Universitatea Alexandru Ioan Cuza Iaşi
Facultatea de Informaticã

Lucrare de Dizertație

Genesis :
A C++ Library For Textual DSLs

propusă de

Ştefan Silvestru

Sesiunea : iulie, 2011

Coordonator științific
Prof. Dr. Gheorghe Grigoraş

Universitatea Alexandru Ioan Cuza Iaşi
Facultatea de Informaticã

Genesis :
A C++ Library For Textual DSLs

propusă de

Ştefan Silvestru

Sesiunea : iulie, 2011

Coordonator ştiinţific
Prof. Dr. Gheorghe Grigoraş

Declarație privind originalitatea şi respectarea drepturilor de autor

Prin prezenta declar cã Lucrarea de dizertație cu titlul *"Genesis : A C++ Library For Textual DSLs"* este scrisã de mine şi nu a mai fost prezentatã niciodatã la o altã facultate sau instituție de învãțãmânt superior din țarã sau strãinatate. De asemenea, declar cã toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate in lucrare, cu respectarea regulilor de evitare a plagiatului :

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altã limbã, sunt scrise între ghilimele şi dețin referința precisã a sursei;
- reformularea în cuvinte proprii a textelor scrise de cãtre alți autori deține referința precisã;
- codul sursã, imagini, etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi dețin referințe precise;
- rezumarea ideilor altor autori precizeazã referința precisã la textul original.

Iaşi, 23 iunie 2011

Absolvent *Ştefan Silvestru*

____

(semnãtura în original)

Declarație de consimțământ

Prin prezenta declar cã sunt de acord ca Lucrarea de dizertatie cu titlul *"Genesis : A C++ Library For Textual DSLs"*, codul sursã al programelor şi celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare sa fie utilizate în cadrul Facultãții de Informaticã. De asemenea, sunt de acord ca Facultatea de Informaticã de la Universitatea Alexandru Ioan Cuza Iaşi sã utilizeze, modifice, reproducã, şi sã distribuie în scopuri necomerciale programele-calculator, format executabil şi sursã, realizate de mine în cadrul prezentei lucrãri de dizertație.

Iaşi, 23 iunie 2011

Absolvent *Ştefan Silvestru*

____

(semnãtura în original)

Acord privind proprietatea dreptului de autor

Facultatea de Informaticã este de acord ca drepturile de autor asupra programelor-calculator, format executabil şi sursã, sã aparțină autorului prezentei lucrãri, *Ştefan Silvestru.*

Încheierea acestui acord este necesarã din urmatoarele motive:

Iaşi, 23 iunie 2011

Decan *Gheorghe Grigoraş*	Absolvent *Ştefan Silvestru*
____________________________	____________________________
(semnãtura în original)	(semnãtura în original)

Contents

Introduction 1

Contributions 3

1 Domain-Specific Languages 4
- 1.1 Overview 4
- 1.2 DSL Types 5
  - 1.2.1 Textual 5
  - 1.2.2 Graphical 8
- 1.3 Opportunities and Drawbacks 9
- 1.4 Examples 10
- 1.5 Design Methodology 12

2 Genesis Library 14
- 2.1 Overview 14
- 2.2 Spirit Parser Framework 16
- 2.3 Design 18

3 Implementation 22
- 3.1 Genesis API 22
- 3.2 DSL Structure 23
- 3.3 Allowed Syntax 25
  - 3.3.1 Literals 25
  - 3.3.2 Bracket Expressions 25
  - 3.3.3 Parsers 26
  - 3.3.4 Operators 28
- 3.4 Grammar Options 31
  - 3.4.1 Grammar Definition 31
  - 3.4.2 Skipper 32
  - 3.4.3 Modes of Operation 34
- 3.5 Semantic Actions 35

4 Usage Scenarios 39
- 4.1 Simple Poll 39
- 4.2 Shapes 41
- 4.3 Security Protocols 43
- 4.4 Packet Filter 45

Conclusions and Future Work 48

References 49

Introduction

The first electronic computers (c.1940) were programmed in machine language using sequences of 0s and 1s, instructing the computer what operations to execute. The operations were very low level (move data from one place to another, add two values, compare two values, etc), thus, making programming slow, difficult and error-prone, programs being hard to understand and modify. The move to higher-level languages was natural. With the pass of time, new programming languages were created, each trying to add original features in order to make programming easier and more natural [B1].

Nowadays, there are myriad of programming languages, which can be classified in different ways, one of them being by generation. Machine-level languages are first-generation languages (1GL), second-generation (2GL) are assembly languages, and third-generation (3GL), the higher-level languages like Fortran, Cobol, Lisp, C, C++, Java and C#. Fourth-generation (4GL) languages are languages designed for specific applications (eg. NOMAD – report generation, SQL – database queries, MATLAB – numerical computing, Postscript – text formating). Fifth-generation (5GL) languages are based around solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages, are fifth-generation languages (eg. Prolog, OPS5) [B1].

The 4GL followed the 3GL in an upward trend toward higher abstraction and statement power. 4GLs have often been compared to Domain-Specific programming Languages (DSLs), and are considered a subset of DSLs [A1, A2]. While 4GLs are designed to build specific programs, 5GLs are designed to make the computer solve a given problem without the programmer.
The programmer needs to specify the problems to be solved and what conditions must be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in artificial intelligence research. 4GL and 5GL projects are more oriented toward problem solving and systems engineering.

We can also make a classification based on the amount of abstraction between a certain language and machine language. 1GLs and 2GLs are usually described as being "close to the hardware" because of their low level of abstraction. 3GLs are high-level programming languages because they provide strong abstraction from the details of the computer. The purpose of this greater abstraction and hiding of details is intended to make the language user-friendly, as it includes concepts from the problem domain instead of those of the machine used. Generally, it's much easier to write programs in higher-level programming languages, but it's less efficient as the target programs tend to run slower. On the other side, using a lower-level language offers a programmer more control over a computation, being able to produce more efficient code, but it's harder to write programs, which are also less portable, harder to maintain, and error-prone.

Let's take for example C++ and Java. Even though they are both high-level languages, they still differ, with regard to the provided level of abstraction. Thus, C++ is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features, whereas Java goes high-level and above, providing a simpler object model (than C++) and fewer low-level facilities. The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. That's why Java is much popular these days than C++. Nevertheless, C++ is still the language of choice when it comes to high-performance software, ranging from server applications to operating systems.

Although being very powerful, C++'s policy is to provide all the necessary ingredients and

let the programmers cook the dishes they want. Even Bjarne Stroustrup, the designer and original implementor of C++, admits that "without a good library, most interesting tasks are hard to do in C++; but given a good library, almost any task can be made easy", suggesting the need for some form of higher components (libraries in this case), for an efficient use of the language.

When dealing with complex problems, the solutions tend to get large and only a small part of the code is being used to actually solve the problem, the rest of the effort being put into writing boilerplate code. Thus, most of the time it's hard to understand the solution and even moving the generic code into libraries doesn't solve the problem. And all that because we have to express our solutions in the (fixed) base language, being unable to extend it with new, user defined, syntactic constructions. We can increase productivity by providing programmers with languages that are close to problem domain, but creating a new full-blown language for each problem we try to solve, can get too expensive and unfeasible [A3].

A natural solution to overcome the aforementioned problems would be to combine DSLs with a General Purpose programming Language (GPL) (like C++ or Java). Most solutions available today target Java or other newer languages, solutions based on and targeting C++ being inexistent. Thus, we propose Genesis, a C++ library meant to ease the development and use of textual DSLs. The base language doesn't provide us mechanisms to create new syntactic constructions, so instead, we create language extensions with Genesis and mix them with our C++ code. Then, these extensions are translated into plain C++, and we can build the code like any regular C++ program.

Genesis provides a generic solution for code generation, so we're not bound to generating only C++ code. A Genesis user creates new languages and also decides what should they generate, that being any kind of text (plain text, C++, Java, HTML, XML and so on). But, only text generation based on some rules doesn't always suffice. Beside code generation, Genesis allows a user to define custom actions (written in C++) to be run when a rule or subrule of the language is matched. Hence, if we ignore the code generation component and assign semantic actions to rules, the result will be an interpreter for the defined language.

The rest of the thesis is outlined as follows: In Chapter 1 we present a more detailed view on DSLs. Chapter 2 introduces the Genesis library and its design. In Chapter 3 we describe its implementation thoroughly, and in Chapter 4 we present some usage scenarios of it. Finally, we summarize and propose certain directions for further improvements.

Contributions

Current trends in software engineering show the increasing needs for tools which should help software engineers enhance the quality, productivity, reliability, maintainability, portability and reusability of both source code and applications. More than that, whenever possible, they should give the ability to work in terms of the problem space, so that even domain experts could understand, validate, modify, and often even develop applications. Some or even all these benefits can be achieved by (using) DSLs. Although not new, DSLs gain more and more attention, mainly because of the need for extra levels of abstractions. DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Once a DSL is made, we can start creating instances of it, which will either execute or generate code (usually) for a GPL, depending on how the DSL is implemented.

Our work has materialized in the creation of Genesis, a C++ library which helps its users to develop and use textual DSLs. Genesis has two main features: code generation and the use of semantic actions. The code generation utility can generate any kind of text (from plain text to any GPL source code), whereas semantic actions must be written in the base language (C++) in order to execute. The main contribution is that Genesis targets the C++ language because, even though similar software exist, they target languages like Java or C#. The host language is very important because, even though we write our programs in a higher level of abstraction, these domain-specific languages either translate into or execute code which is written in the base language. Thus, the power of DSLs is directly dependent on the power of the GPLs used.

DSLs cover a wide range of application domains, like software engineering, systems software, multimedia, telecommunications or security, just to name a few. The security domain is a very important one, and a large number of DSLs were created for its different fields: network monitoring and network intrusion detection [A23, A30, A31, A9, A46], vulnerability analysis on a network [A38], specification of security protocols [A43, A47], cryptographic algorithms [A44], and so on. With Genesis we managed to create several DSLs in various fields: generate questionnaires, manipulate 3D objects, and the most important, in the security domain, specify security protocols, and capture packets over the network and analyze them.

Chapter 1

Domain-Specific Languages

Domain-Specific Languages (DSLs) have been around for a long time but, until now there hasn't been any general treatment of the techniques and characteristics of DSLs in general, as opposed to the traits of a particular DSL. Thus, this chapter is mainly based on Fowler's book on DSLs [B2]. Alongside, there will be references to various articles, to support the statements made.

1.1 Overview

"Programming languages are notations for describing computations to people and to machines"[B1] The world depends on programming languages, because all software running on computers was written in some programming language. In Introduction, we made a brief classification of programming languages. GPLs are the most common and the most used languages, but they can't always provide the needed abstractions. Thus, in search for some higher level of abstraction, we turn our eyes on languages that are closer to the domains of the problems that we're trying to solve. We call these languages, domain-specific. Before we go further, let's see what DSLs are. In order to get a general understanding, we provide the following list of definitions :

- "Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain" [B2]
- "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [A1]
- "A domain-specific language (DSL) is a language whose purpose is to simplify writing programs for that problem domain" [A4]
- "DSLs are generally very high level languages tailored to specific tasks. They are designed to make their users particularly effective in a specific domain" [B3]
- "A Domain-Specific Language is a custom language that targets a small problem domain, which it describes and validates in terms native to the domain" [B4]
- "A DSL is a language designed to be useful for a specific set of tasks" [B5]

The key characteristics of DSLs according to these definitions are : *effectiveness* and *problem domain specificity*. DSLs help programmers (or domain experts) in writing programs using idioms that are similar to the abstractions found in particular problem domains, making their work more effective. Consider the following example :

```
SELECT * FROM Book
WHERE price > 20.00
ORDER BY author;
```

It's easy to recognize this as an SQL query. Just think of how you would implement this in a GPL, and also, do this over and over again. Also, how easy would it be to understand and modify that code, compared to an SQL query? Or what about regular expressions? Regular expressions and SQL are similarly specialized languages, designed for a narrow domain—text processing and database querying, and they both are focused on letting users express what they mean, instead of how some particular things should be implemented (i.e. how the implementation should work) [B6].

Different tasks (eg. low level hardware manipulation) must be done at different levels, but we, as developers strive for simple, readable and maintainable code. Also, computer programs can scale and turn into complex artifacts (eg. enterprise systems), which are hard to build, manage, understand and evolve. Existing software development paradigms don't always cope with challenges such as system size, domain complexity and software evolution, when considering such systems. In order to tackle some of these problems, we need to elevate program specifications. By raising the overall level of abstraction, DSLs make certain kinds of code easier to comprehend, write and modify thus, improving programmer productivity and increasing software quality. The code is easier to manage and analyze, and it's also less error-prone.

DSLs provide convenient abstractions, shielding application developers from much of the complexity and variability of explicitly programming in GPLs. Effective design of such abstractions requires close interaction between researchers developing such languages, and domain experts with a deep understanding of the problems to be solved. This issue is also the second definitive characteristic and benefit of DSLs : improving communication with domain experts.

DSLs are usually small and easy to understand, allowing business people (i.e. non-programmers) to see and directly comprehend the code that drives important parts of their business (i.e. code that implements their business rules). Thus, domain experts can communicate in a much more detailed fashion with the programmers who actually write the code. By exposing the real code to domain experts, we enable a much richer communication channel between programmers and their customers. "This communication with domain experts is a benefit more difficult to achieve, but the resulting gain is much broader because it helps unclog one of the worst bottlenecks in software development – the communication between programmers and their customers" [B2].

1.2 DSL Types

When referring to DSLs, we often distinguish between the textual and graphical types. We'll discuss those types in turn, and look at their properties and uses. But before doing that, we'll discuss the terminology used.

The structure of a DSL is captured in its abstract syntax – also known as its metamodel. A metamodel is a model used to construct another model, where one model is expressed in terms of the other model (is an instance of that model). Simply put, the metamodel is used to define a DSL's abstract syntax, leaving users of that DSL to create instance models out of it. The term abstract syntax refers to a metamodel, having a corresponding concrete syntax in the form of text or diagram notation, which are referred to as textual concrete syntax and graphical concrete syntax. A textual syntax let's users work with textual instance models (just like any other text-based programming language), whereas a graphical syntax enables its users to work with instance models using a diagram surface, the most popular being UML [B5].

1.2.1 Textual

When talking about textual DSLs, Fowler in [B2] distinguishes between internal and external DSLs.

Internal

Internal DSLs are built on top of existing GPLs, and use the syntax provided by these general languages. Internal DSLs are a stylized use of these languages for a domain-specific purpose, trying to express things in a way that makes sense to both the author and the reader, and not to the compiler. The expressiveness of an internal DSL is both influenced and limited by the constraints that are imposed by the underlying language syntax. The general language shouldn't be too rigid in its syntax in order to build such a DSL, so languages like Java and C# aren't a good choice. Instead, popular choices for building internal DSLs are dynamic languages like Lisp, Ruby or Python. These kind of languages have other features that are useful when building DSLs, like closures, macros or duck typing. The major advantage of an internal DSL is that it preserves the language environment, so for example, you won't need to redefine an if statement or operator precedence, because you have those in the host language. Another term by which internal DSLs are known, are **embedded** DSLs [B2, B6]. We will present an example of an embedded DSL in Chapter 2.

