A Pattern Language for Language Implementation

Joel Jones jones@cs.ua.edu, Crutcher Dunnavant crutcher@samedi-studios.com, and Trevor Jay aka1@samedi-studios.com

University of Alabama, Computer Science
Tuscaloosa, AL USA 35487-0290

Abstract. Programming languages are typically considered to be difficult to implement. However, a programming language tailored to an application domain can be an extremely powerful productivity enhancer. We present a pattern language for implementing languages that gathers together many ideas that are known in the language implementation community that should be more widely known. By applying this pattern language, productivity can be increased by the blossoming of more programming languages tailored to a specific purpose.
We begin by discussing some of the dichotomies that shape the language design process. The pattern language itself consists of patterns describing various language flavors, and patterns for doing syntactic recognition, evaluation, and source production.

1 Introduction

Good specification and programming languages provide powerful means of expression within their domains. While there has been much discussion of when producing a new language is appropriate, it has largely been provided by authors as introductory material when describing a new language, and as such, has been largely cursory.

A decision process for determining *if* it is appropriate to implement a new language is difficult to supply. The reason for creating a new language is that no existing language is satisfactory. One source of dissatisfaction is the lack of ability of an existing language to capture the information present in a domain. The other source is the lack of expressiveness regarding the common actions in a domain. All things are possible in any given Turing-complete language, but pragmatics dictate that there are advantages of using a language better suited to given purpose. This is demonstrated by the presence of more than one general-purpose programming language.

The following is a collection of common situations where a new language is appropriate. These situations are not intended to be completely distinct.

Single Point of Truth[30] If a single application has more than one description of the same information, then keeping these different descriptions from conflicting is difficult. The reason why such conflicts exist comes from the

lack of ability to express the information in a single place. This may occur either because two or more implementation languages are used and are not easily cross-callable or the one language is not powerful enough to state the same truth only once. This is an example of the lack of a language (or languages) ability to capture the information of a domain.

For example, in the Press Pot Java annotation system[23], a single file maintains information describing bytecodes. By generating code from this description language to both C and Java, Press Pot is able to keep the semantic description always in sync in both the Java front-end and the C back-end.

As another example, in C programs, it is difficult to express a data file format in a single declaration and from that description drive both serialization and deserialization without using some sort of description language. From a description language, serializers and deserializers can be generated and consistency insured.

Also known as "Once and only Once"[5]. Formalized in [21].

Existing Conventions If there is an existing notation for expressing some formalism, a language representation can allow the manipulation and display of the described information or processing. A domain specific language can provide expressiveness, exemplified by the conciseness to express an idea as compared to an equivalent expression in a general purpose language.

For example, regular expression[13], database queries (SQL)[10], tables, molecular diagrams[6], graphs (grap)[27], parsers, equations[28], logic, all document formatting all have existing conventions and corresponding computer languages. The language pic[26], for specifying line drawings mimics the conventions used in verbal description—`box; arrow; box`.

Configuration The exclusive use of graphical user interfaces to specify the variants in the behavior of a program limits the abilities of the user and/or administrator in various ways. Many applications are very general and specialization for a particular environment is almost always required. A graphical user interface does not readily capture the information in the configuration domain. Useful capabilities that a textual configuration language enables include: automation, version control, and automatic generation. Another advantage is that, in many cases, a text editor may be simpler than any other user interface. Examples include almost every Unix service ever written—init, cron, mount (fstab), apache, etc. Configuration languages for interactive programs are also prevalent—elisp, TCL, lua, and gdb.

Glue Expressiveness of actions, as mentioned above, is a motivation for implementing a new language. While there have been general purpose languages designed for connecting together components in high-level way, none have become popular. This lack of ability to connect together components in an ad-hoc fashion has resulted in the use of scripting languages to fulfill that role. Examples include: Perl, AWK, JSP, ASP, JavaScript, AppleScript, and TCL.

2 Design Dichotomies

Once it has been determined that a new language is appropriate, the form of the language must be determined. Just as we did not give patterns for choosing whether to implement a language, we do not give patterns for determining the form of a language. Instead, we provide a characterization of languages based upon a pair of design dichotomies which we feel greatly shapes language forms.

Psychology, psychiatry, and sociology deal frequently with long standing dichotomies of thought. Most famously discussed is the "Madonna-Whore Complex", a dichotomy in western culture's view of women. The human mind appears readily capable of handling complex domains by splitting them into pairs of opposing, but inter-related, paradigms. At any given time, the mind's perspective will be dominated by one side of a dichotomy, but will be informed by the other. The eastern concepts of Yin and Yang are idealized expressions of this tendency—within every extreme are elements of the opposing nature.

2.1 Flavor

The oral, newsgroup, and email tradition of programming talks about languages in terms of their "flavor" or their "feel". It is full of discussions of what a given language "wants to be", or of what a given language "really is". Computer languages are processed by computers, but they are written, read, and debugged by people. They are vehicles for expression, tools for understanding, and shackles that must be worn for nearly 50 thousand hours over the life of a programmer. Perhaps they should be comfortable? But what patterns need we use to achieve a given flavor?

