation pattern in the middle of Figure 22, the value of the case expression is recorded in a temporary variable named `caseVal` of type `Object`, the latter (in the FLIP implementation) capable of holding any scalar value. The distinct case value-matching specifiers, i.e. ON exprlist ACTION, are transformed to else if statements. The most demanding aspect of this coding pattern is the way the ON list is transformed to an if condition capable of accommodating the evaluation and testing of the expression list with the recorded case value. This is accomplished by the if expression in three steps, the first two carried out in any order: (i) fetch the case value; (ii) evaluate the expressions by collecting the values in a list; and (iii) testing equality of the case value with any of the values in this list. This is mapped to the following if statement in C++: if (fetch()==eval()), assuming eval expresses the evaluation of a list of expressions. Following Figure 22, fetch() is mapped to CaseFetch(caseVal), the latter creating an instance recording internally the case value, while also supporting equality.
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

```cpp
CASEBEGIN(expr)
    ON e1, e2    ACTION stmt1;
    ON e3, e4, e5 ACTION stmt2;
    ELSE        stmt3;
    stmts;      // Always executed, unless a break was met
CASEEND;

class CaseEvalList {
    std::list<Object> 1;
public:
    CaseEvalList& operator,(const Object& obj)
    {
        1.push_back(obj); return *this; }
    bool operator==(Object& obj) const
    {
        return std::find(1.begin(), 1.end(), obj) != 1.end();
    }
};

struct CaseFetch {
    Object& val;
    bool operator<<(const CaseEvalList & 1) { return 1 == val; }
    CaseFetch (Object& v) : val(v) {};
}

#define CASEBEGIN(expr) \  
do { NOTIFY_CASEBEGIN(LINE_); Object _caseVal = expr; if (false) ;
#define ON                if (CaseFetch(_caseVal) == (CaseEvalList (),
#define CASEEND          NOTIFY_CASEEND(LINE_); } while(false)
#define ACTION  )

do { NOTIFY_CASEBEGIN(); Object _caseVal(expr); if (false) ;
else if (CaseFetch(_caseVal) == (CaseEvalList(), e1, e2 )) stmt1;
else if (CaseFetch(_caseVal) == (CaseEvalList(), e3, e4, e5 )) stmt2;
else stmt3;
stmts;
NOTIFY_CASEEND(6); } while(false);
```

Figure 22. The CASE statement syntactic pattern with a generic example of how it maps to C++.

comparison with an expression list; `eval()` is mapped to `(CaseEvalList())`, the latter creating an instance which by overloading the `;` operator populates the values of a trailing expression list.

4.3.2. Functional programming style

In some cases, the source language may display properties that are not mapped to those of the target language. For instance, if a functional programming style needs to be supported, such as the example displayed in Figure 23, mere syntactic mapping does not suffice, since the source programming elements do not have a counterpart in the target language. In particular, we have implemented a small-scale functional-style language named LEAF (details of availability are given in the acknowledgements), using the reported technique, with the following properties:

- function definitions as expressions (with lambda unnamed functions); no block structure, no mutable variables, no imperative state manipulation;
Figure 23. A ‘max’ function implemented in LEAF, as nested expressions with unnamed (lambda) functions and recursive calls, compiled as C++ code, but displaying a syntactic and semantic paradigm shift from C++.

- lazy execution and only call-by-value semantics;
- dynamically typed, with functions as first-class values and closures;
- state transformation by return values, state propagation as immutable actual arguments;

Apparently, reflecting the paradigm of functional languages [9], the semantics of LEAF signify a radical departure from the imperative style (of C++), the latter emphasizing block-structured statements, mutable variables and imperative state manipulation. An analogy to functional programming for Java is discussed in [10]; however, this implementation still involves imperative elements within user programs, while there is no syntax abstraction via emulation as is the case with LEAF (essentially due to the lack of operator overloading and preprocessor support in the Java language). In such cases, the syntactic patterns tend to be relatively simple, while the implementation challenge is shifted towards the semantic aspects which significantly differ in comparison to the target language. As shown in Figure 24 (excerpts), the syntax patterns account for a small number of relatively simple macros, while the semantic domain required implementation of the various functional programming constructs, e.g. functions, expressions, state and values, directly into the target language. The latter is indicated by the list of the special-purpose classes for the semantic implementation, essentially corresponding to the basic functional language constructs, as shown at the bottom of Figure 24 (a subset is provided, the complete implementation is publicly available—see the acknowledgements).

From the syntactic point of view, the parenthesized syntax is easily handled since it can be matched in a straightforward manner via function calls (including constructor calls as well). Notably, the syntax may become severely simplified once we employ techniques to concatenate...
arguments via overloaded operators and macros to hide the operator presence, as we discuss in the next section. Following Figure 24, all lists with a variable number of items, such as formal arguments, statements and expressions, are handled either by overloading the \texttt{\textless} operator (used for formal and actual arguments, i.e. \texttt{FORMAL} and \texttt{ARG} macros) or the comma operator (used for pseudo statements, i.e. \texttt{compound} class).

The keywords for defining functions, arguments and different types of expressions are defined as macros that encompass the declaration of temporary instances. This is possible owing to the safe copy semantics of all classes that implement \texttt{LEAF} language constructs, practically enabling such broad use of temporary instances. This allows the implementation of simple and clean syntactic patterns, since we may freely incorporate expressions to accommodate the desirable syntax matching, although these
may lead to multiple sequential copy actions for a single element (e.g. an add expression instance) before it becomes actually available to the corresponding handling function.