A variation of internal DSLs are the **fluent interfaces**. This term was coined by Eric Evans and Martin Fowler [W3, B2] to describe more language-like APIs (Application Programming Interfaces). Fluent interfaces are ways to structure your API so that operations flow naturally and provide more readable code. They are mostly valid only when used by developers during actual development, which limits their scope compared to other DSL types [B2, B6]. The central pattern of a fluent interface is that of Method Chaining. Our initial code (in C++) might look like this [W4]:

```
class GlutApp{
//class implementation
};

int main(int argc, char **argv) {
     GlutApp app(argc, argv);
     app.setDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_ALPHA |GLUT_DEPTH);
     app.setWindowSize(500, 500);
     app.setWindowPosition(200, 200);
     app.setTitle("GlutApp");
     app.create();
}
```

We can create a new class, FluentGlutApp derived from GlutApp, in which each member function calls the correspondent method from the base class and returns `*this`. For example, let's see the new method which sets the name of the application :

```
FluentGlutApp& FluentGlutApp::named(const char *title) {
     setTitle(title); //from base class
     return *this;
}
```

Thus, we achieve Method Chaining and we can express the same thing as above, with [W4]:

```
class FluentGlutApp : GlutApp{
//class implementation
};

int main(int argc, char **argv) {
       FluentGlutApp(argc, argv)
          .withDoubleBuffer()
          .withRGBA()
          .withAlpha()
```

```
        .withDepth()
        .at(200, 200)
        .across(500, 500)
        .named("GlutApp")
        .create();
}
```

The greatest benefit of internal DSLs is that they are "end user readable". But, it's also important to know the limitations of internal DSLs. When using them, you are constrained by the host language, and any expression must be a legal one with regard to the (base) language. Thus, you must analyze this option with care, especially if you're going to use them extensively in a large project. Otherwise, spending some time in creating an external DSL may pay off, over the life of a project.

External

External DSLs are written in a different language than the host language of the application, and are transformed into it using a compiler or interpreter. These are required because external DSLs have custom syntaxes. External DSLs differ from internal DSLs in that the parsing process operates on a text input which is not constrained by any particular language.

When building a new DSL, we should give it its own unique and individual syntax, in order to provide language constructs which are designed with both the problem domain and the target audience in mind. We would use an external DSL for specifying languages that are too far afield from existing programming languages. SQL and regular expressions are two common examples of external DSLs. They couldn't be built as internal DSLs because they differ too much from general languages, so external DSLs would be the right approach for writing our own SQL dialect or custom regular expressions.

Next, let's see another example. Imagine you are trying to simulate in a very simple manner, the movement of a robot. First, you need to set up the board on which it moves, a starting position and the position and size of the camera. Then you can move it up, down, left, right or at a certain position. An usage scenario of this language might look something like this:

```
Set board: 30x30 squares of 10 units.
Set camera size: 400 by 300 pixels.
Set camera position: 10, 20, 10.
Set initial position: 0, 0.
Move up.
Move down.
Move left.
Move right.
Move up 3 times.
Move down 2 times.
Move to 20, 20.
```

Fragmentary and Stand-alone

We introduce here two ways in which DSLs may appear: fragmentary and stand-alone [B2].

The last example we showed (the movement of the robot) is a stand-alone DSL. Basically, you look at a block of DSL script (usually a single file) and it's all DSL. Someone who is familiar with the DSL but not with the base language of the application, should understand what it does because, the host language isn't present (for an external DSL) or is suppressed by the internal DSL.

Another way DSLs appear is in a fragmentary form. In this form, pieces of DSL code are intermingled with host language code so, without understanding the base language you can't follow what the DSL is doing. The already ubiquitous examples are SQL and regular expressions. For example, you don't have a whole file with regular expressions, but instead, you have little snippets which are mixed with the regular host code. You can think of them as enhancing the host language with additional features. Same applies to SQL – if used with a GPL, you write SQL statements in the context of a larger program. SQL is a good example of DSL which can be used in both stand-alone and fragmentary contexts.

1.2.2 Graphical

Graphical DSLs use shapes and lines to express intent rather than using text. A good example is UML, a DSL for describing software systems. You model your problem in UML and from that, you can generate the rest of the application. Figure 1 displays a small part of a typical UML diagram.

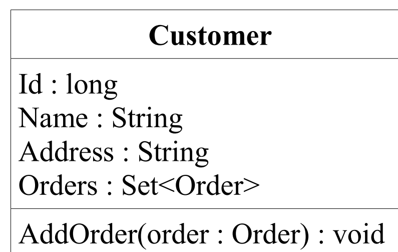

Figure 1. UML Class Diagram Displaying The Customer Object

Graphical DSLs help you express a lot of information in a concise way, making it much easier to understand a problem by seeing it, than when it's explained in words. This approach also allows a higher level communication because of the limitations of the image, making it much easier to understand what's going on when you see the big picture [B6].

A lot of effort has been put to make it possible to write your own graphical DSLs. A framework which lets you build tools similar to the class designer and generate code with them, is Microsoft's Visual Studio DSL Tools. Let's start with an example that captures data for deploying and managing distributed applications. Figure 2 shows a simple model built using a graphical DSL (which is part of Visual Studio 2005 Team Architect) for designing logical data centers. The elements of this language include zones – the rectangular areas surrounded by dashed lines, hosts – the rectangular areas surrounded by solid lines, endpoints – the small shapes (squares, circles and hexagons) placed on the edges of hosts, and connections – the arrows between endpoints. We can build up graphical DSLs like this one from a set of simple diagrammatic conventions (many of them being derived from UML) [B4].

All these are great, but they all share common problems inherent to the graphical DSL model. Graphical DSLs hide information that you don’t want to see, so you can view the big picture. But by doing so, you can't see the whole at the same time, forcing you to spend a lot of time browsing between various elements on the DSL surface. Also these DSLs need a lot of screen space to express notions that would take only a few lines in a textual DSL. Then again, operations like search and replace are difficult. More than that, difficulties appear when working with graphical DSLs in a team environment. Comparing two versions of a file isn't an easy task at all, and it usually breaks down, making it a problem in terms of source control [B6].

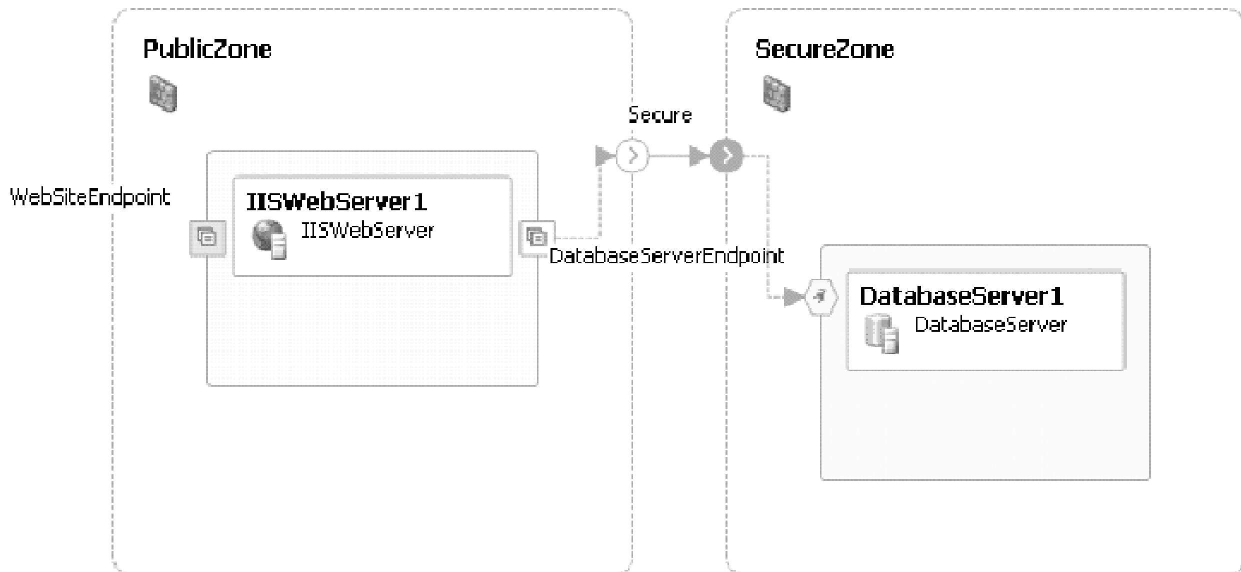

Figure 2. Data Center Design [B4]

1.3 Opportunities and Drawbacks

After seeing what DSLs are, we might ask ourselves what can we gain from using them. DSLs provide limited focus (opposed to object orientation for example), and are a specific tool for very particular conditions. Thus, a project might use several DSLs in various places. As we already said, DSLs aren't more than a facade over a library (or a framework), so we should separate the benefits provided by the two. DSLs have certain advantages and before deciding to use them, we must consider with care which of these benefits are applicable to our circumstances. Next, we present some of the **advantages** [B2, B4, A1]:

- DSLs enhance quality, productivity, reliability, maintainability, portability and reusability
- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain
- DSLs give the ability to work in terms of the problem space, with less scope for making the errors that come from representing it in a general-purpose language
- Models are made more accessible to people not familiar with the implementation technology, including business people. Domain experts themselves can understand, validate, modify, and often even develop DSL programs
- Improve communication with domain experts and between roles in a development team
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge
- DSLs allow validation and optimization at the domain level. Errors in understanding or representation can be picked up much earlier in the development lifecycle, so as long as the language constructs are safe, any sentence written with them can be considered safe
- Models can be used to simulate a solution directly, providing immediate feedback on the model's suitability
- Models can be used to configure an implementation consisting of multiple technologies of different types, which can reduce the skill and effort required to implement a solution using

these technologies

- Models can also be used to generate other models, and to configure other systems, networks, and products, perhaps in combination with other enabling technologies such as wizards

Adopting a DSL approach to software engineering involves not only opportunities, but also risks. Some of the **disadvantages** are [B2, A1] :

- The costs of designing, implementing and maintaining a DSL
- The costs of education for DSL users
- The limited availability of DSLs (ghetto languages) – eg. companies which develop languages which are not used anywhere else (they're only for in-house use), making it difficult for them to find new staff and to keep up with technological changes
- Proliferation of similar, non-standard DSLs
- Lack of proper tooling (Integrated Development Environments (IDEs), debuggers, etc)
- The difficulty of finding, setting, and maintaining the proper scope for a DSL
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs
- The potential loss of efficiency when compared with hand-coded software

There is a cost involved in learning a new language in order for people to understand the code, but we argue that it's a lot easier to understand a code written in a good DSL, than it is to understand GPL code that is poorly written, because the language doesn't have sufficient abstractions for the domain problem.

1.4 Examples

DSLs are definitely not something new. Hundreds of them exist today, but of these, only a subset is described in software engineering literature and used on a larger scale. The most common are classical examples like LEX, YACC, (E)BNF, VHDL, sed, grep, Make, TeX, CSS, HTML, regular expressions, spreadsheet formulas and macros, and SQL, just to name a few. The most known DSLs and their application domains can be viewed in Table 1.

DSL Name	Application domain
SQL	Database queries
HTML	Hypertext web pages
Excel	Spreadsheets
LaTeX	Typesetting
BPEL	Business processes
(E)BNF	Syntax specification
Make	Software building
VHDL	Hardware design
YACC	Creating parsers

Table 1. Some Widely Used Domain-Specific Languages

Other well-known examples and somewhat newer DSLs are Graphviz's Dot language, LINQ, HQL, XQuery, XPath, XSLT, XML Schema, DTD, JavaDoc and SVG.

At the beginning of this chapter we provided an example of an SQL query. To see another DSL example, we introduce Graphviz [A6], an useful package for manipulating graphs and their drawings. Figure 3 shows an example of a finite state machine.

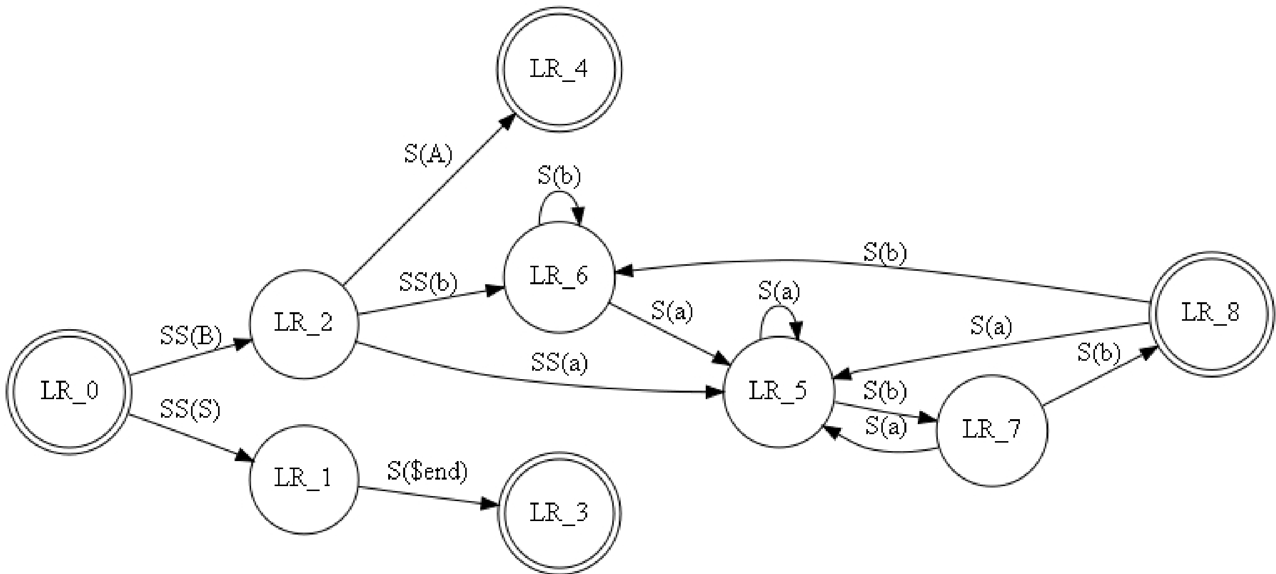

Figure 3. Dot Drawing Of A Finite State Machine [A6]

To create this diagram, you need to write the following code in the DOT language (which is an external DSL) [W1] :

```
digraph finite_state_machine {
      rankdir=LR;
      size="8,5"
      node [shape = doublecircle]; LR_0 LR_3 LR_4 LR_8;
      node [shape = circle];
      LR_0 -> LR_2 [ label = "SS(B)" ];
      LR_0 -> LR_1 [ label = "SS(S)" ];
      LR_1 -> LR_3 [ label = "S($end)" ];
      LR_2 -> LR_6 [ label = "SS(b)" ];
      LR_2 -> LR_5 [ label = "SS(a)" ];
      LR_2 -> LR_4 [ label = "S(A)" ];
      LR_5 -> LR_7 [ label = "S(b)" ];
      LR_5 -> LR_5 [ label = "S(a)" ];
      LR_6 -> LR_6 [ label = "S(b)" ];
      LR_6 -> LR_5 [ label = "S(a)" ];
      LR_7 -> LR_8 [ label = "S(b)" ];
      LR_7 -> LR_5 [ label = "S(a)" ];
      LR_8 -> LR_6 [ label = "S(b)" ];
      LR_8 -> LR_5 [ label = "S(a)" ];
}
```

This example shows how to use nodes and arcs to create a graph. Nodes are declared with the node keyword, and arcs using the -> operator. Both nodes and arcs can be given attributes listed between square brackets. Though not shown in the example, Graphviz has many more useful features for diagrams such as options for colors, fonts, tabular node layouts, line styles, hyperlinks and custom shapes.