A good pattern should always be a name for a means of resolving design tensions which the community already knew, but did not have a name for. For some patterns, the community will need to identify previously unnamed tensions, so that the pattern may be properly captured. We believe that we have identified a pair of design dichotomies which exert tension on the flavor of computer languages. We make no claim that these dichotomies are the only, or even the most important, conceptual dichotomies in language design.

2.2 The Definitional Dichotomy

We have identified the Definitional dichotomy, which covers a continuum of language semantics, holding at one extreme languages which speak in terms of actions ("Algorithmic" languages), and at the other languages which speak in terms of truths ("Constraint" languages). The primary semantic content of "Algorithmic" languages is instructional, specifying which actions to take, and how to take them. C, Java, Scheme, Smalltalk, PostScript, and most other "programming" languages are dominated by the algorithmic side of the definitional dichotomy. The primary semantic content of "Constraint" languages is descriptive, specifying truths, constraints, and the nature of acceptable processing results. VHDL, HTML, CSS, Prolog, SVG, and most "configuration" or "data" languages are

dominated by the constraint side of the definitional dichotomy. Thus we have the definitional dichotomy of Algorithmic versus Constraint languages.

2.3 The Structural Dichotomy

We have also identified the Structural dichotomy, which covers another continuum of language semantics, holding at one extreme languages which speak in terms of hierarchies ("Hierarchy" languages), and at the other languages which speak in terms of Expressions ("Expression" languages). The primary semantic content of "Hierarchy" languages is expressed by their containment relationships, wherein elements derive their meaning relative to their containing parents and their contained children. XML and Scheme are dominated by the hierarchy side of the structural dichotomy. The primary semantic content of "Expression" languages is expressed through the interaction of operators and peer sequence, often evolving into very complex sequential Expressions. C, Java, SQL, and most 'imperative' languages are dominated by the expression side of the structural dichotomy.

3 The Pattern Language

Having dispensed with *whether* and *what*, we come to *how*. Our pattern language addresses how to implement languages that are outside of the domain of traditional high-level language to machine code compilers. Table 1 is a table dividing the pattern language into categories based upon the purpose of each pattern. The listing of patterns from top to bottom is the typical order in which the patterns are used during language processing. A category which is marked as optional indicates that not all applications of this pattern language will use that category of pattern. Cells in the table that contain pattern names indented represent patterns that are subpatterns—specializations of the pattern at the top of list. Patterns in the right-most column are patterns that tend to be used together to implement simple data or constraint style languages.

Each group of patterns is summarized by their purpose:

Higher-order Syntax optional group of patterns that is used when a language is being implemented by translation into a high-level language.

Base Syntax patterns describing the syntax of the language being implemented.

Parsing Technique patterns describing the parsing techniques used to recognize the language being implemented.

Intermediate Representation patterns describing the representation and implementation used for the input language within the translation system.

Transformation Techniques optional patterns describing very common techniques for doing source to source translation.

Execution Techniques patterns describing techniques for executing languages that are not implemented using translation to machine code.

Infrastructure patterns describing techniques used in implementing the other patterns.

Purpose	Patterns	
Higher-order Syntax (optional)	High-order Features Language Composition Embedded Language Language Extension	

Base Syntax	Strictly Contained Language
	  Parenthesis Language
	  XML Language
		Record Language
		  Key-value Pairs
		  Delimiter-separated Values
		  Stanza Formatted Record

Parsing Technique	Generated Parser	Hand-written Parser
		  Cascade Parser
		  Per-type Parser
		  Recursive-descent Parser

Intermediate Representation	Abstract Syntax Tree (AST)
	  Hand-written AST
	  Generated AST
	  Commodity AST
		Flat Intermediate Representation

Transformation Techniques (optional)	Lexical Transformation Tree Transformation	
Execution Techniques	Virtual Machine Interpreter Semantic Evaluator	Record Consumer Immediate Execution
Infrastructure	Runtime Language Output	

Table 1. Guide to the Pattern Language

The Higher-order Syntax and Base Syntax pattern groups describe common syntactic choices which occur well before implementation, when the semantics of a language are still being laid out. While the forces which they resolve are common to language design, many of them have yet to be named. In general, Higher-order and Base Syntax patterns will be selected which provide the best "fit" to the semantics desired for a language.

3.1 High Order Syntax

The first decision when implementing a new language is to decide if the language is to be built on top of an existing language and if so, how. The following patterns give different ways of using an existing language processing system, if appropriate.

3.1.1 High-order Features Most articulated languages posses many features which could be expressed as applications of simpler features of the same language.

A feature is some semantic or syntactic aspect of the language that is readily visible to the programmer. For example, `for` loops in C are a feature.

When language features can be defined in terms of more primitive features of the same language, how can this relationship be used to guide and reduce the cost of implementation?

Many language features can easily be defined as relatively simple applications of other features of the language. "Syntactic Sugar" are such features, but some features go deeper. In most languages, **for**-loops can be defined in terms of **while**-loops and **if**-statements. These features, defined in terms of other features, are High-order Features, as they are produced through composition of lower order features, down to and including zeroth order features, which are atomic.

Therefore:

Seek to identify those portions of your language which can be defined as High-order Features*, and keep these definitions available at later design stages.*

If you have enough High-order Features, implementing them using some transformation pattern, such as Lexical Transformation, or Tree Transformation may become cost effective.