4.4. Prescribing a methodology

4.4.1. Generalization possibilities for the syntactic patterns

We briefly investigate the potential of generalizing various syntactic patterns, so that their deployment becomes more prescriptive. We first focus on expressions of the form shown in the top left part of Figure 25, generally representing whitespace separated expression lists (i.e. no commas); the number of grammar symbols on the right-hand side of the production may vary and in our example we assume a group of three grammar symbols. As shown at the top right of Figure 25, this grammar produces sentences of the form $a\beta^a \gamma^b$. The language elements corresponding to the grammar rules may be typed or not, but we always need to associate every grammar symbol with a special-purpose utility class, since the latter is necessary for implementing the syntactic pattern. The $B$ and $C$ grammar symbols represent repeating items, while the role of the grammar symbol $A$ is to collect upon evaluation all instances of $B$ and $C$ terminals. For instance, for $S$ representing function calls, the $A$ symbol is a wrapper class for the called function.

In Figure 25, grammar symbols are mapped to a corresponding class, where the constructor arguments indicate that the terminals $\alpha$, $\beta$ and $\gamma$ may accept arguments as well. In addition, such terminals are grouped together via an appropriate overloaded operator. For example, $B(args)$ is defined as $+\beta(args)$, meaning $\beta$ instances are collected together via $+$. The choice of operators should guarantee that the relative precedence fits the priorities for matching grammar rules. In particular, assuming a priority $B > C > A$, we have to ensure $B$ items are grouped before $C$ items, and those before $A$ items. To accommodate this, as shown in Figure 25, we chose operators of corresponding precedence. The type signatures for overloaded operators ensure that the resulting expressions constitute syntactically and semantically valid C++ expressions that are evaluated according to the grammar priority semantics (see the example at the bottom of Figure 25).

Addressing key issues in order of operator evaluation. An important remark, not only regarding the previous pattern but more generally related to syntax emulation using overloading of C++ operators, concerns the order of evaluation of expressions pertaining to the same grammar symbol. For instance, let us take the $A$ grammar symbol, whose expressions are collected by overloading the $\ll$ shift-left operator. Clearly, given any $X$, $Y$ and $Z$ expressions of $A$, the $X \ll Y \ll Z$ is an $A$ expression. Owing to the fact that $\ll$ is left-associative, it is certified that the result is always $(value(X) \ll value(Y)) \ll value(Z))$. However, beyond that, there is no guarantee of the order of evaluation of the constituent expressions, meaning we cannot assume that $eval(X)$ before $eval(Y)$ before $eval(Z)$. Based on the C++ semantics, since the overloading operators are essentially functions, the evaluation of the $X \ll Y \ll Z$ is equivalent to $Z.\text{operator} \ll (X.\text{operator} \ll (Y))$, being normal function calls for which there is no imposed evaluation order for $X$, $Y$ and $Z$ expressions. The previous remark should be taken into consideration when designing the syntactic patterns, to avoid evaluation assumptions relying on any particular ordering scheme. In particular, when implementing the syntactic patterns of a language, the first grammar symbol on the left-hand side of such expressions, for example the $A$ of the example in Figure 25, may have to encapsulate code to be executed before expressions for $B$ or $C$ are evaluated. For instance, $\text{LAMBDA}$ for the example of Figure 23 (i.e. $A$) semantically encompasses code.
A grammar for three groups of typed items separated with white space.

<table>
<thead>
<tr>
<th>S</th>
<th>A B C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>α</td>
</tr>
<tr>
<td>B</td>
<td>β</td>
</tr>
<tr>
<td>C</td>
<td>γ</td>
</tr>
</tbody>
</table>

Examples of sentences produced by the grammar, all of the form $αβγ$.

$S \rightarrow ABC$

$A \rightarrow α$ $αγγγγγγ$

$B \rightarrow β | β | ε$ $αβββε$

$C \rightarrow γ | γ | ε$ $αγγγγγγ$

Priority in grammar matching is given with the following order: $B > C > A$. So, we use operators of corresponding precedence to collect expressions for each grammar rule, as follows:

- $*$ for $B$
- $+$ for $C$
- $<$ for $A$

Internally, every $A$, $B$ or $C$ terminal item is an instance of a corresponding class as follows:

```cpp
class α;
#define A(args) α(args) << βγ
```

```cpp
class β;
#define B(args) *β(args)
```

```cpp
class γ;
#define C(args) +α(args)
```

```cpp
class βγset:
βγset is a special purpose collector class populating β and γ instances
```

Operator type signatures:

- $βγset*β$ $\rightarrow βγset$
- $βγset+γ$ $\rightarrow βγset$
- $β β$ $→ β$
- $γ + γ$ $→ γ$
- $α << βγset$ $→ α$

Example showing reduction by evaluation:

$A() B() B() B() C() C() C() \rightarrow α< < β γ set()∗β ()∗β ()∗β ()∗β ()+γ ()+γ ()+γ ()α << βγset ()α$

Figure 25. Grammar and implementation of syntactic patterns regarding space-separated expressions lists.

that must be evaluated before the code snippets for the $FORMAL(n)$ specifier (i.e. $B$) and the trailing function body $=BEGIN...END$ (i.e. $C$). The latter implies that we need to force the evaluation of $A$ independently of the semantics of operators employed to concatenate with $B$ and $C$. The latter is achieved by transforming $ABC$ to $(E, A B C)$, where $E$ is an expression evaluating the code for $A$, while returning an unused built-in scalar value. This technique works owing to the semantics of the comma operator, which is not overloaded for $E$ of a scalar type: $E$ is certified to evaluate before $A$, $B$ or $C$.