The examples presented above are the most known and used ones, but there are various other DSLs covering a wide range of application domains, which can be grouped into the following areas:

- **Software Engineering**: Detection of code clones [A18], processing ad hoc data [A25], evolutionary algorithms [A39, A40], refactoring [A32]
- S**ystems software**: Video device drivers specification [A13], real-time embedded and control systems [A19, A27], programming high-performance server applications [A26], form-based services [A34], distributed systems [A41]
- **Security**: Network monitoring and network intrusion detection [A23, A30, A31, A9, A46], vulnerability analysis on a network [A38], specification of security protocols [A43, A47], cryptographic algorithms [A44], secure multi-party computation [A48], access controls [A42], digital forensics [A14, A15], enforce security policies on web services [A28, A29], semantic web security [A33], securing distributed systems[A24], specify SELinux policies [A35, A36], covert channels [A49]
- **Multi-Media**: Modeling of two-dimensional games [A7], representation of two-dimensional pictures [A37]
- **Telecommunications**: Mobile applications development [A10, A11, A12, W3], internet telephony services [A16, A17], network performance testing [A45]
- **Environment**: Modelling of processes that occur in landscapes [A22], Agriculture [A8]
- **Miscellaneous**: Mathematical equation specification [A20]

1.5 Design Methodology

[A1] presents the steps involved in developing a DSL:

Analysis

1. Identify the problem domain
2. Gather all relevant knowledge in this domain
3. Cluster this knowledge in a handful of semantic notions and operations on them
4. Design a DSL that concisely describes applications in the domain

Implementation

5. Construct a library that implements the semantic notions
6. Design and implement a compiler that translates DSL programs to a sequence of library calls

Use

7. Write DSL programs for all desired applications and compile them

The analysis phase is meant to build up a thorough understanding of the underlying application domain. For acquiring such an understanding, researchers (domain analysts) provide guidelines by investigating ways of modeling domains. For a well designed DSL, these domain analysts should be persons with a lot of background experience in the same problem area, to be able to examine the needs and requirements of a collection of similar systems.

The implementation phase requires the construction of a library which should implement the semantic notions provided from the first phase, and the building of a compiler which should translate DSL scripts into library calls. Essentially, DSLs aren't much different than libraries; "most

DSLs are merely a thin facade over a library or framework" [B2]. Class libraries are reusable implementations of tasks in a certain domain. The library is used based on the provided API which can be viewed as a "language" for using the library implementation. The syntax of this "language" is based on the syntax of the used GPL, which is limited in expressing domain-specific concepts in a natural manner. So, why bother developing DSLs? Simply because they provide domain specificity better. In most cases, DSL programs are translated to library calls and the DSL can be viewed as a means to hide the details of that library. DSLs can provide great benefits (as seen in subchapter 1.3), and should be considered more often. "The facade may be thin, but it is often useful and worth building" [B2].

The last phase implies the creation of DSL programs for the needed applications, and the compilation of these instances, which results in sequences of library calls.

Chapter 2

Genesis Library

2.1 Overview

As we saw in Chapter 1, DSLs are languages that target a specific kind of problem or domain. Although used with success in various areas, they are not widely used in general purpose applications mainly because the popular widely used languages today do not make it easy. The syntax of a GPL does not always allow the appropriate notation and composition of domain concepts. GPLs provide limited facilities for expressing domain-specific concepts in a natural manner, and all domain concepts need to be captured using the same generic syntactic and semantic constructs. Also, developing a DSL usually requires a mature domain knowledge and language development expertise, few people having both. Most people consider that creating your own computer language is a fairly complex matter, because most of the literature assumes that you want to build a complete and mature general language, which is not always the case. Because of that, DSLs are considered at the final stage of the evolution of an object-oriented application or are not considered at all – never getting beyond the application library stage. We consider this an unhealthy approach, because of the potential advantages (as seen in subchapter 1.3) resulting from the use of DSLs, especially for medium and large applications. We try to overcome the aforementioned shortcomings by providing Genesis, a C++ library which facilitates the ease of development and use, of textual DSLs.

With Genesis, all you have to do in order to create a new language is to specify its grammar and what should each of its rules generate when matched. Based on this grammar, Genesis generates and builds the parser that will be used to make the matchings and transformations. In this new language the user writes pieces of code, which he then uses as stand-alone or fragmentary (as seen in subchapter 1.2.1). Either way, after he has written some code, he again uses Genesis to parse these code snippets (with the built parser). The parser tries to match the rules of the grammar on this input and generates the appropriate output (specified when the grammar was built). The resulting text can then be used as is, or compiled with a specific GPL. This way, you can code your application in an existing GPL, and use Genesis to overcome the lack of abstraction in the general-purpose language. The solution for code generation is language independent, so we can generate any kind of output, from plain text to code targeting any kind of GPL.

Code generation is not always enough. Especially when using a DSL in a stand-alone form, you may want that code to be interpreted in a certain way, not necessarily generate something. Genesis lets you do that by allowing you to assign semantic actions to rules and subrules. When parsing, whenever such a component is matched, it will execute the semantic action first and then move on with the parsing. Thus, with the semantic actions functionality, the parser generated with Genesis acts like an interpreter for the defined language. Genesis is based on C++, so the semantic actions also have to be written in C++.

The motivation for choosing C++ is that it's extremely fast and it allows us to work both at a

high level and at a low level. These aspects are very important, making C++ the language of choice in developing cutting-edge software solutions. Still, C++ is a GPL and shares the aforementioned problems of GPLs, an insufficient level of abstraction and limited capabilities in expressing domain-specific concepts in a natural manner, and we may often feel the need to express solutions to some problems at a much higher level. Our vision is that we don't need to extend the C++ language to be able to do that, but, create custom constructions which express better the domain-specific concepts, mix them with C++ code the way we want, and then translate them into the host language. And this is exactly what Genesis does, as presented above. This way we can enhance C++ with different capabilities that aren't provided by the base language. More than that, we provide Genesis in the form of a library because other persons might want to use it in their development environments in a custom way (eg. as an IDE plugin) or build new IDEs for it.

There are many practical reasons why programmers would want to modify the language that they uses: to add constructs that better express their algorithms, to add constructs other languages have that they see as useful, and to reduce boilerplate code and repetitive patterns. All these features can be achieved with Genesis. Just think at large applications with hundreds of thousands or millions of lines of code (eg. enterprise applications or operating systems), and how much boilerplate code they contain. The repetitive patterns reduce the quality of the source code, and to eliminate them we use generative techniques (and among others, source code generation – which is possible with Genesis).

An argument for not using internal DSLs is the concern for proliferation of ad-hoc languages, designed by people with no clue, mixed in the same source code. But this does not stand, because if you change the "ad-hoc languages" with "libraries" you get the same effect. Designing a language is not very different than designing an API of a library. Good API designers will also design good languages, whereas bad designers will mess things up both ways. Also, some may argue that we should prefer libraries instead of language extensions, but why not have both? Why not exploit language extensions in our favor? Just like for an API, if you have it documented, it's easier to read and understand the rules of the grammar. The motivation for designing languages is that they provide syntax flexibility compared to libraries.

We said that DSLs are good for code noise reduction, and by doing this we enhance the quality of the source code. More than that, if we reuse various pieces of source code by copy pasting and not using generic techniques, we will most likely create unwanted and also not easy to spot defects. Thus, with DSLs, by hiding irrelevant technical details and using the technique of source code generation we also create a more secure code. The rules of the DSLs will be single points of failure and if the logic of the application fails when it gets to them, then the error will propagate to each of the code instances that match these rules, and will be easier to trace the bugs. So, the idea is to apply a generic solution for the same problem. Also it's easier to change the implementation of the defined constructs; this change will propagate throughout the code instances.

The last aspect that we need to talk about when considering using Genesis is the improvement of communication with domain experts and between roles in a development team. A DSL uses terms and concepts that map directly to the problem space, simplifying communication between business and technical project participants. With a tool like Genesis we empower persons who are not traditional programmers, with the ability to create their own software solutions. Programmers can create DSLs, which can then be used by domain experts. Thus, the latter, who have a strong understanding of a problem domain but no formal computer programming training, could submit useful code artifacts at any time, or write software applications to solve a specific need in their daily work tasks.

A DSL is intended to overcome the syntactic limitations of a particular language by extending it. And that's where the parser is needed – it parses and interprets the user-defined constructions. Genesis relies havily on the Boost.Spirit parser generator, so in the next subchapter we talk about it.

2.2 Spirit Parser Framework

In subchapter 1.2.1 we said that we will provide an example of a Domain-Specific Embedded Language (DSEL), so we talk here about the Spirit parser framework. This subchapter will be based on [W5] and [W6] which are the main Spirit references.

Spirit is part of Boost Libraries[W7], a peer-reviewed, open collaborative development effort. "Boost Spirit is an object-oriented, recursive-descent parser and output generation library for C++. It allows you to write grammars and format descriptions using a format similar to Extended Backus Naur Form (EBNF) directly in C++. These inline grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable. In retrospect, conventional compiler-compilers or parser-generators have to perform an additional translation step from the source EBNF code to C or C++ code." [W6].

Spirit exposes 3 different DSELs to the user, each of them contained in a separate module :

- For creating parser grammars (Parsing Expression Grammar (PEG)) – **Spirit.Qi**
- For the specification of the required tokens to be used for parsing (Regular expressions) – **Spirit.Lex**
- For the description of the required output formats (Reverse PEG) – **Spirit.Karma**

The input grammar and output formats, are written entirely in C++ so we don't need other tools to compile, preprocess or integrate those into the build process. Both the created parsers and generators are fully attributed, which allows us to easily build and handle hierarchical data structures in memory.

The three Boost.Spirit components can be used in a stand-alone form or together. Normally, a token sequence generated by Spirit.Lex is used as input for a parser generated by Spirit.Qi, and the hierarchical data structures generated by Spirit.Qi can be used for the output generators created using Spirit.Karma. Figure 4 shows the typical data flow of some input being converted to some internal representation. After some (optional) transformation these data are converted back into some different, external representation. The figure highlights Spirit's place in this data transformation flow.

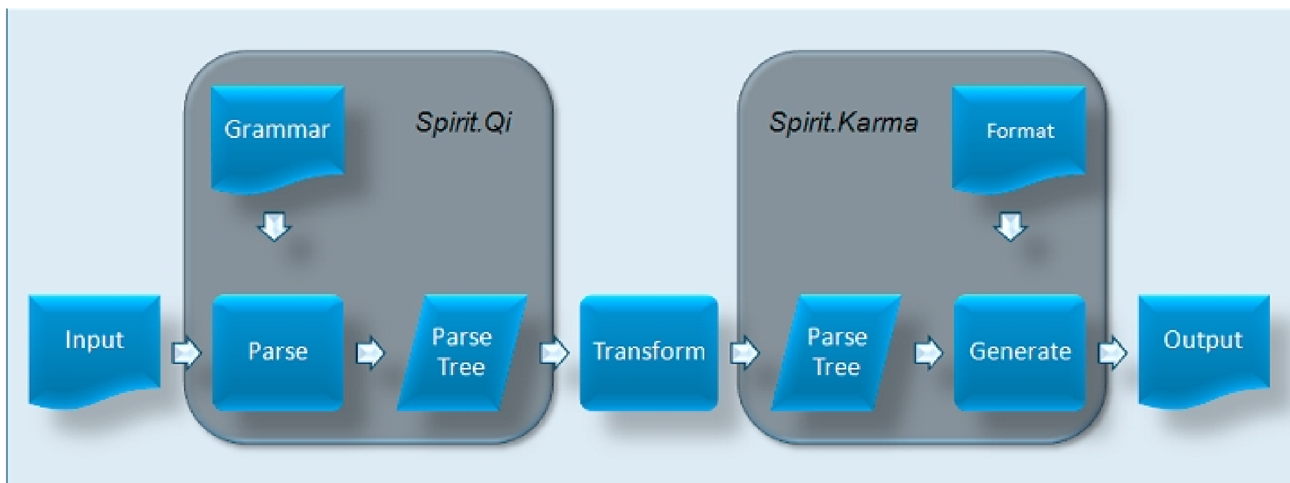

Figure 4. Spirit.Qi and Spirit.Karma In a Data Transformation Flow Of A Typical Application

Spirit.Qi is Spirit's sublibrary dealing with generating parsers based on a given target grammar (essentially a format description of the input data to read). Next, we provide a grammar snippet in EBNF to define a calculator :

```
group       ::= '(' expression ')'
factor      ::= integer | group
term        ::= factor (('*' factor) | ('/' factor))*
expression  ::= term (('+' term) | ('-' term))*
```

The same code can be expressed in Spirit.Qi as follows :

```
group       = '(' >> expression >> ')';
factor      = int_ | group;
term        = factor >> *(('*' >> factor) | ('/' >> factor));
expression  = term >> *(('+' >> term) | ('-' >> term));
```

As we can see, there aren't major differences between the two snippets: * and + operators must precede the operand, and >> operator is used instead of sequence operator in EBNF (which is denoted as single space character). The reason is that C++ operator overloading puts limits on how the operator is overloaded and what could be used as operator in the first place. int_ is a Spirit parser which matches an integer.

As we said earlier, Spirit permits creating PEG grammars in C++. "Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use prioritized choice. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing." [A50]

Like EBNF, PEG is a formal grammar for describing a formal language in terms of a set of rules used to recognize strings of this language. Unlike EBNF, PEGs have an exact interpretation. There is only one valid parse tree (Abstract Syntax Tree) for each PEG grammar. In PEG, in case of ambiguity where one or more branches can succeed, the first case always wins (the first branch which is successful). A crucial difference between PEG and general Context Free Grammars (CFGs) is that in PEG, loops behave greedily. If we try to match a sequence of 'A's and there is another 'A' just before the end-point, it will always fail because the preceding loop has already exhausted all 'A's and there is nothing more left. Table 2 presents PEG and (their equivalent) Spirit operators.

Description	PEG	Spirit Qi
Sequence	a b	a >> b
Alternative	a | b	a | b
Zero or more (Kleen)	a*	*a
One or more (Plus)	a+	+a
Optional	a?	-a
And-predicate	&a	&a
Not-predicate	!a	!a
Difference		a - b

<table>
<tr><td>Expectation</td><td></td><td>a > b</td></tr>
<tr><td>Semantic action</td><td></td><td>a[f]</td></tr>
<tr><td>List</td><td></td><td>a % b</td></tr>
<tr><td>Permutation</td><td></td><td>a ^ b</td></tr>
<tr><td>Sequential Or</td><td></td><td>a || b</td></tr>
<tr><td>Character set negation</td><td></td><td>~a</td></tr>
</table>

Table 2. PEG versus Spirit Qi Operators

Spirit is implemented as a DSEL using expression templates [A51] and template meta-programming [A52]. Template metaprogramming is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates include compile-time constants, data structures and complete functions. The use of templates can be thought of as compile-time execution. Expression templates is a template metaprogramming technique in which templates are used to represent part of an expression. Typically, the template itself represents a particular type of operation, while the parameters represent the operation to which the operation applies. The expression template can then be evaluated at a later time or passed to a function. The purpose of expression template is to inline expressions during compilation, which produces faster and more readable code. The Spirit libraries enable a target grammar to be written exclusively in C++. Inline grammar specifications can mix freely with other C++ code and because of the generative power of C++ templates, are immediately executable.