3.1.2 Language Composition The semantics of many richly articulated languages are most properly seen as compositions of simpler languages.[37]

A single computation may be best described not with a language with a uniform flavor, but with multiple languages. The results of the individually described computations can be performed relatively independently and then composed using a simple model such as string concatenation.

How should a language be structured when the computation to be described has pieces that do not fall close together in the definitional or structural dichotomy spectrums?

We are quite good, as people, in dealing with compositions of language structures and semantics, mixing and matching special purpose language, such as for medicine or mathematics, with our general purpose language. There is no reason why we cannot build our processing environments to do the same. The mixing of both natural languages with each other and of formal languages with each other are processed in a similar fashion, with lexical markers delimiting the transitions between different languages.

Some languages have rich semantic features, and you may find yourself having difficulty planning an architecture for processing them. This problem is frequently solved by composing the semantics of several languages, and processing each layer's language into input for one of the later layers' language.

Therefore:

Implement rich and subtle languages by transforming them into simpler languages. Your processing environment should first resolve any High-order Features *in your input language, and should then apply some transformation technique to produce the composed language for processing.*

As noted by Spinellis[37] and others, the Unix troff document typesetting pipeline is an archetype for this pattern. Languages exist for describing data graphs in terms of a general purpose language for drawing, which is translated into lower-level line drawing and text primitives.

3.1.3 Embedded Language You can not lick your elbow, but sometimes it is the best solution.

Frequently, a given language is almost ideal, save for a small and critical piece of missing functionality.

You may find that for the most part one particular pattern/paradigm for your language's syntax will work well, except for one or two specific and self-contained cases. Writing in the "majority" language is a good fit for the task at hand, but even if providing the missing functionality is possible, it is not desirable. The majority language may be capable of providing the missing functionality, but the effort to write code to do the task may be too high or too error-prone. For example, the ProC language is a superset of C, which is resolved by a preprocessor into C to provide database functionality to C programmers. The use of another language in combination to C makes writing database code easier—by reducing the verbosity—and less error prone—by permitting checks against database schema at compile time, rather than at run time.

Therefore:

Rather than try and "fix" the language for which it is awkward to express certain semantics, embed another language inside it. [37]

Write pre-processors which "resolve" the Embedded Language into expressions in the base language, before further language processing. In languages eventually destined to be interpreted or converted to some "generic" programming language such as C, embedding this higher-order language can be almost trivial. At worst two parsers and language processing steps must be written, but

this cost is often mitigated by the fact that the base language is often a popular language for which many processing tools exist.

ProC; Lex and Yacc's input languages; and Emacs' use of Scheme are good examples of the use of Embedded Language. The embedded language code will typically be translated to calls into a RUNTIME.

3.1.4 Language Extension You may be able to lick your elbow, but only with help.

You have a language that is almost what you need, but is missing some functionality. Further, the language provides extension mechanisms. What is the best way to implement additional semantics using those mechanisms?

The forces which drive LANGUAGE COMPOSITION and EMBEDDED LANGUAGES are common; and so, certain types of language enhancements can be anticipated by a language implementor. As a result, some languages have been designed to permit extension directly in their existing semantics.[37]

For example, ML provides very robust mechanisms for type and operator extension; Scheme provides powerful macros; C++ provides mechanisms for operator overloading; and even C provides a reasonably flexible lexical macro system. However, it is very easy to overuse these features to produce code that is incomprehensible to others.

Therefore:

When you need features which represent a superset of the semantics of an existing language (which you are comfortable with), and the language provides mechanisms for extending its semantics, produce those features directly in the language's extension mechanisms, but maintain understandability.

The problem domain should drive whether or not a particular extension feature is appropriate. For example, If the problem domain forms an algebra, then it is appropriate to use operator overloading, *if* there is a natural mapping from the semantics of the pre-defined operators to the operators of the problem domain. In C++, operator overloading is properly used when it is used to implement numeric classes for doing matrix multiplication by overloading `*`, etc. It would not be a good choice of operator overloading to have `+` defined to create new database tables as the composition of preexisting tables, as scheme definition is not really viewed as a algebraic or compositional operation in the problem domain of relational databases.

3.2 Syntax

The choice of the syntax for a programming language contributes greatly to its resulting feel. We present two categories of language syntax patterns—STRICTLY CONTAINED LANGUAGES, which require more sophisticated parsers; and RECORD LANGUAGES, which are typically used as data input to another program, usually for configuration purposes.

3.2.1 Strictly Contained Language One valuable thing to recognize regarding the dichotomies is the fact that languages that fall to an extreme are often very easy to process. One can consciously take advantage of this fact when designing a language. A good example are domains that lend themselves towards being described in a very hierarchic manner.

For languages or domains that are essentially hierarchical, processing models centered around this fact would be ideal. What sort of models would be useful?

Containment languages feature containment syntax (such as parenthesis in Scheme, or tags in XML). Many domains can be readily described by Hierarchy Languages, and it is simple to map hierarchic structures to containment syntax. If we restrict things even further, calling for *all* expressions to be contained and all containers to be contained, with the exception of the root; then we have a language that is very easy to parse, because it can only be a tree, not a complex graph.