At the top of Figure 23, the $E$ expression for $LAMBDA$ is $func::newfunc(func::lamdaid().c_str())$, which actually creates a new lambda function entry according to the $LEAF$ language semantics.

The concatenation of source language expressions by overloading the C++ language operators is a powerful method that may lead to astonishingly simple syntax that is very natural to the language user. For instance, in Blitz++ [3], matrix initialization for a $3 \times 3$ array $A$ is possible in a form such as $A = 1, 2, 3, newline 4, 5, 6, new line 7, 8, 9; \ (taking\ advantage\ of\ comma\ overloading)$.

### 4.4.2. Limitations

In this section we elaborate on the key limitations of the approach, mainly inherent in the complete absence of a strict syntax analyzer, the latter as required according to the traditional compiler construction theory. We recall that the proposed method is not put forward as a substitute
to the common language development practice, but as a conditionally advantageous technique. In the following discussion, while presenting the most evident limitations, we also show how these are considered as less critical and avoidable in our proposed method.

1. Not every unambiguous context-free grammar can be implemented as it is. Manual grammar adaptation is likely to be required so as to transform the grammar rules to a form eligible for syntactic emulation as C++ programming patterns. However, since the usability of a programming language may be accomplished with alternative syntactic and semantic forms [11], the grammar transformation as such is not considered as a serious restrictive barrier.

2. Strict syntax checking of the source language is not possible, since target language code snippets embedded in source language units may pass as syntactically correct, as everything is essentially mapped to target language code. Effectively, a legal C++ declaration, statement or expression, when put at an appropriate place inside a code snippet of the new source language, will almost always pass as legal source code. In our case, this particular characteristic is surprisingly desirable for the following reasons. First, we emphasize the support of hybrid code, meaning the knowledgeable programmer should be able to mix language paradigms together into a single code unit, while the novice programmer should be kept aware of the keywords that, although syntactically legal, are semantically outside the language. Second, when it comes to delivering languages for teaching purposes, on the one hand we do not focus on the strict syntactic forms of the programming constructs but on their generic semantics independently of lexical or grammatical form, while on the other hand we demonstrate to students the incremental transition of a genuine FLIP program to its equivalent pure C counterpart via intermediate hybrid programs. To accommodate the latter with traditional compiler development methods one has to pay a significant development overhead.

3. Owing to the presence of syntactic patterns, compile errors are those detected by the C++ compiler, meaning they always narrow down to C++ compile errors. As a result, the compile errors eventually reported may be semantically or syntactically distant from the original source errors, a fact which renders their recognition and assimilation by novice programmers or students less than trivial. To accommodate this issue we have: (i) reproduced common errors made by students, such as incomplete statements, unbalanced parentheses, missing declarations, misspelling of identifiers, unbalanced blocks, etc., and studied the resulting compiler messages; and (ii) documented such messages to students, spotting and explaining their actual meaning and showing how to apply the correct wraparound to resolve the error. From our experience in an academic teaching context, this approach seemed to constitute an effective and well received recipe to address the error recognition problem. It should be noted that this issue of decreased compile-message readability is also an issue of pure C++ code involving templates, since template-specific error messages may become so obscure, verbose and miss-targeted that specialized tools to filter such messages have been developed, for example STLFilt [12]. Naturally, the option of crafting a trivial syntax checker, as an add-on to the language, is always an eligible and apparent solution. The reason it was not adopted is twofold: (a) the described approached proved to be satisfactory in a ‘protected’ teaching environment; and (b) we intended to put students in front of a real development environment from the beginning, so that they become familiar very early on with error messages of the tools they are going to deploy latter in real practice.
5. SEMANTICS

The implementation of the language semantics is an issue that may significantly vary among different language categories. Typically, imperative languages, such as FLIP or JET, since they provide elements for state manipulation, inherently require the delivery of a memory model, the support of program variables (mutable storage) and an explicit runtime environment. On the other side, functional languages such as LEAF require a semantic layer that supports lazy computation and functional computation elements. Eventually, domain-specific languages may reflect an underlying semantic layer beyond the traditional programming language semantics, linking essentially to a domain-oriented software library, available externally to programmers through a language abstraction. In this context, we briefly discuss various details for quickly implementing and linking the syntactic layer with the semantics aspects of the FLIP and JET languages. The emphasis is not put on the intrinsic semantic details of these languages, but mostly on the contact side of the semantic layer with the syntactic patterns and on the flexibility for rapid development and extensibility of the language semantics.

5.1. Memory, variables and runtime environment

5.1.1. Memory system

Although the memory models of the FLIP and JET languages were fundamentally different, since FLIP offers a metaphoric high-level model while JET memory is very close to the machine, they were both implemented as a respective library with typical read/write access and allocation services. In the FLIP language, memory cells pertain to dynamically typed objects, while in the JET language they are unsigned 32-bit integer values. In order to support variables, it is necessary to handle the runtime association among symbolic names and automatically-allocated memory cells. The following technique has been applied in the FLIP language, which can also be adopted for any language offering statically or dynamically typed variables. Every distinct type \( T \) is mapped to a respective type class, the latter playing the role of a proxy to memory cells, as follows (see Figure 26): the proxy class uses the memory system to allocate, upon construction, memory cells as needed, depending on the storage requirements of the type, while subsequent read/write accesses are delegated to the memory system using the internally stored memory-cell references. Static typing is directly accomplished at the level of the class \( T \), by filtering out all \( T \) incompatible values while raising type conflict errors. Dynamic typing support (for scalar values) is straightforward once any value is allowed to be stored within the referred memory cells. In addition, this delegation scheme, separating variable types from real storage, allows pointers (either statically or dynamically typed) to be easily implemented, since they become similar proxy classes whose values are actually memory-cell references.