Expression templates and template meta-programming are powerful tools for designing highly generic libraries like Spirit, but they don't come for free. One of the problems is the compilation time. Because Spirit pushes the compiler to its limits, expect slow compilation times, especially on complex grammars. Another problem are the compilation errors. When some aspect of our grammar written in Spirit fails, we often get errors which run to multiple pages of impenetrable garbage. The complex templatised code used with the Spirit library causes the compiler to report the error, as each layer of templatised code is unwrapped, making it hard to spot the defect.

The Boost.Spirit library contains an impressive suite of classes to make the creation of object oriented parsers both quick and clean. It has the advantage of being very quick and easy to use without having to pre-process grammar files or auto-generate code using external tools. In order to implement the library, the programmers have used many cutting edge features of ANSI Standard C++, so the library has its problems, most notably impenetrable error messages and slow compilation times. Nevertheless, it is worth using it for its advantages, and mainly for the clean definition of parsers which makes it a definite candidate for writing parsers easily.

The current Genesis version uses only the Qi component, so whenever we talk about Boost.Spirit or just Spirit further on in the thesis, we actually refer to the Boost.Spirit.Qi component. Also, the Qi components can be found in `boost::spirit::qi` namespace; for simplicity we'll use `qi::` followed by the Qi component we refer to.

2.3 Design

In subchapter 2.1 we briefly introduced the working steps of Genesis. In this subchapter we present them in detail (but not too much details, because it's beyond the purpose of this thesis), accompanied by an example of usage. Figure 5 will be our main reference.

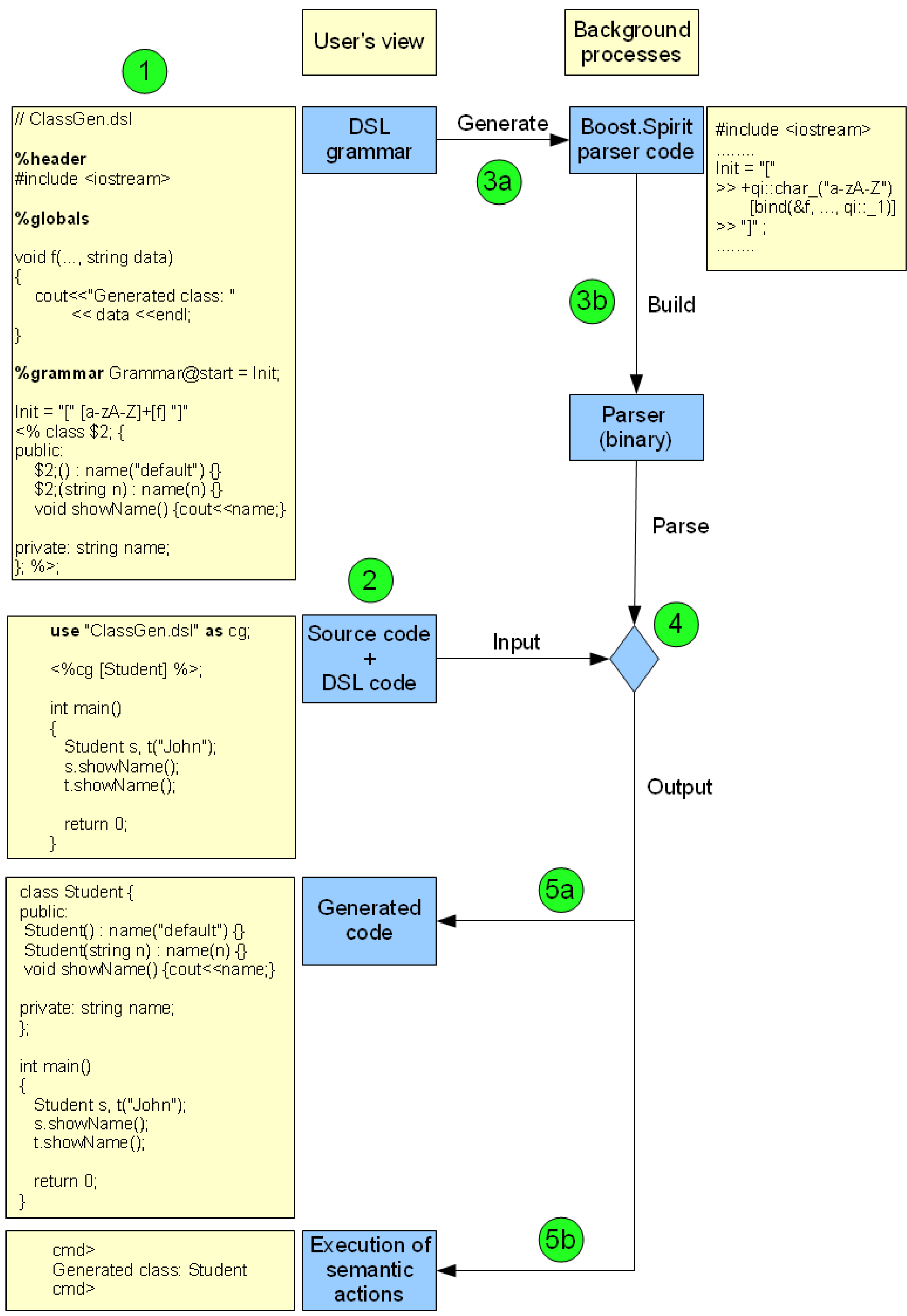

Figure 5. Genesis Working Steps

At the top of Figure 5 we have two labels, the user's view and background processes, to know what the user sees and what happens in the background. Below them, in blue are displayed the resources (used or generated), near them, in yellow are presented their structures, and the green circles show the order of the working steps. Between resources are arrows with labels on them, indicating the actions that are executed in order to transform one resource into another.

A user who wants to use Genesis must first create the grammar of his DSL (the green circle with value 1). The *ClassGen.dsl* file which implements a new grammar for particular class generation, comprises of a rule, *Init*, which matches a pair of square parantheses with an identifier between them. When the identifier is matched, a semantic action is executed (the function `f()`), and if the whole rule is matched, then this rule will generate a new class definition based on the identifier. The structure of a *.dsl* file will be analyzed in detail in the following chapter.

The second step is to create the source code of your application, and mix it with code of the newly created DSL. You don't need to use Genesis from the beginning of a project, you can use it at any step of your application's lifecycle. Just create the DSLs you need and use them in your source code as shown in the example, with: `use "DSL-file" as tag;`- this indicates that *tag* will use the DSL defined in *DSL-file*. Throughout the source code, we can use the DSL by attaching snippets of the following form : `<%tag dsl-instance %>;` . Tag indicates the DSL (from *DSL-file*) that *dsl-instance* should match.

The third step is separated into two. First, we use Genesis to generate code for a Boost.Spirit parser out of the DSL definition and then, this code is built with a C++ compiler (this version supports only gcc/g++) which creates a binary parser. Along with the generation of the parser, there is also generated a storage file (has the *.gs* extension). This is a serialized instance of `GeneratorStorageStatic` or `GeneratorStorageDynamic` classes. These classes contain the code that is to be generated when a rule is matched. The serialization is made using the Serialization library from Boost [W7].

In the forth step we use the binary parser to parse the source code mixed with DSL code. The DSL code is matched by the appropriate parser and Genesis constructions throughout the code are consumed. If using semantic actions, the parser is run twice. First it parses for correctness and then it does the actual matching, generates code and executes semantic actions.

The fifth step which results from parsing is actually the output, and it comes in two possible ways (the green circles marked with 5a and 5b). First, we can generate code by replacing the *dsl-instance* occurences with the matched rules code – remember that we said that when a DSL rule is matched, we consume the input tokens and replace them with a certain code that we defined for that particular rule (eg. Init rule in our example will generate a custom class definition). Aside code generation, we also have the semantic actions utility. In *ClassGen.dsl* we saw that after matching the identifier, we execute a semantic action whose behaviour is defined in function `f()`. In our example, when matching the DSL instance `<%cg [Student] %>;` , the function `f()` will be called, with *"Student"* as string data, and it will print on screen *"Generated class: Student"*. Obviously, there could've been a more complex behaviour, but this is just an example that shows how Genesis facilitates the creation and use of DSLs.

If we use Genesis without semantic actions, we can think of it as being a code generator. If we use only the semantic actions, then it acts like an interpreter for the provided code. Thus, we can think of Genesis that it facilitates the creation and use of internal DSLs when used as a code generator or external DSLs when used as an interpreter. Our initial vision was to exploit the power of C++, allowing it to intermingle with user defined higher level languages, which would translate into C++ code when processed. Genesis does more than that because its code generation functionality is language independent, meaning that we can generate any kind of text, independently of the base language. Only when we want to execute additional actions we use C++ in order to define the behaviour of semantic actions, which are nothing more than C++ functions being called.

Using a DSL this way can often make up for limitations in a host language, allowing us to

express things in a comfortable DSL and then generate code that's used by the actual execution environment. Obviously, DSLs don't solve and are not intended to solve all our problems, and we should know when it's appropriate to use them. So, when should we consider making a DSL? Usually, when an aspect of a system is rich in business rules or work-flow. A well-written DSL would allow customers to understand the rules by which the system works. This isn't going to lead to a "cacophony of languages" [B2] which will be hard to learn because, we already have a cacophony of libraries and frameworks that programmers have to learn. As said in Chapter 1, a DSL is nothing more than a facade over a library or framework and as a result, they contribute little complexity over what is already there. A good DSL should make things better by making these underlying libraries and frameworks easier to use. Also, with Genesis it's really easy to create DSLs because all the user has to do is to learn the (close to EBNF) syntax allowed and specify what should the rules (he defines) generate. The learning curve is fast and the advantages can be much bigger than the time spent on learning how to use this tool.

Chapter 3

Implementation

In the previous chapter we saw an introduction of the Genesis library, its design, and a motivation on why should someone use this tool. This chapter covers Genesis' implementation in detail.

3.1 Genesis API

Genesis comes in the form of a library (with the motivation presented in the previous chapter), so we must first talk about the functions it exposes – its API. We will present each of the files that can be included into an application that wants to use Genesis, and what they expose, briefly. The functionality exposed by the library can be used by accessing the namespace `genesis` (eg. the current Genesis version can be accessed by `genesis::GENESIS_VERSION`) – for simplicity we won't mention it further on.

GenesisBerry.hpp

This is the main file we need to include, to be able to work with Genesis. It provides the `GenesisBerry` class which exposes the following methods :

- `InitProjectConfig` – This is the first function that must be called when wanting to work with Genesis. It takes a configuration file, parses it and updates the global `ProjectConfig` structure (see ProjectConfig.hpp below). These configurations are used later on to generate the parser, build it, and generate code when processing the source file
- `GenerateDslParser` – This function first parses the specified DSL file, generates the source code for a Boost.Spirit parser and builds it, resulting in a binary parser
- `PreprocessStringSource` and `PreprocessFileSource` – Parses the source code for occurences of the following form : `use "DSL-file" as tag;`
(see PreprocessReturnStruct.hpp below)
- `GenerateCodeFromFileToFile`, `GenerateCodeFromFileToString`, `GenerateCodeFromStringToFile` and `GenerateCodeFromStringToString` – These functions use the built parser to process the source code and generate appropriate code (as specified in the definition of the DSL) and/or execute the appropriate semantic actions. The functions differ only in the parameters they take just like their names say – the source and the output can be either a string or a file

Each of these functions returns an int – 0 for no errors, 1 for errors.

ProjectConfig.hpp
Exposes the `ProjectConfig` Structure. In order to work, Genesis relies on a configuration file (*config.gsis*) which keeps all the configurations for a project: working directory, compiler, compiler path, compiler build options, aditional include directories, a map for dsl-to-binary bindings and paths to some extra header files. The map of bindings is needed in order to link DSL files to their corresponding binary parsers. The extra header files are used when generating the source code for the parser. Genesis uses all this options in the same way an IDE does, except that in the IDE we can select all these options from a graphical user interface, whereas here we have to edit the configuration file in a text editor.

PreprocessReturnStruct.hpp
Exposes the `PreprocessReturnStruct` structure. This keeps two fields: `code` and `used_tags`. `Code` is actually the remaining source code after the source file was preprocessed for occurences of this kind: `use "DSL-file" as tag;` . When found, we insert a new key-value pair into the `used_tags` map with `DSL-file` as key and `tag` as value. This line actually links the tag to the the DSL found in the `DSL-file`.

LastError.hpp
Exposes `gsisLastError` which keeps the last error (if any). This is actually an uError structure defined in ulib (an utility library Genesis depends on). This structure has 3 fields: the actual error message (a string), the error code (a long) and a vector of boost::any (can store any type). The last field is useful for two main reasons – we can store warnings or less important error messages or solutions in order to solve the current error, OR before we return from the place where the error happened, we can store into this vector the state of the program in that point (local variables, etc) which will help the user remedy the error faster (without this vector, the programmer must define its own global data structures to keep additional information about the error and program state, which makes programming more difficult).

version.hpp
Exposes `GENESIS_VERSION`, the current version number of Genesis.

genesis.hpp
Include this file into your source code if you want to include all of the above headers in one step.

In the same package we can also find GenesisCLI – a command line interface application which uses the exposed Genesis library functionality. For now, this is enough to demonstrate how Genesis works but, in the future, the library should be integrated in an IDE.

3.2 DSL Structure

In Listing 1 we can see the structure of a default DSL file. It comprises of 3 main sections:

Header
This part is used if we want to include some C++ code into the same file as the parser source code. Anything you type after `%header` and until `%globals` will be copied verbatim into that source file, so no processing or error checking is made (this is made by the compiler when building the parser). This section is optional.

Globals

Anything that is typed between `%globals` and `%grammar` will be copied verbatim into a .cpp file, and this source will be included (with `#include`) in the main parser source code. Just like with the header, no checks are made here (the compiler does them when building), and this section is also optional.

Grammar

This is actually were we define the grammars of our DSLs. This part starts with the keyword `%grammar` and requires a grammar name and the name of the start rule. These are needed for when we generate the Spirit parser – Spirit defines its grammars in classes (so we need a name for them) and also needs the grammar start rule, to know where to start. Also between the two we must put the '@' symbol, and the line must end with a semicolon. Below, we can start defining grammar rules.

A rule looks like this: it starts with a rule name, then an '=' must follow, then the actual definition of the rule (other rules or subrules – we'll see the allowed syntax in subchapter 3.3) , and then specify what to generate. This is done between '<%' and '%>;' and we write the text to be generated when the rule is matched.

```
%header

//C++ code

%globals

//C++ code

%grammar GrammarName@start_rule;

//rule definitions
rule_1 = r_11 r_12 ... r_1i <% string to generate when rule is matched %>;
....
rule_m = r_m1 r_m2 ... r_mj <% string to generate when rule is matched %>;
```

Listing 1. DSL File Structure

After `%grammar`, we can put C or C++ comments anywhere in text, they will be completely ignored. The only place where they are not allowed is between '<%' and '%>;' because everything that's in between is copied as is. Also there are some other things you need to be careful about:

- `%header` takes the text that follows in a verbatim way and stops when it finds `%globals` (so everything you put there will be copied as is). Same applies to `%globals` which will take everything until it encounters `%grammar`
- The start rule must not be defined as a rule separately (if you want to give it a custom definition do it there – we'll see how, later on)
- There should be at least one rule defined
- Rules must not be redefined
- There should be at least one space between the last subrule and '<%'
- What's between '<%' and '%>;' will be generated as is (spaces, newlines, etc, will be kept unchanged)
- Identifiers (grammar name, start rule name, name of rules) must start with a letter or with an underscore and can continue with letters, underscores or digits. Identifiers are case sensitive!