Therefore:

Structure your language as a Strictly Contained Language. *A Strictly Contained Language consists of one or more container types. With the exception of the "root" container, all content and containers are themselves contained.*

Strict containment permits the use of some of the simpler parsing techniques. Lisp has always been one of the easiest languages to parse.

Strictly contained languages can be characterized by the kinds and number of containers they feature. At their simplest, such languages only have a single type of container. At their most complex, as seen in languages like SGML, they may have an arbitrary number of containers of different types.

Strictly Contained Languages are usually dominated by the hierarchical side of the structural dichotomy; though even Expression Languages such as C often possess several forms of containment. Function, structure, and loop definitions are examples of containment forms in C. The use of the Strictly Contained Language pattern places no constraints on the definitional dichotomy, and these languages may easily be constraint or algorithm based.

Two popular forms of Strictly Contained Languages are Parenthesis Languages and XML Languages.

3.2.1.1 Parenthesis Language Even though a Strictly Contained Language is a rather restricted context, there are still a number of syntactic choices that can be made based on the nature of the domain the language is attempting to cover. Once a containment based language has been chosen as appropriate, the remainder of decisions tend to concern the level of complexity of that containment.

Strictly Contained Languages *intended for domains which have very simple containment or hierarchal models need to be able to take advantage of this fact so as to ease processing and allow the language to map easily to the domain.*

For simple hierarchy structures a unilateral containment model is appropriate and only one "type" of container is needed. There need not be any distinction between the root and all its children save that the root is first. Parenthesis are a logical and popular container.

Use a Parenthesis Language. *A* Parenthesis Language *is a* Strictly Contained Language *where there is only one kind of container, the parenthesis.*

While few utilities and libraries exist for explicitly handling parenthesis languages, the process is quite straight-forward. The usual approach is to recursively "resolve" each container's content. This can be done with handwritten or generated code quite easily. Large subsets of LISP can be handled in this way.

3.2.1.2 XML Language Sometimes, instead of a straight forward containment model, a more complex hierarchy is appropriate. Some hierarchical domains are inherently typed, and for these domains the ability to distinguish between containers by type becomes crucial. In cases where the structure is likely to be extremely complex it may be best not to "reinvent the wheel" but instead use a preexisting meta-language.

Use an XML Language*, that is, a markup language that is valid XML.*

There are several advantages to using an XML Language. Foremost amongst these is the fact the widespread availability of XML libraries, such as Xerxes[34] for Java and libxml[15] for C. These libraries mean that even lower level implementation languages, such as C, will not require programmers to write their own parsers or ASTs.

XML is by far the most popular data interchange language and as a result is the meta-language most likely to be familiar to users. Tools for its use are available in almost any system environment. XML is particularly well suited for constraint languages, but it has been successfully used in more algorithmic ways as well.

Since XML content is so easy to transform: Tree Transformation*s and* Lexical Transformation*s are definitely options for processing when an* XML Language *is used.*

3.2.2 Record Language Many times you need a description language for little more than being a form of non-interactive input, a way of shoving data into a machine processable environment. This comes up a lot in contexts like program configuration or in certain kinds of data driven programs.

Some languages are needed only for simple data entry or configuration purposes. For these, a large amount of processing is overkill and undesirable. A simpler language paradigm is needed.

Record Languages are based on the idea that your needed input is mostly constraint based and can be processed on a per-record basis. Record Languages are usually syntactically very simple and used for such tasks as setting a series of values. Typically, Record Languages' records are single line, but many forms have more complex record structures.

Structure your language's semantics and syntax into a Record Language*. A record language has little to no hierarchic structure, and is processed one record at a time.*

A few common forms of Record Languages include: Key-value Pairs, Delimiter-separated Values and Stanza Formatted Records.

3.2.2.1 Key-value Pairs You would think that the decision to use a Record Language would simplify data processing enough, but there are domain specific syntaxes that can save time and effort, even within such a restricted environment.

Some Record Language*s are meant to set attributes or properties. Such a language should take advantage of this fact so that it maps easily to this domain for its human users.*

A Key-value Pairs syntax may be just what is needed in this situation. Key-value Pairs typically consist of a key, the actual attribute or property name, a delimiter (popular is the colon or equals) and a value for that property or attribute.

There are a number of advantages to such a syntax in certain situations. Order no longer need be important, a great boon to human editors. Further, defaults can be assumed for unlisted Key-value Pairs, again a boon to those using the language. Configuration files are frequently structured as key-value pairs, and most applications use some form of configuration file. Thus, key-value record languages are some of the most common computer languages.

When a language exists for the specification of constraints on a collection of properties, use a key-value syntactic structure. Each Key-value Pair *consists of a descriptive key that identifies which attribute's value is being set, some separator, and the value the attribute is being set to.*

It is worth noting that some standardized versions of Key-value Pair languages exist, and that editors called property editors may be present on some systems allowing for a great amount of power in editing these types of files.

3.2.2.2 Delimiter-separated Values A main characteristic of a Record Language is its simplicity. Sometimes however, a degree of flexibility is required. A common situation is one where a Record Language seems appropriate because a property or attribute is being set but this attribute/property may be more like a list, or a collection of properties.