5.1.2. Runtime environment

The runtime environment of the FLIP language is essentially the runtime environment of the target language. In particular, the semantics for actual parameter passing, calling and returning from functions, the structure of activation records, lifetime of local variables, etc., are effectively those of C++. This is because of the fact that functions, variables, statements, expressions and blocks are all syntactically mapped to their counterparts in the C++ language. Consequently, there
Figure 26. Rapidly implementing types as proxies to memory cells, using the underlying memory system.

is no need to implement from scratch a runtime environment for the FLIP language, meaning FLIP essentially inherits/delegates its runtime environment from/to the target language. Typically, this flexibility is inherent in the proposed development method, since, by building the semantic layer on top of a high-level language, in comparison to operating system calls and a CPU instruction set, an execution environment infrastructure is taken for granted. The easy mapping, instead of implementation, of most FLIP elements to the C++ language is due to the fact that FLIP is an imperative procedural language where blocks, statements and expressions have direct counterparts to C++, with equivalent execution semantics. In contrast, the JET language as a virtual processor assembly language has a primitive execution environment, being essentially the software processor emulation system. In this sense, the JET language practically has no real runtime environment, since the latter denotes, for programming languages, the execution framework built on top of the target machine and the operating system. Regarding the implementation of variables for the JET language, the technique previously described, as illustrated within Figure 26, is directly adopted. Even though the memory system of the FLIP language is radically different in comparison to that of the JET language, the proposed approach of treating variables as storage proxies, rather than as direct synonyms to memory locations, allows the implementation of varying memory systems under a single, uniform and simple layer for program variables. Finally, the runtime environment of the LEAF system is a sort of oxymoron, as for functional languages there is no mutable storage. The runtime infrastructure of the LEAF language narrows down to the implementation of the language semantics, requiring at the implementation level the support for internal representation of functions and lazy evaluation. In particular, the largest part of the semantic implementation, excluding the extra code related to the syntactic patterns, is entirely orthogonal to the syntactic aspects of the language. Consequently, one may build an interpreter of the LEAF language in C++ by directly reusing the original semantic implementation. This implies that the proposed technique may be employed for rapid prototyping, followed by the final interpreter implementation, dropping only a minimal part of the code, i.e. the definitions of the syntactic patterns of Figure 24. Surprisingly, this very small piece of code essentially constitutes the prototype of the syntactic analyzer for the final interpreter.
5.2. Operators, expressions and statements

5.2.1. Operators

Language operators may be implemented in various ways, depending on the lexical, syntactic and semantic requirements within our source language. In case the operators are present in the target language, for example like the common arithmetic or relational operators available in C++, then there is always a possibility for straightforward deployment via overloading. In this case the operator of the source language is essentially inherited from the target language. Care should be taken as to whether the operator syntax and semantics are compatible among the source and the target languages. For example, an infix operator cannot be converted to a postfix or prefix form, while the precedence rules cannot be altered.

For operators not supported in the target language, a verbal notation should be adopted, i.e. a keyword style. In this case, the keyword may be a macro for another operator in the target language that is used to emulate the source language operator. For instance, if the source language does not aim to support the C++ exclusive or operator, lexically \( \land \), but supports an exponent operator, then the overloading of \( \land \) may be used for number types to implement the exponent, while a macro such as \( \text{EXP} \) may be defined as \( ^\land \) to allow expressions of the form \( x \text{EXP} y \). In this case, the operator of the source language is essentially emulated via a different operator of the target language.

Eventually, if none of the above holds, then the operator of the source language is implemented from scratch in the target language, syntactically deployed in a functional form \( \text{Op}(x, y) \), \( \text{Op} \) implying the call of an appropriate semantic function to apply the actual operator logic.

In all of the previous cases, the operator semantics are implemented from scratch for the types of the source language, applicable normally to constants and variables (the latter for imperative languages only). For the FLIP language, operators are directly mapped to semantic functions with immediate evaluation upon call, performing type checking according to the semantics of the language, while in the LEAF language operators are implemented as classes that support lazy evaluation. In the context of the JET language, operators are semantically associated to functions implementing the corresponding processor instructions.

5.2.2. Expressions

In the FLIP language, once the implementation of the operator semantics is provided, the management of expressions is essentially redirected to the target language, since all expressions of the source language are mapped to expressions of the target language. Hence, expression evaluation is carried out according to the evaluation semantics of operators and operator functions that constitute the semantic layer implemented in the target language. From the development point of view, the common practices applicable in compiler or interpreter construction for expression evaluation such as expressions trees and evaluation algorithms are not required, since expression evaluation is delegated to the basic mechanisms of the target language. Surprisingly, although this is an apparent significant saving in development time, it does not imply that there is a compromise on the repertoire of potential source language expressions. In contrast, as we have seen, the capabilities of implementing new types, storage models, type systems, expression operators and syntactic patterns on top of the C++ language are very broad. The expressions of the JET language are very primitive, narrowing down to literal or
primary expressions, not involving any operators. Hence, instead of expression evaluation there is only a straightforward expression mapping, such as mapping symbolic names and memory addressing expressions to their respective memory cells, or register names to register variables. Finally, since the semantics of expressions in the LEAF language are radically different than those of the target language, there is no way to accommodate their implementation through the mapping method. In this context, lazy expression evaluation is implemented through the construction of expression trees, which, for functional languages, is based on the fundamental distinction between \( x \ op \ y \), which evaluates to the creation of a primitive expression tree of the form \( x \rightarrow \ op \leftarrow \ y \), and \( \text{eval}(x \ op \ y) \), which effectively evaluates such a subtree as \( \op(\text{eval}(x), \text{eval}(y)) \). In this context, there is no straightforward redirection of the expression evaluation algorithm to the target language, as there is no trivial semantic mapping, but implementation of the expression evaluation scheme from scratch to separate expression instantiation from later on-demand evaluation.