3.3 Allowed Syntax

We've seen in the previous subchapter what's the structure of a DSL file, but haven't talked about how to write rules. This is what we do in this subchapter – present the allowed syntax for rules.

Genesis generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to EBNF and regular expressions. We saw in Listing 1 that when we want to define a rule, we write its name, put the equal sign and start defining what it matches – its *subrules* or *components*. Basically, the user defines a grammar in Genesis syntax, but in the background this is translated into an equivalent Spirit grammar. The intention of this chapter is not to show how Spirit works, but to present how Genesis works, and what its subrules translate into – the equivalent Spirit parser components (we can find references to them in [W6] which is also our main reference).

3.3.1 Literals

The literals we try to match must be put either between `' '` or `" "`. This matches a single character or a string of characters, and in Spirit it translates to `qi::lit('')` or `qi::lit("")`. For example `"test"` is a literal parser which matches the string *test*. `''` and `""` (empty literals) don't make sense and are not supported by Genesis – this would mean "match an empty string", but this is always true (we have an infinite number of empty strings in the text we process) and the result is that the parser will never stop.

A special case with literals is when we want to match ' or " as literals. To match ' we must put it between "" and to match " we must put it between ' '. For example, if we want to match the string *this " is a ' test*, we can write a rule like this :

```
rule = "this " '"' " is a " "'" "test"
```

3.3.2 Bracket Expressions

`[ ]` – is a bracket expression, and it matches a single character that is contained within the brackets. For example `[abc]` matches "a", "b" or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y" or "z", as does `[a-cx-z]`. A bracket expression of this kind translates into `qi::char_("")`, which takes as a parameter a string – the pattern we try to match (eg. `[a-z]` translates into `qi::char_("a-z")`). Other bracket examples are:

- `[a-zA-Z]` – alphabetic characters
- `[0-9a-fA-F]` – hexadecimal characters
- `[actgACTG]` – DNA identifiers
- `[\x7f\x7e]` – Hexadecimal 0x7F and 0x7E

`[^ ]` – Matches a single character that is not contained within the brackets. For example `[^abc]` matches any character other than "a", "b" or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z". As above, literal characters and ranges can be mixed. An expression of this kind translates into `(qi::char_ - qi::char_("pattern"))`, which says "match any character but not a character which matches the specified *pattern* (in the second char_)".

At some point we may also want to be able to match "]", "-" or "^" as literals. Genesis handles these special cases as follows:

- `[^]` – Matches the "^" character because you can't have an empty bracket expression
    - `[^^]` – Matches any character but not "^"
    - `[ab^]` – Matches "a", "b" or "^"
- "]" is the ending bracket and Spirit is a greedy parser, so it's hard to parse this in the middle of a bracket expression. Thus, if we want to match a pattern and also "]" then we must put "]" as the last character:
    - `[]]` – Matches only "]"
    - `[^]]` – Matches any character but not "]"
    - `[abc]]`, `[^a-z]]`
- `[]^]` – Matches both "]" and "^"
- The "-" character is treated as a literal character if it is the last or the first (after the "^") character within the brackets: `[^-abc]`, `[-abc]`, `[abc-]`. Note that backslash escapes are not allowed
    - `[-a-z]]` – Matches "-", a lowercase letter, or "]"

It's important to write our rules with care, because we may omit some characters and get a different result than expected, without knowing why that happened. For example `[ab [cd]` will match "a", "b", space, "[", "c" or "d" and not (("a" or "b") AND ("c" or "d")) – this is because we forgot to put a closing bracket after "ab". To get the second behaviour we need to write the expression like this : `[ab] [cd]`.

To match any single character, Genesis provides two parsers : $char and "." (a single dot). They are semanticly equivalent (they both translate to "qi::char_"), but we have them both in order to retain the "$ syntax" of Genesis parsers (see below) and also regular expressions ("." from regex has the same meaning as in Genesis). In the end, it's the user's choice which form he uses.

3.3.3 Parsers

Character Parsers

Bracket expressions are also a type of character parser, but with a different syntax than those presented here. Spirit comes with a lot of single character parsers, for character classification. In Table 3 we can see the parsers provided by Genesis, their equivalent parsers in Spirit and their semantics. These parsers have an associated Character Encoding Namespace (in Table 3 represented by `ns::` which can be ascii, iso8859_1, standard or standard_wide). Also, in the table we can see an *"Equivalent to"* field. For example, $alnum (and Spirit's alnum) matches a character based on the equivalent of `std::isalnum` in the current character set, so we put `std::isalnum` in that column.

Genesis	Spirit	Semantics	Equivalent to
$alnum	ns::alnum	Matches alpha-numeric characters	std::isalnum
$alpha	ns::alpha	Matches alphabetic characters	std::isalpha
$blank	ns::blank	Matches spaces or tabs	std::isblank
$cntrl	ns::cntrl	Matches control characters	std::iscntrl

$digit	ns::digit	Matches numeric digits	std::isdigit
$graph	ns::graph	Matches non-space printing characters	std::isgraph
$lower	ns::lower	Matches lower case letters	std::islower
$print	ns::print	Matches printable characters	std::isprint
$punct	ns::punct	Matches punctuation symbols	std::ispunct
$space	ns::space	Matches spaces, tabs, returns, and newlines	std::isspace
$upper	ns::upper	Matches upper case letters	std::isupper
$xdigit	ns::xdigit	Matches hexadecimal digits	std::isxdigit

Table 3. Character Parsers

Numeric Parsers

Similar to the character parsers, Spirit (and Genesis) provides a number of numeric parsers. We can see the available ones in Table 4 along with their attributes (exposed types) and semantics.

Genesis	Spirit	Attribute	Semantics
$float	float_	float	Parse a floating point number into a float
$double	double_	double	Parse a floating point number into a double
$long_double	long_double	long double	Parse a floating point number into a long double
$bin	bin	unsigned	Parse a binary integer into an unsigned
$oct	oct	unsigned	Parse an octal integer into an unsigned
$hex	hex	unsigned	Parse a hexadecimal integer into an unsigned
$ushort	ushort_	unsigned short	Parse an unsigned short integer
$ulong	ulong_	unsigned long	Parse an unsigned long integer
$uint	uint_	unsigned int	Parse an unsigned int
$ulong_long	ulong_long	unsigned long long	Parse an unsigned long long
$short	short_	short	Parse a short integer
$long	long_	long	Parse a long integer
$int	int_	int	Parse an int
$long_long	long_long	long long	Parse a long long

Table 4. Numeric Parsers

Boolean Parsers

Similar to the above, we have boolean parsers which can be seen in Table 5.

Genesis	Spirit	Semantics
$bool	bool_	Matches a boolean value
$true	true_	Matches "true"
$false	false_	Matches "false"

Table 5. Boolean Parsers

Auxiliary Parsers

Similar to the above, we have auxiliary parsers which can be seen in Table 6.

Genesis	Spirit	Semantics
$eol	eol	Matches the end of line (\r or \n or \r\n)
$eoi	eoi	Matches the end of input (first == last)
$eps	eps	Matches an empty string

Table 6. Auxiliary Parsers

3.3.4 Operators

Grouping ()

This is not an operator but instead, parentheses are used to define the scope and precedence of the operators, which are also used for grouping different components. We'll provide examples which use them, later on.

Sequence (a b)

In Genesis this is not an operator because it's just a space that delimits two components, but this space translates into Spirit's sequence operator ">>". The sequence operator, `a >> b` (just `a b` in Genesis), parses a and b in sequence. It's also important to mention that between two components, we can have any number of spaces or C/C++ comments.

Expectation (a : b)

Like the sequence, the expectation operator `a : b` parses a and b in sequence, but *exactly* a followed by b. b is expected to match when a matches, otherwise, an exception is thrown. You can think of the expectation operator as a deterministic point in the grammar. Those are the places where backtracking cannot occur. In Spirit, `a : b` translates into `a > b`.

Expectation is important when you have an exact pattern to match (can't be different). For example a credit card number is composed of 4 groups of 4 digits each, with an "-" between them, so you can use this operator because you expect to match this pattern precisely.

Alternative (a | b)

This operator matches one of two or more operands. For example `a | b | c` matches a or b or c. The operands are tried one by one from left to right, and the first successful match short-circuits the search. This short-circuiting implicitly gives the highest priority to the leftmost alternative. More

than that, short-circuiting improves the execution time, so if the order of the alternatives is irrelevant, try to put the most common (most expected) choice first, for maximum efficiency.

Parantheses are useful in the context of the alternative operator and should be used to specify the desired intent. For example `a | b c` can be either `(a | b) c` OR `a | (b c)`, so you should specify your intent with care.

Difference (a – b)

The difference operator, `a - b`, is a binary operator that matches the first (Left Hand Side) operand but not the second (Right Hand Side). The expression will always fail if the RHS is a successful match, even if the RHS matches less characters. For example, the rule `"policeman" - "police"` will always fail to match.

Again, for readability and expressing the correct intent, the difference expression should be put between parantheses. To try to match (once) any character between ' and ' , we would write `rule = "'" ($char - "'") "'"` . This, first matches an ', then any character but not ', and then again ' - we do it like this because Spirit is a greedy parser and can't stop automatically just before the last '; if it can consume ' , then it consumes it regardless of what the next parser is. This small example is equivalent to `rule = "'" [^'] "'"`. The difference operator will become more important after we introduce quantifiers.

Kleene (a*)

The Kleene operator (the asterisk), `a*`, is a unary operator that matches its operand *zero or more* times. In Spirit `a*` translates into `*a`. A rule that matches C comments looks like this:

```
rule = "/*" ($char - "*/")* "*/"
```

The second component means that we want to match zero or more times, any character ($char) but not "*/". Again, Spirit is greedy and if we say *any character any number of times* (`$char*`) then it consumes all the input and won't stop regardless of what the next parser is (the literal "*/").

Another example would be to try to match a list of integers separated by commas, for example *"11, 22, 33, 44, 55"*: `rule = $int ("," $int)*`

Plus (a+)

The plus operator (plus sign), `a+`, is a unary operator that matches its operand *one or more* times. In Spirit, `a+` translates into `+a`. If we want to parse one or more strings (that are separated by spaces) containing one or more alphabetic characters, we can write a rule like this:
`rule = ( $alpha+ $space+ )+` - this matches alphabetic characters (must be at least one), followed by at least one space, and all this at least one time.

Optional (a?)

The optional operator (the question mark), `a?`, is a unary operator that matches its operand *zero or one* time. In Spirit, `a?` translates into `-a`. For example, if we want to match the name of a person which is between quotation marks, and optionally its age, we can write a rule like this:

```
rule = '"' ($char - '"')+ '"' (',' $int)?
```

This rule matches a ", then at least one character of any kind but not a quotation mark, then ", and then optionally a comma and an integer (the age of the person).

And-Predicate (&a)

Syntactic predicates assert a certain conditional syntax to be satisfied before evaluating another production, and don't consume any input. The and-predicate (lookahead operator), `&a`, is a positive syntactic predicate that returns a zero length match only if its predicate matches.

The & operator passes only if its predicate rule passes. `&"a"` will only continue if the next character to be parsed is 'a'. `rule = "end" &";" ";"` matches the input string *"end;"* as follows : first matches the string "end", then peeks at the next character (";") without consuming it (if the character is ";" then move on to the next subrule to match, else the parser fails), and then matches ";".

Not-Predicate (!a)

The not-predicate, `!a`, is a negative syntactic predicate that returns a zero length match only if its predicate fails to match. The operator passes only if its predicate rule fails: `rule = !"a"` - checks to see if the next letter is "a" and if it is, it fails, otherwise it passes. In any case, the current iterator position is not moved. To parse a list of characters terminated by a ";" (eg. "abcdef;") we can write: `rule = (!";" [a-z])* ";"`

Note that the order of operators is important : the most important one (the one that applies first) is the not operator (!), followed by quantifiers (*, +, ?) and the lookahead operator (&). Thus :

- `![a-z]*` = `(![a-z])*` - put parantheses to get: `!([a-z]*)`
- `&[a-z]+` = `&([a-z]+)` - put parantheses to get: `(&[a-z])+`
- `&![a-z]` = `&(![a-z])`
  - Put parantheses and change the position of operators to get: `!(&[a-z])`
- `&![a-z]?` = `&((![a-z])?)`

List (a % b)

The list operator, `a % b`, is a binary operator that matches a list of one or more repetitions of a separated by occurrences of b. This is equivalent to `a (b a)*` (In Spirit `a >> *(b >> a)`).

We review the (Kleene operator) example which matched *"11, 22, 33, 44, 55"*. To parse this easily we can write : `rule = $int % ","` which is equivalent to `rule = $int ("," $int)*`

Permutation (a ^ b)

The permutation operator, `a ^ b`, matches a or b in any order. The operands are the elements in the permutation set. Each element in the permutation set may occur at most once, but not all elements of the given set need to be present. Note that by this definition, the permutation operator is not limited to strict permutations. For example, `rule = "a" ^ "b" ^ "c"` matches *"a", "ab", "abc", "cba", "bca"* ... etc.

If we want more than one occurences of a component we can use the Kleene operator. To parse a string containing DNA codes (ACTG) (eg. to match *"ACTGGCTAGACT"*), we can write `rule = ("A" ^ "C" ^ "T" ^ "G")*`

Sequential Or (a || b)

The sequential-or operator, `a || b`, matches a or b or a followed by b. That is, if both a and b match, it must be in sequence; this is equivalent to `a (-b | b)` (In Spirit: `a >> -b | b`). For instance, `rule = ($int || ('.' $int))` matches :

- "123.456" – the whole number
- "123" – the first part
- ".456" – the second part (just the fraction)

Note that it is advisable to put parantheses around the binary operators to express your exact intent and get the expected result. In the generated Spirit parser, the operands are separate components and using parantheses groups them into a single component. Because of that, using operands and binary operators without parantheses may create unexpected or unwanted results.

More elaborate examples will be shown in Chapter 4, which will use many of the parsers and operators shown in these subchapters.

3.4 Grammar Options

3.4.1 Grammar Definition

Before talking about this, we need to mention something that was omited in the previous subchapters: we can use defined rules as parsers (subrules) in the current rule. For example: `rule1 = "hello"` matches the literal "hello" and `rule2 = rule1 "world"` matches what rule1 matches (the literal "hello") and the literal "world". Also, we can use all the above operators on subrules, just as on any other component.

In subchapter 3.2 (and in Listing 1), we saw that to define the grammar of a DSL, you must write something like this: `%grammar GrammarName@start_rule;`. This is the default pattern: type `%grammar`, (at least one) space, grammar's name, the @ symbol, the name of the start rule and end with a semicolon. If in a grammar like this, we have for example m rules (rule_1, rule_2, ..., rule_m) then, the start rule will function as if it was defined as:

```
start_rule = (rule_1 | rule_2 | ... | rule_m)*
```

This is the default behaviour, to match each rule any number of times. We can override this behaviour by defining the start rule the way we want. We do it like this:

```
%grammar GrammarName@start_rule = pattern;
```

where pattern can be any valid expression (which uses the parsers and operators (and subrules) shown in the previous subchapters). In this new way of defining the start rule, we can also use rules defined below (so we can view the start rule as being top level and having access to all other rules of the grammar). Note that even if you don't use already defined rules in the definition, you must have at least one rule defined, even if it does nothing. For example

```
%grammar GrammarName@start_rule = ($int || ('.' $int));
```

parses numbers with optional fractional digits, and does not use any other rule, but we must have a defined rule below, even if it's only `rule = $eps` which does nothing.