You have a Record Language *and each record has a fixed number of attributes. However, each attribute may have a varying amount of data associated with it.*

This kind of problem can often be solved with creative use of delimiters. Server type programs often need lists of users to allow or deny. Such chores can be handled by having each list appear as a "record" on a separate line with the usernames separated by commas.

Use a Delimiter-separated Values *syntax. Each record is placed on a single line by itself and the attributes of each record are separated by some delimiter, typically a single character.*

A language such as this can be processed largely by a tokenizer working on individual records. Some minor complications may arise depending on whether there are a variable number of records or if record order might need to change. Many of these problems can be addressed by embedding Delimiter-separated Values languages inside Stanza Formatted Records. Stanza Formatted Record is also worth looking into in that they can solve similar problems.

Though it is not a good idea to take it further than two levels, it should be noted that this pattern can be applied somewhat recursively by using multiple delimiters.

3.2.2.3 Stanza Formatted Record Just as much as by its simplicity, a Record Language is marked by its atomic nature. For some problems a solution that is atomic but further along in complexity is necessary. Records that may have too much information to fit on a single line or data that may need to be grouped together.

Use a Stanza Formatted Record*. Stanza records are multi-line records separated by some delimiter. Usually this delimiter also features a label for the record.*

Most Unix configuration files use some form of Stanza format. Usually this is of the labeled variety. Very common as a delimiter is percent signs enclosing the record label. One of the easiest ways to implement a stanza record reader is to have a two state state machine recognizing delimiters and processing the records themselves respectively.

Stanza formatting can be combined with other forms of Record Languages to allow for a variable number of Key-value Pairs or Delimiter Separated list for example.

Stanza Formatted Record has been around a long time[36] and probably represent as far as Record Languages should be taken before more serious processing should be considered.

3.3 Parsers

In processing most languages, sooner or later you will need a Parser. There is no One True Way in parser design, and so there are many parallel patterns for Parsers. The largest split lies between the Generated Parsers, and the Hand-written Parsers, though there are many distinctions at lower levels.

3.3.1 Generated Parser Many languages have a structure which is easily describable in grammars For example, the LALR subset of context-free grammars is capable of some fairly complex and articulated structures.

When a stable grammar exists for a language, and when error recovery is of minimal importance, a class of generated solutions becomes a good choice for constructing the Parser.

A parser generator takes a specification of the grammar of a language and generates a parser for this language. Typically one also uses a lexer generator in conjunction with the use of a parser generator.

The first generation of parser generator tools provide support for generating the tables and engine for a table driven parser and allow the firing off of action code when a production in the source language was recognized. Also, most of these tools can utilize an associated lexer generator and generate a set of token type definitions from the grammar for the source language. This class of tools

includes yacc and lex [31], Bison and flex [31], java_cup [19] and jlex[7], javacc [8], yacc++ [9].

The second generation of parser generator tools adds to the first generation by supporting other aspects of the translation process. Sly [42] and LPT [12] generate code for doing source-to-source translation and the implicit generation of AST definitions. TXL[20] and eli[16] also provide tools for generating such additional pieces.

Therefore:

When working with context-free languages, and when error recovery is of minimal importance, use a Generated Parser *if tools for doing so are available in the implementation environment.*

3.3.2 Hand-written Parser It is not always possible or desirable to use a Generated Parser. Sometimes the complexity of a language's grammar is so low that the development cost of using a parser generator is greater than writing the parser by hand; and sometimes the semantics of the language demand extensibility mechanisms which are not achievable with a Generated Parser.

When a language's parser cannot be readily handled by parser generation tools, a substantially different class of parsers are needed.

Extensible grammars and sophisticated error reporting and recovery are difficult to achieve with Generated Parsers; and the grammars for some languages are so simple as to not require the use of such tools.

Therefore:

When a language's costs or processing semantics demand, use a parser written directly in the implementation language. There are many ways to implement such a parser. The choice is driven by the implementation language and the complexity of the language.

You may be able to implement Hand-written Parser using Cascade Parser, Per-type Parser, or Recursive-descent Parser.

3.3.3 Cascade Parser In many environments, such as parsing command lines, tokenization is unnecessary or already provided.

When lexing and first level structure are provided by a language's environment, and when a language need not be extendable, the semantic actions of the parser dominate the cost dynamics.

The high cost of development of most parsers lies in the need to structure a input stream. In some situations though (such as command lines), the input is already structured into tokenized records of some form. In this situation, all that the parser need do is detect the different configurations a record may be in, and effect the semantics of the language. This can be accomplished by a simple cascade of if-else statements, or a switch.

Therefore:

When an environment provides token statements, and a language does not require extensibility; use a Cascade Parser. *Construct a cascade of if-else statements or switch statements to handle the language's semantics.*

The dominant feature of a Cascade Parser is the cascade, which detects and handles the various record configurations of the input. If the language requires extensibility, consider a Per-type Parser instead. Many scripting languages, notably AWK, have a Cascade Parser as their primary execution paradigm.

3.3.4 Per-type Parser In many environments, such as parsing command lines, tokenization is unnecessary, or rather, already provided. Some languages for these environments require extensibility, often during execution; a common way to achieve this is by defining the semantics of records which begin with some identifying keyword.