5.2.3. Statements

The support of statements concerns only the FLIP language, since in the JET language there are only basic machine instructions, while in the LEAF language there are only expressions (compound is a pseudo statement, i.e. an expression list). While most of the statements in the FLIP language are syntactically and semantically transformed to their counterparts in the target language, there is no actual barrier in the proposed method mandating any strict syntactic or semantic conformance to the target language statements. For instance, domain-specific categories of basic statements may well be introduced, or specialized versions of control-flow statements may be easily implemented. For example, while there are FLIP control-flow statements mapping to the ‘if else’ and ‘while do’ of the C++ language, the FLIP case statement displays severe syntactic and semantic variations in comparison to the C++ switch analogy. From our experience, the key challenge is to support an intuitive and usable syntax by implementing the necessary syntactic patterns while internally linking to the statement semantics. Overall, the effort put in supporting statements met only in the FLIP language practically narrows down to their semantic implementation, since the syntactic emulation relies on reusable techniques as previously discussed. In a way similar to supporting expressions, the accommodation of the rest of the statements in the source language is automatic, once the syntactic mapping to the target language is accomplished. This convenience is expected in all cases where there is compatibility among the programming paradigms of the source and the target languages.

6. TRACING AND VISUALIZATION

The implementation of an execution tracing and visualization instrument, apart from being a prominent objective so as to better support programmers of a new domain-specific language, is generally known to be a very demanding and complex programming challenge. Source-level tracers and visualizers are typically deployed for defect detection and testing purposes, known as source-level debuggers, while they may allow experienced programmers to gain more intrinsic knowledge of the underlying language runtime system. We will briefly show that this task becomes easily manageable, when implementing languages with the proposed approach, following a clean architecture that allows alternative visualization or tracing policies to be modularly incorporated (such as performance statistics, memory access charting, function call frequency, runtime call graphs, etc.).
6.1. Architecture

Following Figure 27, the tracing infrastructure relies on a runtime event manager that provides an exhaustive list of all possible execution events. Calls to trigger the generation of such events are effectively hooked in the syntactic patterns and the semantic layer of the language. Effectively, in the absence of a runtime execution tracer, such event generation hooks are stripped off. On top of the event manager lays the execution visualizer super-class, providing virtual call-backs, corresponding to the handling of distinct execution events, in which derived visualizer classes are expected to specialize accordingly. For instance, the console tracer provides such call-backs with implementations displaying console messages, while the graphical visualizer supplies functions that manage to interactively deliver execution tracing information to programmers (see Figure 28, for the graphical visualizer of the FLIP language).

The memory system mirror illustrated in Figure 27 is a very simple public API providing read-only access and update notifications regarding the memory contents (i.e. linking to memory system), appropriately deployed by the graphical user interface implementation of the visualizer to enable interactive inspection of memory contents (e.g. the subwindow entitled ‘NOTEBOOK’ in Figure 27 for the FLIP language, and ‘MEMORY’ in Figure 29 for the JET language). Program tracing is facilitated owing to the large repertoire of execution events that may be handled. For instance, the categories of execution event notifications that can be enabled/disabled during execution for the FLIP language are shown in the left part of Figure 28.

As we will discuss under Section 6.2, every such event corresponds to a notification type of the underlying execution event system. An interesting facility of our method is the support of fine-grained semantic tracing, allowing, among others, stepping at the level of single operators and lifetime of variables (i.e. instantiation and destruction), in comparison to existing source-level debuggers.
Figure 28. The FLIP visualizer with tracing/step control, memory inspection, program trace, output and console input.

The implementation overhead to accomplish the latter is minimal, requiring: (i) the introduction of the necessary notification categories; and (ii) injection of code triggering notifications upon operator evaluation and variable instantiation/destruction. The tracer of the LEAF language is a little different, since there is no need for memory inspection. Currently, the only facility offered is a type of pretty-print C-style display of the functions during evaluation, typically relying on string conversion virtual methods for all classes of LEAF expressions in the semantic implementation.

For instance, the example in Figure 30 shows the evaluation of three expressions. The first is the evaluation of a max function having a single argument named set, with an actual argument list of numeric parameters (7.000000...1197.000000), with a result of the value '1197.000000'. The second concerns the evaluation of a generic compose(f,g) function, for two function arguments, returning a value being the composite function. Finally, the third concerns the evaluation of the composite function for two numeric actual arguments. The incorporation of tracing hooks together with a graphical or console execution visualizer and tracer for the LEAF language is straightforward following our technique: event generation is hooked within the evaluation methods of expressions, while the visualizer may trace the type of evaluated expression with its pretty-print C-style code. More sophisticated tracers may construct expression trees as well, for more comprehensive
visualization of the functional implementation style relying on expression nesting. In this context, it is important to note that such a visualizer would constitute a valuable instrument, since the availability of source-level ‘debuggers’ and visualizers for functional programming languages is severely limited.