For when we use the default behaviour, we have a functionality that allows us to not include certain rules in the default definition (as shown above) of the start rule. This can be used by putting an exclamation mark '!' before the name of the rule we want to define: `!rule = <definition>`. If Genesis finds this flag, then it won't put this rule into the definition of the start rule. For example:

```
%grammar GrammarName@start_rule;
rule1 = <definition>
!rule2 = <definition>
```

The start rule will be defined as `start_rule = (rule1)*` , and not `start_rule = (rule1 | rule2)*`. When defining our custom start rule, these '!' flags are ignored; the exclamation mark influences only the default behaviour of the start rule. But, with all these, if we define our custom start rule, we still need to have another rule defined. In the previous example, if both rule1 and rule2 would have had exclamation marks, then, Genesis would have failed to parse because there was hasn't been any rule defined.

Next, we present a grammar we ran across when doing tests, which produces erroneous results for a specific input:

```
%grammar TestGrammar@start_rule; //start_rule = (rule1 | rule2)*
rule1 = "save" $space ([a-z]* $space)*
rule2 = ("save" | "load") $space ([A-Z]* $space)*
```

We give it the following string as input: *"save this to a load THIS FROM B "* , and if not paying attention, we would expect it to match *"save this to a"* and *"load THIS FROM B "* but this is not the case. Instead (again, Spirit is greedy), what it does is to match "save", space, and all the strings formed using lower case letters followed by a space. Thus, it actually matches *"save this to a load "* leaving only *"THIS FROM B "* as the remaining input, which can't be matched by any other rule, and the parser fails.

3.4.2 Skipper

A large part of this subchapter is based on the "Parsing Skippers and Skipping Parsers" tutorial from [W5]. Spirit supports skipper based parsing and so does Genesis. When parsing formatted data stream we may want to ignore some parts of the input, for example whitespaces and comments when parsing computer languages. We can define a rule which matches those, and put it whenever we can encounter them in input, but this is a tedious and error-prone task because the parts we try to skip can appear at any point in the input. Let's assume that we want to parse a simple key/value expression: key=value, and allow for any number of space characters before, in between and after the key or the value. A grammar that would match the key/value pair without whitespace skipping would look like this :

```
pair  = key '=' value
key   = [a-zA-Z_] [a-zA-Z_0-9]*
value = [a-zA-Z_0-9]+
```

If we want to modify rule pair to match the whitespace characters, we get:

```
pair = $space* key $space* "=" $space* value $space*
```

which, while produces the desired result, is error prone, difficult to write, understand and maintain. Genesis (based on Spirit's functionality) overcomes this by allowing the user to specify the desired skipper parser. The default syntax is :

```
#skipper SkipperName@start_rule = pattern #end
```

This definition starts with the keyword `#skipper`, then it requires at least a space, skipper's name, the @ symbol, the name of the start rule, the equal sign, a pattern, at least a space and the keyword `#end`. This definition has to be put under the definition of the grammar. The pattern definition is

similar to when defining a grammar, and the current Genesis version supports only parsers and operators, and not subrules. These definitions (`%grammar` and `#skipper`) produce in the background two different grammars, that is two different classes, and thus, can't "communicate" one with the other, so we can't use rules defined below (for the grammar) in the definition of the skipper. For the record, the grammar class actually uses the skipper class in its definition. Also, the skipper name is used to name the skipper class, and the name of the start rule is used to name the first rule used by the skipper. Pattern definition doesn't end with a semicolon, but with the `#end` keyword.

If we want to use the skipper parser in Genesis, all we have to do is define it (as seen above), and the skipper will be automatically enabled. This means that it will skip the mentioned parts from input. For example, if we define a skipper with `start_rule = $space` then, if at input we have the string *"this is a test"*, this will be seen as *"thisisatest"* (the whitespaces were skipped) by the parser. Same applies to our pair rule above. When actually parsing, we don't need the spaces before, in between, and after the key or the value, because they don't have any value, we just want to allow them to be written by the user and nothing more. For that, we can define a whitespace skipper, use the rule as above (`pair = key '=' value`), and provide an input which can contain whitespaces.

The following example defines a skipper which ignores whitespaces, C and C++ comments:

```
#skipper
     skipper@start = $space
     |    ("/*" ($char - "*/")* "*/")     // C comments
     |    ("//" ($char - $eol)* $eol)     // C++ comments
#end
```

Directive lexeme[]

A problem with the skipper parser is that it applies to all rules (also note that the Spirit code has the skipper parser attached to each of the rules of the grammar). Thus, if we want to turn off skipping for a small part of the grammar, we need to use our rules in combination with the lexeme[] directive. This directive inhibits skipping during the execution of the embedded parser. For example, in a larger grammar, if we have a whitespace skipper enabled and want to parse some quoted text without skipping the whitespaces between quotes, we can write the following rule:

```
Text = '"' lexeme[($char - '"')*] '"'
```

(you can also put the quotes in lexeme[]). Here, the lexeme directive disables skipping while matching the string, which avoids "loosing" characters otherwise matched by the skipper. Please note that lexeme[] performs a pre-skip step (Spirit supports the no_skip[] directive which does not do pre-skipping in any case, but this directive is not currently supported by Genesis).

Lexeme can use between its brackets any expression based on the parsers and operators shown above, but can't use rules! This is because when the skipper is defined, automatically in the Spirit code each rule has the skipper attached; lexeme inhibits skipping, so calling a rule which (has the skipper attached and) requires skipping to work, results into a compilation error. Note that a rule without skipper (when we don't define one globally) acts like an implicit lexeme.

You need to be careful at how you write expressios with lexeme[]. For example, `rule = lexeme[[a-z]]` is wrong and results into a compilation error. This is because in the translation process, Genesis parses lexeme[, and then (the greedy Spirit parser which tries to match as much as possible) [a-z]] which is a valid bracket expression, leaving lexeme[without a closing bracket. Thus, we should rewrite the rule into `rule = lexeme[[a-z] ]` (note the whitespace).

One last thing you need to know about lexeme is that, Genesis does not allow you to put operators around it, because lexeme is just a directive, it inhibits skipping and expressions like

`&lexeme[]` or `lexeme[]*` don't make sense. Instead, you can put those inside lexeme's brackets and form any kind of expression you want.

3.4.3 Modes of Operation

In subchapter 3.2 (and Listing 1) we said that after we define what a rule matches, between '<%' and '%>;' we specify what it should generate. These delimiting sequences are mandatory, and can't be omited. As you may have observed, we didn't put them in the previous subchapters, and this is because we wanted to show the allowed syntax for defining rules. Now we introduce them, and also the two modes in which they can operate: static and dynamic.

Static

Before explaining how this mode works, we provide an example :

```
%grammar StaticGrammar@start;
#static
rule1 = "hello" $space [a-z]+ <%Matched rule1%>;
```

Note that we said that the generation mechanism is language independent, so Genesis just replaces the matched strings from input, with the text to generate specified for each rule. In our case, if the input is *"hello world"*, then this will be consumed and replaced with *"Matched rule1"*.

To activate this mode just put the keyword `#static` after the `%grammar` definition, and before defining any rule. This mode is useful when we just want to do plain generation (find and replace) – replace pieces of the input string with the text to generate from the rules that match. This mode is faster than the dynamic mode and also compiles the parser faster.

Dynamic

The static mode is limited to only generating a specified text, but we can't insert in this text the components the rule just matched. This is where the dynamic mode comes into play. It allows you to use the values of the matched components as strings, between the code generation delimiters. You can do that with the following syntax: `$component_number;` (for example: $1; , $2; , etc). $1; refers to the first matched component,$2; to the second, and so on. We modify our example into:

```
%grammar DynamicGrammar@start;
rule1 = "hello" $space [a-z]+ <%Matched [$1;][$2;][$3;]%>;
```

For the input *"hello world"*, the output is *"Matched [hello][ ][world]"*. We can see easily that it generated the matched components. The numbering starts from 1, and can't be less than that (eg. 0, -1, etc), or greater than the number of components the rule has (eg. In our case $4; , $5; etc). You may have also noted that after $ and the number of the component we put a semicolon. This is done because there may appear ambiguities without it – if for example in the string to generate we put $11, and our intention is to generate the first matched component and put a 1 after it, but the (greedy Spirit) parser which matches an integer will take as much as possible, in our case 11, but there isn't an 11th component and the parser will fail. That's why we put a delimiter after the number – could have been space, but for an exact generation we delimit it like this.

Note:

- This mode is the default one so, we don't have to put any option flag to enable it
- What's between '<%' and '%>;' is a string (C++'s `std::string`) – all matched components

must be converted into strings (in the background), and that's why this mode in a bit slower than the static mode

- Any expression (no matter how complex) between parantheses is counted as a single component (and implicitly has a single index)
- & and ! operators have an index but their values are "" (blank), because they don't consume any input
- Literals and spaces in Spirit don't have an attribute (don't return a value, they just match something), but in Genesis they do, so you can use the index on a literal or space component to get their values

3.5 Semantic Actions

Only the code generator component isn't enough; we also want to be able to access and manipulate the parsed values. We can do this with semantic actions. We can attach semantic actions to any components of a rule, or the rule itself, by putting between brackets the name of the function we want to execute. For example:

```
rule = [a-z][f] [A-Z]*[g] <%Generated text%>[h];
```

has 3 semantic actions, each calling a certain function : f, g and h. Note that the semantic action for the second component is put after specifying quantifiers (that's how it must be done for all components when using quantifiers). f, g and h are actually C++ functions that have a certain pattern for their parameter list:

```
void function(error_handler, pass, val, data)
```

- `pass` must be a `bool&` - if in the function it gets the value false, then the parser will fail, else (unused or set to true) it doesn't fail, and moves on with parsing
- `error_handler` must be a `string&` - when we put false to `pass,` we can assign `error_handler` a string error message that will be shown on screen
- `val` can be either `string&` or `vector<string>&` - it refers to the already accumulated value from parsing (we'll see this later on)
- `data` - stores the parsed value
- All these values can be without &, but we must put it if we want to change the base value

As we've seen in the previous subchapter, static and dynamic modes are different in behaviour. The dynamic mode converts everything into strings (in the background), because it needs them to be of this type, to be used at generation. The static mode doesn't do any convertion and the parsed values have their original types (eg. in static mode $int returns an integer type, whereas in dynamic mode it returns an integer but it's converted to string). For this reason, `data` and `val` can have different values based on the mode of operation used.

For the static mode, `val` is always a string. We'll provide a list of functions with different parameter lists, and then link the available components with the functions they can use:

```
void S1(string& err, bool& pass, string& val)
void S2(string& err, bool& pass, string& val, string& data)
void S3(string& err, bool& pass, string& val,<depends on the parser used>& data)
```

- **Literals**: as we said before, in Genesis they return a string so we can have a rule like rule = "test"[S2] which takes the literal as a string
- **Bracket expressions**
  - rule = [a-z][S3] – S3 with char& data
  - rule = [a-z]+[S3] – S3, also with char& data ! It calls the function for each character encountered. To force it call only once, use parantheses:
  - rule = ([a-z]+)[S3] – S3 with vector<char>& data calls only once, with the vector of chars
- **Group component**: rule = (any expression)[S3] OR (any expression)[S1!]
  - In the first form we write any expression between parantheses, but the problem is that we need to deduce the type for data ourselves – eg. ($int) will be int& data but ($int+) will be vector<int>& data , and so on
  - In the second form, observe the exclamation mark – this only applies to group components and is used when the expression between parantheses is more complex and you don't know what type data has, or don't want to use the data anyway, just use the semantic action to call the function – its semantics (for our example) is "S1 only!", meaning "function only!", without bothering about the type of data
  - The group component can be used to enforce the change of type for the data parameter, as we saw in the previous bracket expression
- **Parsers**
  - They use S3 with different types for data depending on the parser used (int for $int, double for $double, etc)
  - Auxiliary parsers use S1 because end-of-line ($eol), end-of-input ($eoi) and epsilon ($eps) don't return data which can be used
- **& operator:** rule = &component[S1] uses S1 because it doesn't return data which can be used (as it does not consume any input)
- **lexeme[]:** rule = lexeme[<expression>][S1] uses S1 because it doesn't return data which can be used (it's just a directive to inhibit skipping)
- **Subrules:**

  ```
  rule1 = <definition> <%String to generate%>;
  rule2 = rule1[S2]
  ```

  data will be of type string and store the generated string by rule1! In our case, string& data will contain *"String to generate"*
- **End of the rule**: rule = <definition> <%String to generate%>[S1]; - this is without data because there is no parsed value, just the generated string. An important aspect is that we can access val, which means we can access the string this rule generates.

For the dynamic mode, val can be string or vector<string>. We'll again provide a list of functions with different parameter lists, and then link the available components with the functions they can use :

```
void D1(string& err, bool& pass, string& val, vector<string>& data)
void D2(string& err, bool& pass, vector<string>& val)
void D3(string& err, bool& pass, vector<string>& val, string& data)
```

- **Literals, bracket expressions, parsers, and & operator**: All use D3! So, data is a string, but val is a vector<string> ! This is because each of these components is put into a separate rule in the background, and these (background) rules always return

`vector<string>` no matter what the rule definition is – components are put in different rules in order to transform them in strings, to be able to use them in the generation part with $ indexes

- **Group component**: `rule = (c1[D3] (<any expression>)[D2])[D1]`
  - The c1 component uses D3 for the same reason literals, bracket expressions and parser use it (as explained above)
  - Any inner parantheses use D2 and don't have data, because they can be of any complexity - like for the above components, we move group components into other rules, but only the outer-most group components and not inner ones, because it would add too much complexity to the parser
  - The outer-most component uses D1 and has `data` of type `vector<string>`, because it synthesizes a string and can use quantifiers so, we need a vector of strings to keep them
  - In the static mode we could use the exclamation mark for group components semantic actions. We may switch between static and dynamic mode and in order to not get an error, we allow the same syntax with '!' after the semantic action function, but the exclamation mark is unused here
- **lexeme[]**
  - `rule = lexeme[<expression>][D1]` uses D1 and `data` is a `vector<string>`
  - `rule = lexeme[c1[D2] c2[D2]]` - inner lexeme components (c1, c2, etc – can be of any complexity) use D2 for semantic actions, so they don't have data
- **Subrules:**

```
rule1 = <definition> <%String to generate%>;
rule2 = rule1[D1]
```

  `data` will be of type `vector<string>` because the rule can have quantifiers (like * and +) and then it takes all strings. In rule2, rule1 returns the string it generates (or another string, generated in a semantic action)
- **End of the rule**: `rule = <definition> <%String to generate%>[D1];` - again D1, and `data` is of type `vector<string>` but stores all the previous components !!! so you can access this vector the same way you use $ indexes (except that here it starts from 0)

Note:

- For any component, we need to have the brackets of semantic actions attached to the component (i.e. when finished defining a subrule and want to use a semantic action, open a bracket just there, and then write the function and the closing bracket. So, Genesis does not allow for spaces between the definition of the component and the semantic action, because the rule definition would not be understandable anymore). Inside the semantic action's brackets we can put spaces and comments wherever we want
- In dynamic mode we can't have a lexeme[] inside a group component (inside parantheses) because lexeme[] does not return anything and when trying to synthesize a value, the parantheses rule can't deduce it. A workaround for this is to define a rule only for the lexeme and use that rule inside the group component
- In dynamic mode even if everything is a string, in the semantic action (in the function that gets called) you can convert each component in its original type, and use it like this if you need it (eg. $int gets converted to string, but in a semantic action you can reconvert it to an integer)
- At the beginning of this subchapter we said that `val` can be either a `string` or a `vector<string>` and it refers to the already accumulated value. To explain this better, consider the following rule and code:

```
rule = "hello"[f] [a-z][g] <%generated string, %>[h];

void f(..., val, ...) { val  = "called f(), "; }
void g(..., val, ...) { val += "called g(), "; }
void h(..., val, ...) { val += "called h()"; }
```

For example, for the input string *"hello world"*, this rule outputs *"called f(), called g(), generated string, called h()"*. This thing was possible by manipulating the `val` parameter. In `f()` we assign it a value, then in `g()` we update this value by concatenating another string, then it comes the generated string, and then we again add to this *accumulated* value, the string from `h()`. An important thing to keep in mind is that Genesis, when it gets to the generated value of a rule, it uses the += operator to add the generated string to the big grammar value. This means that the generated value will be concatenated to whatever value `val` has accumulated throughout the rule (by semantic action manipulation). Also, note that you can pass values between components of a rule by using `val` – in one component give `val` a value, and in another component (from the same rule) use the passed value

- It's also important to manipulate correctly not only `val` but also `error_handler`. You can again use this to pass values between functions in the same rule or other rules, by changing its value with = or += (if want to accumulate values). It's good to remember to use = and += correctly with `error_handler`. You may want in a function to store in it only a warning (so you'd use the = operator) without changing `pass` to false, and further on in another function, to add an error message (using the += operator) and change `pass` to false to fail the parser – so the problem is to not forget to use = only when want to give `error_handler` (or `val`) a new value, and += when want to add a string to the already accumulated value
- You should know to use $eps (epsilon) in your advantage. You may want to just use a semantic action and not parse anything, so you can use $eps with that semantic action. Also when it comes to accumulating values (in `val`) you can use `$eps[function]` just before `<%%>;` to pre-process `val`, and use `<%%>[another_function];` to post-process `val` (pre-process and post-process in the context of the generated value)

Chapter 4

Usage Scenarios

In subchapter 1.4 we presented some well known DSLs (like SQL or regular expressions) and also, other less known DSLs, targeting different application domains. In this chapter we show what Genesis is capable of doing, and for that, we present some DSLs created in Genesis and provide some usage scenarios for them. Please note that in the following subchapters we'll only present the grammars of DSLs, usage scenarios (input data) for them and how the output looks, and not what each rule generates or what semantic actions it uses, etc. To see the full implementation (called functions, used libraries, semantic actions, etc) of the following samples, refer to the `samples/` folder of the Genesis package, where all samples (with full implementation) can be found.

4.1 Simple Poll

The artifacts generated from a DSL don't have to be just implementation artifacts. We can build a suitable model which can be used to generate build scripts, purchase orders, questionnaires, documentation, bills of materials, plans or skeletons of legal contracts. The sample presented in this subchapter is a simple DSL which allows us to define polls (or questionnaires). This scenario is based on a similar simple poll example from [A53], created with the (Java based) Xtext framework [W8]. With this, we wanted to show that Genesis can build a similar DSL and generate the same output as in [A53].

In Listing 2 we can see the Genesis grammar for the simple poll sample. A poll starts with the *"Poll"* keyword, then it needs a name (which should be between quotation marks), and after that it can have any number of questions. These questions can be text, single choice or multiple choice. A text question translates into a HTML text input. A single choice question can have some options for the user to choose from, and all these translate into HTML radio buttons. A multiple choice question can also have some options, which translate into HTML checkboxes.

```
%grammar PollGrammar@start = Poll;

#skipper skipper@start = $space #end

quoted_string = '"' lexeme[(. - '"')*] '"'
name = lexeme[(. - ( '[' | '(' | $space ) )*]

Poll = "Poll" quoted_string Question*

Question = (TextQuestion | SingleChoiceQuestion | MultipleChoiceQuestion)

TextQuestion = "question" quoted_string name (&"question" | $eoi)
SingleChoiceQuestion = "question" quoted_string name &"()" SingleChoiceOption+
```

```
MultipleChoiceQuestion = "question" quoted_string name &"[]"
                         MultipleChoiceOption+

SingleChoiceOption = "()" quoted_string name
MultipleChoiceOption = "[]" quoted_string name
```

Listing 2. Grammar of the Simple Poll Sample

In Listing 3 we can see an instance of our DSL. When processed with Genesis, this instance translates into HTML code. In Figure 6 we can see the generated HTML file opened in a browser window. When visually comparing Listing 3 and Figure 6 we can see that they are very similar and the listing is very expressive, making it easy to understand even for non technical persons. Also, compare how much HTML code must be written by hand in order to get that, and how easy it was to generate it.

```
use "D:\Genesis\samples\poll\poll.dsl" as poll;

<%poll Poll "Simple Poll"

question "What's your name?" name

question "How do you feel today?" howareyou
() "Fine" fine
() "Great" great
() "Bad" bad
() "Not too bad" not2bad

question "What's your favorite food?" food
[] "Pizza" pizza
[] "Pasta" pasta
[] "Sushi" sushi
[] "Mexican" mexican
%>;
```

Listing 3. Instance of Simple Poll DSL

Figure 6. Generated Output for the Simple Poll Sample

4.2 Shapes

In this sample we manipulate two 3D objects (a cube and a sphere) on a 500x500 board (each square is 10x10 units). In Listing 4 we can see this sample's grammar and in Listing 5 an instance of the DSL we create. The poll sample used Genesis' code generation utility, while this sample uses the interpreter functionality.

At first, we want to be able to initialize the components we work with. We do this by typing Initialize and between curly braces we can modify the attributes of objects. These attributes can be specified either between round parantheses (and change color) or between square brackets (and change position). In our example, we modify the color of the board (a color attribute for each of the two different squares), and also modify the color of both square and sphere, and assign them an initial position. Before beginning, we may also want to change the perspective. We do this with the *"Perspective"* keyword and a pair of attributes between square brackets for the position and look of the camera. Afterwards, we can start moving the square of sphere the way we want. Our DSL allows objects to move left, right, up and down. If mentioned only once, the object moves once in the specified direction, but we can also put a number after the direction and move the object that number of times. Lastly, in the context of objects or outside the place they move, we can delay actions with the keyword "Delay" followed by the number of milliseconds to delay.

```
%grammar ShapesGrammar@start = Init;

#static

#skipper
      skipper@start = $space
      |    ("/*" ($char - "*/")* "*/") // C comments
      |    ("//" ($char - $eol)* $eol) // C++ comments
#end


Init = "Initialize" "{" Config "}"
      (Perspective | Shape | Delay)*


Config = Board ^ Cube ^ Sphere

Board = "board"
      "(" ($int % ',') ")"
      "(" ($int % ',') ")"

Cube = "cube"
      "(" ($int % ',') ")"
      "[" ($int % ',') "]"

Sphere = "sphere"
      "(" ($int % ',') ")"
      "[" ($int % ',') "]"


Perspective = "Perspective"
            "[" ($int % ',') "]"
            "[" ($int % ',') "]"

Shape = "Shape" ("cube" | "sphere")
      "{" (Move | Delay)* "}"
```

```
Move = "move" (
            ( ( "up" | "down" | "left" | "right" ) $int)
            | ( "up" | "down" | "left" | "right" )
            )

Delay = "Delay" $int
```

Listing 4. Shapes Sample Grammar

The code shown in Listing 5 is intepreted and the "output" is the sequence of actions which manipulate graphical objects as described in code. A snapshot from when running the interpreter can be seen in Figure 7.

```
use "D:\Genesis\samples\shapes\shapes.dsl" as sh;

<%sh

Initialize{
      board (0, 145, 195)(252, 250, 218)
      cube (253, 253, 0)[30, 0, 20]
      sphere (78, 197, 56)[35, 5, 15]
}

Perspective [30, 75, -30][30, 0, 40]
Delay 200

Shape cube
{
      move right
      move down 2
      move left 2
      move up 2
      move right
}

Perspective [-20, 75, -20][30, 0, 40]

Shape sphere
{
      move left
      move up 2
      move right 2
      move down 2
      move left
}

Perspective [30, 75, -30][30, 0, 40]

Delay 2000

%>;
```

Listing 5. Instance of Shapes DSL

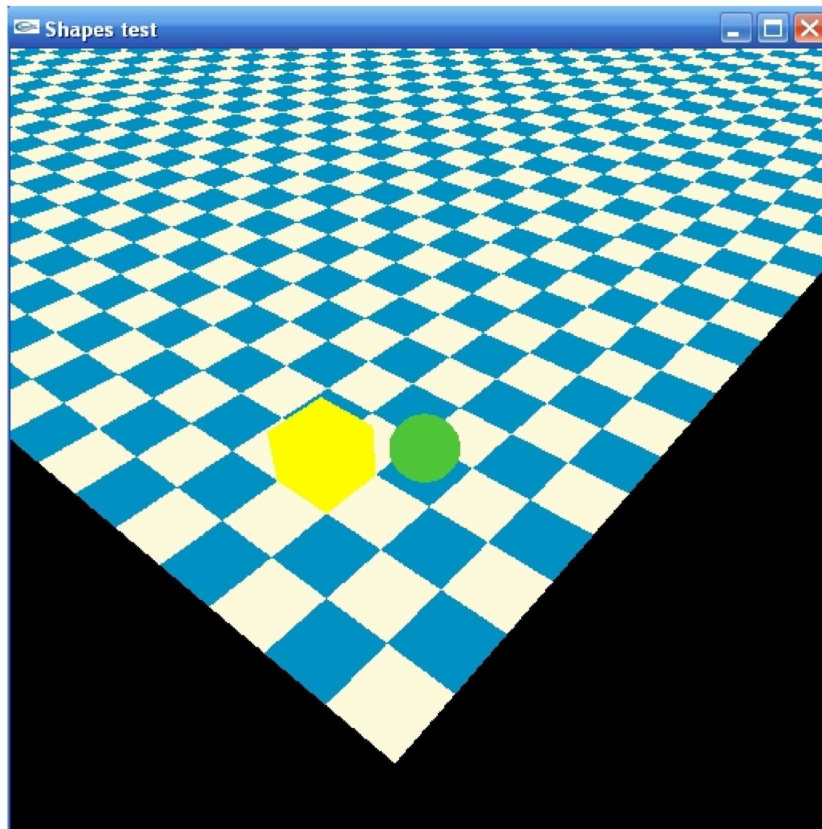

Figure 7. Instance of Shapes Sample Being Interpreted

Such a DSL and the way it's interpreted, is a good starting point for thinking of more realistic examples which could be adapted: the movement of a robot or a car, simulations in meteorology, military or medical fields, and so on.

4.3 Security Protocols

This sample targets the security field and it's about making a DSL which allows us to write security protocols. In Listing 6 we can see this DSL's grammar. First we may want to give our protocol a name – we do that with the *"Protocol"* keyword and specify a name between quotation marks. Then, we need to specify the components the protocol will use. These components can be: agents, servers, intruders, timestamps, nonces, public keys, shared keys or signatures. Once we declared what we need, we can start writing our protocol. This is done in the classical manner: write the number of the step we are currently at, the sender and the receiver with an `->` in between, a colon, the message, and end the step with a point.

```
%grammar SecurityProtocolsGrammar@start = Protocol;

#skipper
      skipper@start = $space
      |    ("/*" ($char - "*/")* "*/") // C comments
      |    ("//" ($char - $eol)* $eol) // C++ comments
#end
```

```
Protocol = Name? Setup ProtocolSteps

Name = "Protocol" Text
Setup = (Component ConstDeclarations ";")+
ProtocolSteps = ($int "." ID "->" ID ":" Message ".")+

Component = ( "Agent" | "Server" | "Intruder" | "Timestamp"
            | "Nonce"  | "PKey" | "SKey" | "Signature")

ConstDeclarations = (ID % ",")

ID = [a-zA-Z_] [a-zA-Z0-9_]*
Text = '"' lexeme[($char - '"')*] '"'

Message = ((CryptoComponent | ID | Text) % ",")
CryptoComponent = "{" Message "}" ID?
```

Listing 6. Security Protocols - Grammar

Having this DSL we can start writing various protocols. In Listing 7 we can see the Needham-Schroeder-Lowe public key protocol and in Listing 8 the Kerberos protocol.

```
use "D:\Genesis\samples\secProtocols\secProtocols.dsl" as sp;

<%sp
Protocol "Needham-Schroeder-Lowe Public Key"

Agent A, B;
Nonce NA, NB;
PKey K_A, K_B;

1. A -> B : {NA, A}K_B .
2. B -> A : {NA, NB, B}K_A .
3. A -> B : {NB}K_B .
%>;
```

Listing 7. Needham-Schroeder-Lowe Public Key Protocol

```
use "D:\Genesis\samples\secProtocols\secProtocols.dsl" as sp;

<%sp
Protocol "Kerberos"

Agent A, B;
Server S;
Nonce NA, NB, NS;
PKey K_AB, K_BS, K_AS;

1. A -> S : A, B .
2. S -> A : {NA, B, K_AB, {NS, K_AB, A}K_BS }K_AS .
3. A -> B : {NS, K_AB, A}K_BS , {NA, A}K_AB .
4. B -> A : {"NB + 1"}K_AB .
%>;
```

Listing 8. Kerberos Protocol

When parsing an instance of this DSL, it should print on screen the components matched and if there were any errors (redefined components, use of undefined components, wrong step numbers, etc). In Listing 9 we can see the output from processing the Needham-Schroeder-Lowe protocol. It first prints the name of the protocol, the defined components, and then, for each step, the sender and the receiver, and the name of the used components.

```
===Protocol Needham-Schroeder-Lowe Public Key===

Agent A
Agent B
Nonce NA
Nonce NB
Public Key K_A
Public Key K_B


===Step 1===
Sender[Agent]: A
Receiver[Agent]: B
{
   NA  <- Nonce
   A  <- Agent
}K_B  <- Public Key Encryption

===Step 2===
Sender[Agent]: B
Receiver[Agent]: A
{
   NA  <- Nonce
   NB  <- Nonce
   B  <- Agent
}K_A  <- Public Key Encryption

===Step 3===
Sender[Agent]: A
Receiver[Agent]: B
{
   NB  <- Nonce
}K_B  <- Public Key Encryption
```

Listing 9. Output from Processing the Needham-Schroeder-Lowe Public Key Protocol

A DSL of this kind is a good starting point for more complex solutions; we can: create theorem provers for security protocols, simulate the exchange of packets on the network or simulate intruders trying to break a protocol, generate generic implementations for GPLs, and so on.