When lexing and first level structure are provided by a language's environment, and when a language is statement based, and structured in the form KEYWORD ARGS, but the language requires extensibility, handler dispatch dominates the cost dynamics.*

The high cost of development of most parsers lies in the need to structure a input stream. In some situations, such as the command line, the input is already structured into tokenized records of some form. In this situation, all that the parser need do is detect the different configurations a record may be in, and effect the semantics of the language. If the language requires extensibility, a dispatcher will be necessary, to associate keywords with the appropriate handler for that type.

Therefore:

When an environment provides token statements, and a language is keyword based, and requires extensibility; use a Per-type Parser.

The dominant feature of a Per-type Parser is the dispatcher, which looks up the appropriate Per-type handler for a given statement based upon its keyword, and then dispatches the statement to the handler. If the language does not requires extensibility, consider a Cascade Parser instead. In an Object-oriented implementation language, the dispatch is to a Factory Method[14] or constructor.

3.3.5 Recursive-descent Parser In some situations, error reporting and recovery is a very important factor. With the Generated Parsers, it is often difficult to know where an error happened in the parse tree.

When error reporting and recovery is important, a parser must provide means of passing through known states.

LL(1) grammars can be parsed by Recursive-descent Parsers, and it is always possible to know "where you are" deterministically in the parse tree of a Recursive-descent Parser, so it is always possible to provide good error semantics.

Therefore:

When a language must provide high quality error reporting and recovery, and the grammar for the language is LL(1) and does not require extensibility, use a Recursive-descent Parser.

Build the parser by manually translating the grammar rules into recursive code in the implementation language which recognizes the language by identifying tokens and recursively identifying productions of the grammar[41].

3.4 Intermediate Representations

All but the simplest Immediate Execution languages need some representation form other than the form used by the parser. In this section, we present patterns for presenting programs in an "intermediate form." This term comes from the compiler community, but our use here reflects a spectrum of representation choices that overlaps on the simpler end of the representation spectrum with that of traditional compilers.

3.4.1 Flat Intermediate Representation Only in the simplest cases can a program be executed directly as it is parsed. These simple languages can be likened to a four-function calculator, doing calculations one step at a time. *How should more complex languages be represented internally?* Most languages are complex and need a way to build an intermediate structure which is subsequently used to produce another source program or executed. Such a form is called an intermediate representation (IR).

Therefore, *use an* Flat Intermediate Representation *(Flat IR) that is tailored to the kind of* Record Language *that is being recognized and use the appropriate method for building it.*

These Flat IR techniques are commonly used with scripting languages such as AWK or Perl. These languages are interpreted and loosely typed, traits which are well suited to text-to-text translators. In addition to regular expression matching and string manipulation, both AWK and Perl have associate arrays—Dictionaries to the Smalltalk and Java programmer—which map keys to values. Our examples will be in AWK. AWK in its simplest mode of operation reads standard input one line at a time, tokenizes the current line, and places the values of each column into the variables $1, $2, etc. Then, the user supplied sequence of pattern/action pairs is evaluated, with the action fired when the pattern matches the current input line. A pattern/action pair with no pattern is fired for every input line.

For Key-value Pairs, build a dictionary of string to string mappings. This separates the parsing from use, but has the disadvantage of moving error-checking away from immediate notification.

The following AWK code:

```
    { keyValueMap[$1] = $2; cnt++ } /* match every line */
END {
        for (i in keyValueMap) { print "char *" i ";" }
        for (i in keyValueMap) {
            print i "=" "\"" keyValueMap[i] "\";"
}
    }
```

When given the input:

```
a b
c d
```

Produces the output:

```
char *a;
char *c;
a="b";
c="d";
```

For Delimiter-separated Values, each record is stored as an object with an array of Strings. The client does the conversion to the needed structure. In an object-oriented implementation language, typically a constructor will take an argument of either an array of strings or a record structure holding the delimiter separated values. In a language like AWK or FORTRAN, parallel arrays for each column are used.

The following AWK code:

```
BEGIN { cnt = 0 }
    { firstName[cnt] = $1; lastName[cnt] = $2; cnt ++ }
END { for (i = 0; i < cnt; i++) {
          print lastName[i] ", " firstName[i]
      }
    }
```

When given the input:

```
Joel Jones
Trevor Jay
Crutcher Dunnavant
```

Produces the output:

```
Jones, Joel
Jay, Trevor
Dunnavant, Crutcher
```

For Stanza Formatted Records, processing can be done in one of two ways. [1]

The first approach is to process each record one at a time. This has to be done piece-meal, as more than one line must be read for each record. Except for the first record, the detection of the start of a new record triggers the generation of a client record from gathered information. The following AWK code illustrates.