6.2. Event management

The event management system is split in two parts: (a) the basic execution event system, which is a key component of the architecture as briefly introduced earlier, implemented as a traditional event system with event types and associated callbacks; and (b) the event generation hooks, concerning the encapsulation of event notification calls within the syntactic and semantic implementation of the source language. We have already discussed cases of embedded event notifications in the implementation of syntactic patterns, such as notifications for control-flow statements (e.g., if else, while do, case,
break, continue) or automatic memory allocation/disposal for aggregate variables (e.g. arrays, records). In Figure 31, the names of the event notification functions for the FLIP language are listed.

An example illustrating the various implementation components needed to effectively accommodate the management of execution tracing for the block-enter execution event is provided within Figure 32. As shown, the execution event manager is a simple singleton class named ExecutionEvents, which plays the role of a call-back holder with a ‘single-callback per event’ registration policy. Event generation (notification) is hooked within the corresponding syntactic pattern, i.e. BEGIN being the block start marker. Finally, the VisualiserSuper is the base class for visualizers, offering the implementation of the respective callback, for example OnBlockEnter, which calls the BlockEnter local, the latter tracing a ‘block entered’ message. The tracing policy, i.e. how a tracing message is actually handled, is up to derived visualizer classes, indicated by the fact that Trace is a pure virtual method of the VisualiserSuper class.
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

class ExecutionEvents {
    static void (*onBlockEnter)(unsigned line);
    static void setOnBlockEnter (void (*)(unsigned)) { onBlockEnter = f; }
    static void NotifyBlockEnter (unsigned ln) { (*onBlockEnter)(ln); }
    ...
};
#define NOTIFY_BLOCKENTER (1) ExecutionEvents::NotifyBlockEnter(1)
#define BEGIN (...) NOTIFY_BLOCKENTER(__LINE__);
class VisualiserSuper { ...
    virtual void Trace(const char* format,...)=0;
    static void OnBlockEnter (unsigned line)
    {
        if (GetVisualiser())
            if(GetVisualiser()->IsNotifyingEvent("BlockEnter"))
                GetVisualiser()->BlockEnter(line);
        if(GetVisualiser()->IsSteppingEvent("BlockEnter"))
            GetVisualiser()->stepWait();
    }
    void BlockEnter (unsigned line){ Trace("Block entered %d", line); }
    void Initialise (void) {
        ExecutionEvents::SetOnBlockEnter(OnBlockEnter);
        SetNotifyingEvent("BlockEnter", true);
        SetSteppingEvent("BlockEnter", false);
        ...
    }
};

Figure 32. The necessary entries to accommodate tracing of the ‘block enter’ event.

7. DISCUSSION

7.1. Mixed-language development support

Mixed-language programming is very common for large-scale application development, if different aspects of the application system may be better addressed via special-purpose micro-languages. Typically, in such cases, a key part of the system is built in a main language, like C++, while the rest is implemented in the various special-purpose languages. In this context, the overall proposed method, in particular language grammar emulation via coding patterns that pass as eligible source code, is arguably a more effective recipe compared to crafting a typical parser per embedded language, as the latter has the following disadvantages:

- separate distinct parsers need to be developed explicitly;
- each parser implementation should locate and analyze embedded language snippets, generating the respective code in the main language that is then mixed with the main language code; this makes the resulting source code highly unreadable and the debugging process very tedious and non-intuitive as programmers view the integrated generated code;
System built in a main language and multiple embedded languages

- Source code in embedded languages is mixed with other types of source code
- Source code in embedded languages is exclusively written as standalone files
- Every source file involves at most one embedded language and the main language
- Some source files involve more than one embedded language, and, possibly, code in the main language
- Embedded languages that co-appear do not have common syntactic constructs
- There is common syntax among some co-appearing embedded languages
- Common syntax concerns also common keywords
- Common syntax concerns common operators and / or syntactic structures

The only case requiring grammar repair

Figure 33. All possible scenarios for mixing languages showing the only scenario that may lead to a conflict, which, however, requires trivial grammar repair (i.e. changing keywords).

- parallel language-specific source-level debugging, tracing and visualization is not possible, as it is with our method; the main language debugger should be deployed for all purposes.

In addition, the proposed implementation approach is very reliable. More specifically, every embedded language, following our proposed architecture, is delivered with its own runtime system and data management that may appropriately interoperate with the main application system in a way that is dependent on its underlying semantics and runtime library. For instance, in the context of video games, character animation specifications internally cause the instantiation of animation structures via calls to the underlying game engine system. Apparently, the implementation of each distinct language runtime in a way orthogonal to the remainder of the languages is really a matter of typical software engineering practice so as to implement the language-specific semantics, not essentially an issue inherent in, or further complicated by, the proposed implementation method. Regarding the potential of syntactic conflicts among the various deployed embedded languages, Figure 33 indicates all plausible different cases that may naturally appear. As shown, only the top right case, concerning the possibility of common keywords, leads to a conflict. This is easily repaired by choosing non-conflicting keywords. The reason that common operators do not introduce conflicts is due to the fact that the C++ operators can be naturally overloaded, so distinct classes of the different embedded language implementations may overload the same C++ operators, without causing conflicts when those are deployed by different specific types in the various embedded language code snippets. Finally, common syntactic structures are no issue at all, as they do not explicitly map to named constructs in our method, but indicate commonalities in structure implicit in the way the grammar patterns are implemented in the source language.