4.4 Packet Filter

Lastly, we present another security sample, this time a packet filter. Its grammar is presented in Listing 10, and its implementation is based on WinPcap [A54], a packet capture library. Here, we want to start a new capturing session and then analyze the captured packets. We need to pick a device on which to "listen" for packets, choose an output file (a file in which the capture will be dumped), possibly a filter and an analyzer. For analysis we use an external tool, in our case the *tshark* analyzer which comes with the Wireshark [W9] network protocol analyzer. After we

specified these options, the packet capture session starts on the mentioned device, and we end it by pressing the return key. After that, we analyze the capture with the `show` command. We can show all packets (`show all`), show all packets which match a filter (`show all "filter"`), show a certain frame (`show frame`) or show a frame in an extended form (`show frame extended`).
Please note:

- Rule Show is important in that it shows how Genesis (and Spirit) behaves and how we should write rules correctly: if we have two rules and one rule is the prefix of the other, then we must put the longer rule first and then its prefix. Because Spirit (and implicitly Genesis) is a greedy parser, in an alternative sequence it will choose the first rule that matches. Thus, in our case if we had `show all` and then `show all Text`, it would always match `show all` but never `show all Text` because the first rule matched is always successful. Thus, we need to be careful how we write rules in Genesis
- Because the listing of the captured packets can be long, we have a PressKey rule which is identified in a DSL instance by the @ symbol. This means that at that point, all actions must stop until the user hits the return button. In grammar, we can see that it does not generate anything, but the rule has the `waitKeyPress` semantic action which has only one instruction : `cin.get()`

```
%grammar CapturePacketsGrammar@start = Init;

#skipper
      skipper@start = $space
      |    ("/*" ($char - "*/")* "*/") // C comments
      |    ("//" ($char - $eol)* $eol) // C++ comments
#end


Init = (Session | Show | PressKey)*

Text = '"' lexeme[($char - '"')*] '"'

Session = "session" "{" Device OutputFile Filter? Analyzer "}"

Device = "pick" "device"
OutputFile = "output-file" "=" Text
Filter = "filter" "=" Text
Analyzer = "analyzer" "=" Text

Show = "show"  (
              ("all" Text)
              | "all"
              | ("frame" "extended")
              | "frame"
              )

PressKey = "@" <%%>[waitKeyPress]; //cin.get();
```

Listing 10. Packet Filter's Grammar

In Listing 11 we can see an instance of our DSL and in Figure 8 the output of the `show all` command (which captured ping requests and replies).

```
use "D:\Genesis\samples\pcap\pcap.dsl" as pcap;

<%pcap

session{
      pick device
      output-file = "out.txt"
      filter = "tcp"
      analyzer = "C:\Program Files\Wireshark\tshark.exe"
}

@ show all
@ show all "udp"

show frame
show frame extended

%>;
```

Listing 11. Instance of Packet Filter DSL

```
1    0.000000 192.168.0.150 -> 209.191.122.70 ICMP Echo (ping) request
2    0.186180 209.191.122.70 -> 192.168.0.150 ICMP Echo (ping) reply
3    0.941969 192.168.0.150 -> 209.191.122.70 ICMP Echo (ping) request
4    1.130853 209.191.122.70 -> 192.168.0.150 ICMP Echo (ping) reply
5    1.885504 192.168.0.150 -> 209.191.122.70 ICMP Echo (ping) request
6    2.073308 209.191.122.70 -> 192.168.0.150 ICMP Echo (ping) reply
7    2.828062 192.168.0.150 -> 209.191.122.70 ICMP Echo (ping) request
8    3.016079 209.191.122.70 -> 192.168.0.150 ICMP Echo (ping) reply
```

Figure 8. Show All Command Output

This sample is important because, based on it we can think of implementing more realistic applications which could benefit from the usage of DSLs : network and protocol analyzers, network monitors, traffic loggers, user-level bridges and routers, network intrusion detection systems, network scanners and other various security tools.

Conclusions and Future Work

This thesis presented Genesis, a C++ library which eases the development and use of textual DSLs. To create a new DSL you need to follow a few steps. First, we have to express the domain specific notions of the problem domain by creating the grammar of the DSL. This implies creating rules for the new language and specifying how they should behave: generate code, execute code (with semantic actions) or both. The code generation functionality is language independent (can generate any kind of text, from plain text to source code of any language), whereas semantic actions rely on C++ to define the way they work. Next, we can start writing instances of our newly created DSL. We can either create stand-alone code or intermingle our DSL code with another (usually general programmning) language. From our DSL grammar and implementation, Genesis generates a Boost.Spirit [W5, W6] parser, which built, creates a binary parser. With it, Genesis parses the DSL instances we wrote, consumes them, and generates another code in place (if we choosed to generate code) or executes the specified semantic actions (if we choosed this), or can even do both.

Even though powerful, Genesis can still be enhanced with new features. Genesis implements most of the features provided by Boost.Spirit.Qi, but it should implement all of them in order to create even more powerful parsers. The code generation utility should also have an escaping character available because otherwise, if we want to generate code for a scripting language like Perl which also uses *$number* expressions, we may get unexpected results or even crash the application. Genesis comes under the form of a library, and also has a command line interface application which uses the library to show its features, but a GUI application would be more useful because it could implement a lot of features that could help the user in creating, testing and using DSLs. Last but not least, certain optimizations could and should be made to get Genesis work even faster.

Although frameworks for developing DSLs exist, most of them are based and target languages like Java or C#. Growing in level of abstraction (and adding a large number of features) usually results in slower applications. Instead, we can use Genesis to create our high level languages, and either generate or execute code out of instances of these languages, without getting any penalty. Thus, with Genesis we try to create a new perspective on developing DSLs in C++, and also take advantage of the benefits of the C++ language: create highly optimized code for high-performance applications in order to tackle some of the most difficult problems, and produce cutting-edge solutions.

References

Books

[B1] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley. August 2006
[B2] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley. September 2010
[B3] Terence Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages*. The Pragmatic Programmers. May 2007
[B4] S. Cook, G. Jones, S. Kent and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley. June 2007
[B5] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley. March 2009
[B6] Ayende Rahien. *DSLs in Boo: Domain Specific Languages in .NET*. Manning Publications. January 2010

Articles

[A1] A. van Deursen, P. Klint and J. Visser. Domain-Specific Languages: An Annotated Bibliography *ACM SIGPLAN Notices* 35(6):26-36. June 2000
[A2] J. Heering and M. Mernik. Domain-Specific Languages for Software Engineering. *Proceedings of the 35th Hawaii International Conference on System Sciences.* 2002
[A3] A. Annamaa. MetaBorg: Domain-specific Language Embedding and Assimilation. *Technical report*. November 2009
[A4] T. L. Veldhuizen. Active Libraries and Universal Languages. *Doctoral Dissertation*. Indiana University Computer Science. May 2004
[A5] M. Mernik, J. Heering and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, Vol. 37, No. 4, December 2005, pp. 316-344
[A6] E. N. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software - Practice And Experience*, Vol. 30, No. 11, 2000, pp. 1203-1233
[A7] F. E. Hernandez and F. R. Ortega. Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games. *Proceedings of the 10th SPLASH Workshop on Domain-Specific Modeling*. 2010
[A8] C. Gaucherel, N. Giboire, V. Viaud, T. Houet, J. Baudry and F. Burel. A domain-specific language for patchy landscape modelling: The Brittany agricultural mosaic as a case study. *Ecological Modelling* 194:233-243. 2006
[A9] D. Spinellis and D. Gritzalis.A Domain-specific Language for Intrusion Detection. *Proceedinds of the 1st ACM Workshop on Intrusion Detection Systems ACM*. November 2000
[A10] D. Kramer, T. Clark and S. Oussena. Mobdsl: A domain specific language for multiple mobile platform deployment. *Proceedings of the IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, IEEE. 2010

[A11] A. A. Kejriwal and M. Bedekar. MobiDSL – a Domain Specific Language for Mobile Web: developing applications for mobile platform without web programming. *Proceedings of 9th OOPSLA Workshop on Domain Specific Modeling*. Orlando, Florida. October 2009
[A12] D. Kramer, T. Clark and S. Oussena. Platform Independent, Higher-Order, Statically Checked Mobile Applications. *Intentional Journal Of Design, Analysis And Tools For Circuits And Systems*, Vol. 1, No. 1. May 2011
[A13] S. Thibault, R. Marlet and C. Consel. A domain-specific language for video device drivers: From design to implementation. *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 11–26. Berkeley, CA, 1997
[A14] Jeroen van den. Bos. Domain-Specific Languages for Digital Forensics. *Doctoral Symposium of the International Conference on Software Language Engineering*. 2010
[A15] P. G. Bradford and D. A. Ray. Models of Models: Digital Forensics and Domain-specific Languages. *Cyber Security and Information Infrastructure Research Workshop*. Oak Ridge National Laboratory. 2007
[A16] C. Brabrand and C. Consel. Call/c: A domain-specific language for robust internet telephony services. *Research Report*, LaBRI. Bordeaux, France, 2003
[A17] C. Consel and L. Reveillere. A DSL paradigm for domains of services: A study of communication services. *Domain-Specific Program Generation, International Seminar.* Dagstuhl, 2004
[A18] R. Tairas, S.H. Liu, F. Jouault and J. Gray. Coclorep: A DSL for code clones. *Workshop on Software Language Engineering*. Nashville, Tennessee, USA, 2007
[A19] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. *Proc. Conf. Generative Programming and Component Engineering* (GPCE '03), Lecture Notes in Computer Science, Springer-Verlag. 2003
[A20] R. Heinzl, P. Schwaha, M. Spevak and T. Grasser. Performance Aspects of a DSEL for Scientific Computing with C++. *Proceedings of the POOSC Conference*, pp. 37–41. Nantes, France, 2006
[A22] P. Degenne, D. Lo Seen, D. Parigot, R. Forax, A. Tran, A. Ait Lahcen, O. Cure and R. Jeansoulin. Design of a domain specific language for modelling processes in landscapes. *Ecological Modelling,* In press. 2009
[A23] P. Salgueiro and S. Abreu. On using Constraints for Network Intrusion Detection. *INForum 2010 - Simposio de Informatica*. Braga, Portugal, 2010
[A24] H. Hamdi, M. Mosbah and A. Bouhoula. A domain-specific language for securing distributed systems. *Proceedings of the International Conference on Systems and Networks Communication*. Cap Esterel, France, August 2007
[A25] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. PLDI '05: *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, ACM Press, pages 295–304. New York, NY, USA, 2005
[A26] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger and M. D. Corner. Flux: A language for programming high-performance servers. *Proceedings of USENIX Annual Technical Conference*. May 2006
[A27] J. Sorber, A. Kostadinov, M. Brennan, M. Garber, M. Corner and E. D. Berger. Eon: A Language and Runtime System for Perpetual Systems. *Proc. ACM SenSys*. November 2007
[A28] E.G. Sirer and K. Wang. An access control language for web services. *Proc. ACM SACMAT,* 2002
[A29] M. Mecella, M. Ouzzani, F. Paci and E. Bertino. Access control enforcement for conversation-based web services. *Proceedings of the International Conference on World Wide Web*. 257–266. 2006
[A30] M. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth and E. Wall "Implementing a generalized tool for network monitoring". *Proc. LISA '97,* USENIX 11th

Systems Administration Conference. San Diego, October 1997
[A31] V. Paxon. Bro: A system for detecting network intruders in real-time. *Proceedings of the 7th USENIX Security Symposium*. San Antonio, TX, 1998
[A32] P. Klint, T. van der Storm and J.J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. *IEEE International Workshop on Source Code Analysis and Manipulation* (SCAM'09), 168-177. Los Alamitos, CA, USA, 2009
[A33] L. Kagal, T. Finin and A. Joshi. A Policy Based Approach to Security for the Semantic Web. *2nd International Semantic Web Conference* (ISWC2003). September 2003
[A34] D. Atkins, T. Ball, G. Bruns and K. Cox. Mawl: A domain-specific language for form-based services. *DSL-IEEE*, pages 334–346. June 1999
[A35] P. White. Security Configuration Domain Specific Language (DSL). *SELinux Developers Summit*. Ottawa 2008
[A36] B. Agreiter. Towards Application-Oriented Policy Configuration for SELinux. *SELinux Developers Summit*. Ottawa 2008
[A37] S. N. Kamin and D. Hyatt. A Special-Purpose Language for Picture-Drawing. *Proceedings of the Conference on Domain-Specific Languages*. Santa Barbara, California, October 1997
[A38] X. Ou, S. Govindavajhala and A. W. Appel. MulVAL: A logic-based network security analyzer. *14th USENIX Security Symposium*. Baltimore, MD, USA, August 2005
[A39] M. Zubair. Design Patterns for Programmable Parameter Control for Evolutionary Algorithms. *Master Thesis*. Department of Computer Science, California State University, Fresno. December 2009
[A40] S. Dower and C. Woodward. Evolutionary System Definition Language. *Technical Report.* Swinburne University of Technology. 2010
[A41] F. Fleurey and A. Solberg. A Domain-Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. *Proc. ACM/IEEE 12th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 09), ACM Press. 2009
[A42] M. W. Weissmann. Domain Specific Language for Specifying Access Controls. *Diploma Thesis*. Georg-Simon-Ohm University of Applied Sciences, Faculty of Computer Science. 2007
[A43] Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Modersheim and L. Vigneron. A high level protocol specification language for industrial security sensitive protocols. *Proc. Workshop on Specification and Automated Processing of Security Requirements* (SAPS). 2004
[A44] L. Erkok and J. Matthews. High assurance programming in Cryptol. In CSIIRW'09: *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, ACM. Oak Ridge, TN, April 2009
[A45] S. Pakin. The Design and Implementation of a Domain-Specific Language for Network Performance Testing. *IEEE Transactions On Parallel And Distributed Systems*, Vol. 18, No. 10. October 2007
[A46] S.T. Eckmann, G. Vigna and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Proceedings of the ACM Workshop on Intrusion Detection Systems*. Athens, Greece, November 2000
[A47] J. Jurjens. A domain-specific language for cryptographic protocols based on streams. *Journal of Logic and Algebraic Programming* (JLAP): 54–73. 2009
[A48] J.D. Nielsen and M.I. Schwartzbach. A domain-specific programming language for secure multi-party computation. *Proceedings of Programming Languages and Security* (PLAS), ACM press. 2007
[A49] A. Shaffer, M. Auguston, C. Irvine and T. Levin. A Security Domain Model to Assess Software for Exploitable Covert Channels. *Proceedings of the ACM SIGPLAN Third Workshop on Programming Languages and Analysis for Security* (PLAS'08), ACM Press,

45-56. Tucson, Arizona, 2008
[A50] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *Symposium on Principles of Programming Languages*. Venice, Italy, January 2004
[A51] Todd Veldhuizen. Expression templates. *C++ Report*. June 1995
[A52] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, Vol. 7, No. 4, pp. 36-43. May 1995
[A53] Peter Friese. Building DSLs with Eclipse. *Eclipse Day*, GooglePlex. August 2009
[A54] L. Degioanni, M. Baldi, F. Risso and G. Varenni. Profiling and Optimization of Software-Based Network-Analysis Applications. *Proceedings of the 15th IEEE Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2003), Sao Paulo, Brazil. November 2003

Web resources

[W1] Graphviz - Graph Visualization Software, Finite state machine sample, http://graphviz.org/content/fsm
[W2] mobl - Building Mobile Applications, http://www.mobl-lang.org/
[W3] Fluent Interfaces, Martin Fowler, http://martinfowler.com/bliki/FluentInterface.html
[W4] Fluent Interfaces example, Wikipedia, http://en.wikipedia.org/wiki/Fluent_interface
[W5] Boost.Spirit Home, http://boost-spirit.com/home/
[W6] Boost.Spirit Tutorials and References, Version 1.46.1 http://www.boost.org/doc/libs/1_46_1/libs/spirit/doc/html/index.html
[W7] Boost Libraries, http://www.boost.org/
[W8] Xtext Language Development Framework, http://www.eclipse.org/Xtext/
[W9] Wireshark Network Protocol Analyzer, http://www.wireshark.org/