[1] Neither of the following examples are idiomatic AWK usage, as line separators can be set to be field separators q.v. [30]

```
$1 == "lastName:" { lastName = substr($0, index($0, $2)) }
$1 == "firstName:" { firstName = substr($0, index($0, $2)) }
/^$/ { emitRecord() } # empty line
function emitRecord() {
    if (lastName != "") {
        printf("%s", lastName)
        if (firstName != "") printf(", ");
    }
    printf("%s\n", firstName)
    lastName = ""; firstName = "";
}
END { emitRecord() }
```

The other approach is to collect all of the information and create the records after the entire file has been processed. In a scripting language, use a counter to generate a record identifier and collection of associative arrays, one per attribute, and use the record identifier as a key and the attribute value as the indexed value. This technique is useful if duplicates need to be removed or if records are filtered based upon aggregate information. The following AWK code illustrates.

```
BEGIN { id = 0 }
$1 == "lastName:" { lastName[id] = substr($0, index($0, $2)) }
$1 == "firstName:" { firstName[id] = substr($0, index($0, $2)) }
/^$/ { id++ } # empty line
END { for (i = 0; i <= id; i++) {
          if (lastName[i] != "") {
              printf("%s", lastName[i])
              if (firstName[i] != "") printf(", ");
          }
          printf("%s\n", firstName[i])
      }
    }
```

3.4.2 AST You are implementing a language, and the language is sufficiently complex that Immediate Execution or Flat Intermediate Representation is not desirable. *How do you represent the essential characteristics of the structure of the input and avoid making errors in constructing this representation?*

Parsing a non-trivial language usually involves the implicit or explicit creation and traversal of a tree structure. This tree is a consequence of the context-free language that is being parsed. However, the tree induced by parsing contains extraneous information and may have a structure that is not convenient to deal with.

Therefore, implement an Abstract Syntax Tree *(AST) using implementation language specific idioms.* An Abstract Syntax Tree (AST) captures

the essential structure of the input in a tree form, while omitting unnecessary syntactic details. ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. ASTs also omit tree nodes that represent unary productions in the grammar. Such information is directly represented in ASTs by the structure of the tree. An AST is a tree that is specific to the language being represented, rather than a generic tree structure consisting of information about the node represented as a reference to "Object" and a collection of "Tree" nodes to represent the children. The use of a specialized representation allows the implementation system to detect errors at translator build time, through the use of type-checking on the elements of the AST.

When designing the nodes of the tree, a common design choice is determining the granularity of the representation of the AST. That is, whether all constructs of the source language are represented as a different type of AST node or whether some constructs of the source language are represented with a common type of AST node and differentiated using a value. One example of choosing the granularity of representation is determining how to represent binary arithmetic operations. One choice is to have a single binary operation tree node, which has as one of its attributes the operation, e.g. "+". The other choice is to have a tree node for every binary operation. In an object-oriented language, this would results in classes like: AddBinary, SubtractBinary, MultiplyBinary, etc. with an abstract super class of Binary. The second form is preferred if there will be behavior associated with the tree nodes. More information on how to implement ASTs can be found in [22].

ASTs can be implemented in one of the following ways—Hand-written AST, Generated AST, or Commodity AST.

3.4.2.1 Hand-written AST You have decided that you need to implement an Abstract Syntax Tree (AST) because the translation or execution process was too complex to perform directly from the input source.

How do you implement an AST? There are several factors to consider. First, if the parser implementation is Generated Parser, then the parser generator may have mechanisms for generating an AST. In that case, consider Generated AST. Second, if your implementation language is supported by an AST generator tool, then the code for the AST can be generated. Again, consider Generated AST. Third, if you intend to do fairly commonplace transformations on the tree, then a existing tree rewrite system like XSLT may be useful. In that case, consider Commodity AST. If none of these apply, then another approach to implementing the AST must be taken.

Therefore, *produce the* AST *implementation directly by writing the code that implements the desired structure and function.* Use appropriate idioms for your implementation language in implementing the AST. For imperative languages, such as Pascal or C, use a variant record structure with a variant for each AST node type. In ML or Haskell, use a `datatype` declaration. In an object-oriented

language, use a class hierarchy with a class for each AST node type and an abstract base class for representing the general AST node.

To produce the AST during parsing, the AST is built by having nodes added to the tree when a complete production is recognized.

As an illustration of how to implement a HAND-WRITTEN AST in an imperative language like C, we use the example from the discussion of INTERPRETER from [14]. Figure 1 is the `.h` file and figure 2 is the `.c` file.

```
typedef struct booleanExp *booleanExp_ty;
typedef char* identifier;
typedef int boolean;

enum booleanExp_type {
        VARIABLE, CONSTANT, OREXP, ANDEXP, NOTEXP
} ;

struct booleanExp {
        enum booleanExp_type kind;
        union {
            struct { identifier id; } variable;
            struct { boolean b; } constant;
            struct {
                booleanExp_ty left;
                booleanExp_ty right;
            } orExp;
            struct {
                booleanExp_ty left;
                booleanExp_ty right;
            } andExp;
            struct { booleanExp_ty exp; } notExp;
        } u;
};
```

Fig. 1. HAND-WRITTEN AST Example`.h` File

3.4.2.2 Generated AST You have decided that you need to implement an ABSTRACT SYNTAX TREE (AST) because the translation or execution process was too complex to perform directly from the input source. Also, either your parser generator supports the generation of ASTs or your implementation language has tools for generating ASTs.