Mixed-language style, as implemented with the proposed method, can become particularly powerful in cases where, during runtime, data to the embedded language units may be passed by preceding code units implemented in the source-host language. As discussed earlier, the first time a similar technique was put into practice concerns the Boost Spirit library, which is planned to become part of the next C++ Language Standard. In Boost Spirit, EBNF-like code snippets to generate recursive descent parsers...
can be mixed with C++ statements, the latter essentially injecting runtime conditionality on the actual recursive descent parser outcome.

7.2. Supporting hybrid programs for learning purposes

This feature, enabled by the proposed method, is very advantageous in the context of learning-based languages as we now explain. Typically, learners are introduced to a target language that is appropriate for larger-scale programming purposes like C, and are appropriately taught the transition from a source program in the initial language (FLIP in our case) to an equivalent program in the target language (C in our case). This transformation is systematic, and is described as an iterative process with well-defined incremental substitutions for specific language constructs, both at the lexical and syntactic level; the purpose of each such partial program modification is denoted and explained, until the text becomes a pure program in the target language. In this process, while students have to assimilate every incremental program transformation, they are not able to actually evaluate or test how their intermediate programs behave, or even to introduce additional code in the target language relevant to the type substitution just applied (e.g. for experimentation purposes). From real-life experience in applying this method for learning purposes we have reached the conclusion that it is always better to provide such a facility, i.e. fully working hybrid programs, than not supporting it at all.

It should be noted that in this process every intermediate program represents an intermediate language with a well-defined non-vague syntax. This can be informally shown as follows (a formal definition needs far more space, although it is also trivial). Let $L_s$ and $L_t$ be the source and target languages, respectively, with their own grammars, and let the informal notation $L_s < A_s \rightarrow A_t >$, where $A_s$ and $A_t$ are the grammar elements of $L_s$ and $L_t$ respectively, denote the language resulting by substitution of $A_s$ by $A_t$. In particular, $A_s \rightarrow \varepsilon$ means $A_s$ is removed, and $\varepsilon \rightarrow A_t$ means $A_t$ is introduced. Hence, $L_t$ will result from a finite sequence of such transformations over $L_s$. Every intermediate program complies with the previous language, minus the removed and substituted features, plus the introduced and substituting features. Finally, the only issue is whether the incremental program transformation process is really an appropriate teaching strategy. This is essentially a question not related to the proposed method of enabling compilation and execution of the intermediate programs, but to the general challenge in teaching the departure from one language to another of a similar style. In this context we do follow the common practice of stepwise transformations, while ordering the $A_s \rightarrow A_t$ transitions overall as follows: built-in types, declarations, type definitions, operators, constants, branches, loops, console input/output, address, allocations and functions. In this way, students gain a better insight into programming languages, based on the semantics rather than on the syntax, and thus start to distinguish and separate meaning from form.

7.3. Handling potential grammar adaptations

Grammar adaptation in our context is not a difficult task, since it is not related to ambiguity resolution, left factoring or left recursion elimination, to name the most prominent cases of repair for context-free grammars during compiler construction. In our case, adaptation merely prescribes the modification of a non-ambiguous grammar either to allow, or in some cases simply to make easier, its syntactic emulation via C++ language elements. In most cases, the required modifications become apparent once source language sentences cannot be programmed as C++ language expressions. In particular,
such modifications aim to bridge the gap between the particular source language expressions and the candidate C++ expressions that aim to emulate them, in an iterative way, until the subject source expressions become directly eligible as legal C++. This bridging is performed from both sides: either the grammar is modified to move closer to the current version of the corresponding C++ expressions, or the C++ expressions are modified to approach the source language syntax. The latter is usually more demanding, while the former is mostly trivial. In practice, the overall technique is heuristic in nature, being accompanied by the profound tradeoff between the potential of syntactic-usability compromise, inherent in the necessity of adopting in some cases more verbose syntactic notations to accommodate C++ emulation, and the significant gains due to ease of development and the support for runtime interoperability, inherent in the fact that the host language boundary is never crossed [2].

A comprehensive example is provided in Figure 34, demonstrating the steps to accomplish grammar adaptation for sentences of a functional programming language close to LEAF, adopting a Lisp-like syntax for simple function definition expressions. We show the progressive adaptations to achieve emulation in C++, with an iterative process relying on the processing of sentences in an outside-in and left-to-right fashion.

8. SUMMARY CONCLUSIONS

We have reported an implementation technique for the rapid development of languages whose source programs compile as C++ code. The method relies on the codification of syntactic patterns in the C++ language, thus mapping source code patterns to target C++ coding patterns, where calls to the underlying language semantics are encapsulated inside such syntactic patterns.

We have successfully applied the method for three different languages, each pertaining to a different programming paradigm, i.e. an assembly language, a high-level procedural language and a functional programming language. An important technical advantage of the proposed method is the ease of implementation and extension of execution tracers and visualizers. This is due to the fact that the encapsulation of code triggering execution events, the latter to be handled by such tracers and visualizers, is trivially accommodated within the syntactic patterns and the semantic implementation. Notably, the capability to track down a large repertoire of semantic execution events (i.e. events strongly linked to the semantics of the source language) is not offered by typical source-level debuggers.

In particular, we consider that a specific scenario for which the technique is highly appropriate concerns domain-specific proprietary (i.e. in-house) scripting languages. In a way similar to compiled or interpreted techniques, the scripting language may be implemented to eliminate the possibility of programming defects interfering with the host library through a 'closed-world' memory system for the scripting language. For example, we have applied this method in the FLIP language, so that there is no way for a user program to cross the boundaries of the FLIP language memory system. This remark implies that many of the benefits of adopting a scripting language, like ease of use and minimal interference with the host system, can also be gained with the proposed approach. The latter is possible not only without a compromise, but with the additive advantages due to the capability of providing rapidly more comprehensive support instruments.

Our technique is dependent on the C++ language, since it relies on a blending of preprocessor tricks, templates, overloading, inheritance, etc., to accommodate the desirable syntactic patterns and
IMPLEMENTING LANGUAGES TO COMPILE AS C++ WITHOUT CRAFTING A COMPILER

1: Enclosing parenthesis disable \( f \) to be a local variable declaration, so they are dropped. 2: \( \text{def} \) can be a macro for a ‘function’ class of the semantic implementation, so \( f \) becomes a normal local variable. 3: The \((x,y)\) that follows is assumed to be a constructor signature, so we leave it for the next step. 4: We also need a trailing semicolon since functions definitions are expression statements for C++. 5: \( x \) and \( y \) as locally defined variables in this context are not allowed. The only alternative is to refer to them as strings inside the function expression tree, so we add an \( \text{arg} \) macro to essentially turn such ids to strings-arguments, recorded appropriately by the underlying semantics. 6: The argument expression list should handle any number of arguments, such as \( \text{arg}(x), \text{arg}(y), \text{arg}(z) \), for example. This is only possible by overloading the comma operator for \( \text{arg}(x) \) type expressions. But such overloading will not work if the expression list is tested against a function signature. There are two ways to resolve this. 7: put parenthesis around the arguments group. Or 8: turn \( \text{def} \) to a macro accepting \( f \) as an argument, internally producing a temp instance that collects the expression list. Both alternatives are shown below. Notably, the second alternative allows parenthesis to be put back around the function definition, as \( f \) name may now become a string instead of a local variable. 9: Prefix operator notation is not allowed in C++. 10: Also, the binding of \( x \) and \( y \) inside \( f \) expression tree is feasible only via strings, not ids. So we introduce \( \text{var} \) and we use normal overloaded infix +. 11: The space in C++ cannot be overloaded, so we need a concatenation operator to link \( f \) to its trailing expression. We overload the assignment operator ‘=’ for this purpose. 12: As an alternative syntax to associated the function with its expression we may use the overloaded array subscript \([\,]\) operator, with the expression type as the index argument. This gives the syntactic impression of a surrounding block. The grammar is now ready, forming legal C++ code, so we have a semantically equivalent notation, where the adapted one became more verbose. However, the speed in implementing the syntax, and also preparing the ground to easily accommodate the semantics for lazy evaluation, is significant.

Figure 34. An example of progressive grammar adaptations to achieve syntactic emulation.

semantic implementation. Apparently, apart from the more elaborated cases discussed under the section on related work, we assume there are numerous cases where techniques resembling those presented may have been applied at a micro-scale, to allow programming with more compact expressions, internally mapping to more complicated C++ coding patterns. However, we are not aware of the broader application or formulation of such methods as an effective language development paradigm, alternative to the prevalent code of practice driven by compiler construction theory.

The method is also very flexible when it comes to combining multiple languages within a single application development project, as far as there are no conflicts among the various syntactic patterns and the semantic implementations. Mixed style development with micro-languages rapidly implemented to fit discrete development roles is a facility that may meet the needs of various domains. In particular, it is common in the context of game development to separate animation specifications from state-based character intelligence; for example, the support of state-based specifications relying on preprocessor
syntactic abstractions is reported in [13]. Apart from the various benefits of the approach, there are also
a few shortcomings that have to be manually addressed on a case by case basis by programmers. More
specifically, the key difficulty is that syntax emulation is not prescriptive following an algorithmic floor
plan, but should be handled by implementing programming patterns similar to those discussed for the
tree language examples. Essentially, the proposed approach is a heuristic purpose-driven alternative
to well-established compiler construction paradigms, since we did not manage to formulate an algo-

rithmic method for syntactic pattern derivation from its corresponding grammar rules. Consequently,
programmers are put in a position where they seek in an exploratory fashion the programming of the
syntactic patterns, until either emulation is successful or the grammar is eventually modified. In this
context, we have tried to report the general techniques towards syntax emulation, so that sentences of
the source languages can be reduced through operator evaluation to expressions of the target languages.
In conclusion, the key competitive advantages of this development technique are mainly the support
for rapid language development, the increased modifiability and extensibility owing to the small turn-
around time in applying language changes, the support for mixed-language programming and the
capability to modularly introduce execution tracing and visualization instruments.

ACKNOWLEDGEMENTS

The reported language cases have been employed in the Fall semesters of 2003, 2004, 2005 and 2006, in the context
of the course CSD100 ‘Introduction to programming’ taught by the author at the Department of Computer Science,
University of Crete. The valuable feedback of all students is highly appreciated. All code for the language cases
discussed in the paper is publicly available from http://www.ics.forth.gr/hci/files/plang/FLIP_LANGUAGE.ZIP,
LANGUAGE.ZIP.

REFERENCES

1. Savidis A. Introductory teaching of imperative programming through an anthropomorphic computation model.
2. Abrahams D, Gurtovoy A. Domain-specific embedded languages. C++ Template Metaprogramming, ch. 10. Addison-
5. McNamara B, Smaragdakis Y. Functional programming with the FC++ library. Journal of Functional Programming
   the ACM SIGMOD 2000 International Conference on Management of Data (May 16–18), Dallas, Texas, 2000; 583–583.
    221–236.