How do you implement an AST? As in any project, you want to reduce the implementation effort for your AST. The implementation of an AST is mostly rote—once the structure of the desired AST is determined, the implementation code is straightforward. Most node types of the AST have an invariant number of children, with the AST nodes represented as record structures and the children

```
#include "booleanExp.h"
booleanExp_ty
mkVariable(identifier id) {
        booleanExp_ty p;

        p = (booleanExp_ty) malloc(sizeof(*p));
        p->kind = VARIABLE;
        p->u.variable.id = id;
        return p;
}

booleanExp_ty
mkConstant(boolean b) { /* ... */ }
/* ... */
```

Fig. 2. HAND-WRITTEN AST Example.c File

are represented as references to the appropriate node type. Some nodes will have a variant number of children, e.g. argument lists to method calls. These are represented using collections of references. Given the rote nature of this process, you want to proceed through its implementation as quickly as possible.

Therefore, *use a* GENERATED AST. There are two kinds of GENERATED ASTs. The first takes a specification of the AST and generates the necessary code for implementing the AST. The second generates an AST from its implicit specification in a grammar specification.

The input language for the first kind of AST generator will typically contain the name of the generic node's type, the name of all specific node's types, and member names and types for the specific nodes. One such tool is Zephyr, the generator for Abstract Definition Language (ASDL) [40] which includes C as one of its output languages. It is also not hard to build a simple version of such a tool using a scripting language such as AWK or Perl. In addition to generating the data type declarations, an AST code generator should also create "constructor" functions which take as arguments the children of the node and return initialized instances of the specific nodes of the AST.

As an example of what the specification for an AST is like, we take the example from the discussion of INTERPRETER from [14]. We use the Zephyr ASDL format, an example of which is seen in Figure 3.

3.4.2.3 Commodity AST You have decided that you need to implement an ABSTRACT SYNTAX TREE (AST) because the translation or execution process was too complex to perform directly from the input source. *If the source language is in XML or easily lexically transformed into XML, how do you implement an AST?*

If the source language is XML or in another format for which there is a pre-existing tree format, then the desired AST structure is likely very similar to the input structure. There are many network protocols, such as SOAP, which are

```
booleanexp = Variable(identifer id)
           | Constant(boolean b)
           | OrExp(booleanexp left, booleanexp right)
           | AndExp(booleanexp left, booleanexp right)
           | NotExp(booleanexp exp)
```

Fig. 3. GENERATED AST Example

based upon XML. Writing an AST implementation in such a situation is a waste of effort, as there are already parsers for XML for various environments.

Therefore, *use a* COMMODITY AST *implementation, most commonly one supporting XML.* Use a library for implementing tree structures. Such a library will support node creation, iteration over children, and access to a dictionary like structure to store node attributes with name-value access patterns. The library may provide memory management, validation, pickling, pretty-printing, etc.

3.5 Transformation Techniques

Language implementation always involves some form of transformation technique. However, for many DSLs, the transformation techniques are somewhat ad hoc and tightly related to the semantics of the input language. Many language implementations make the use of similar techniques that should be examined for their applicability. In this section we present two of the most common categories of transformation styles.

3.5.1 Lexical Transformation Some transforms on a language, even useful ones, do not require deep or even shallow semantic understanding. Macro expansion, many forms of syntactic sugar and sometimes even rewriting of HIGH-ORDER FEATUREs can often be accomplished with simpler non-AST based transforms.

Many languages require some degree of transformation or rewrite. In cases where deeper understanding such as that represented by AST's is not needed another method is desirable to avoid unneeded work and complexity.

Commonly LEXICAL TRANSFORMATION is handled by some sort of regular expression engine. This might be in the form of some mechanism such as a PERL script run before other stages of language processing.

LEXICAL TRANSFORMATION is not necessarily a means to an end itself. Many transforms simply make matters easier for other portions of the processing toolchain. While it performs other functions the C pre-processor is a good example of such a situation.

Use a LEXICAL TRANSFORMATION *based purely on less than shallow semantic understanding such as string manipulation through regular expressions.*

While the technique should only be stretched so far, it should be kept in mind that if a LEXICAL TRANSFORMATION is being applied in a somewhat ad-hoc manner, as for example through a PERL script; then slightly more advanced

features requiring mild understanding, such as brace counting, may be easy to implement.

If more advanced transformation is required a full AST as in a Tree Transformation may be required.

3.5.2 Tree Transformation Quite often the best way to view language processing, or just a portion of processing, is as a transformation. This transformation may be simple, requiring little semantic knowledge as is the case with a Lexical Transformation or it may be more complex.

Some language processing is best achieved as a transformation and some kinds of transformation require a reconfiguration of the structure of the language itself.

In order to restructure a language the structure itself must first be captured, this is often done using an AST. Once some semantic statements are in AST form then transformations on that AST become an option.

Use Tree Transformation. *Read your input language into an AST and then restructure it using some form of transformation engine.*

While Tree Transformation may become a rather complicated enterprise as when it is used for tasks such as optimization, it has simpler more approachable forms as well. There are generic tree transformation engines available such as the XSLT language available for transforming XML data structures. This level of power alone makes it possible to perform a great number of transformation task such as expansion, collapsing, reordering, and a great deal of restructuring. Tree Transformation forms the core of traditional non-optimizing compilers.

By itself or when used in combination with Lexical Transformation, Tree Transformation can be used as a step in a more complex processing scheme or may be all that is necessary to perform Source Output or Embedded Language. Tree Transformation is *real* language processing and can accomplish most of what needs to be done with an input language.

3.6 Execution Techniques

In the continuum of execution techniques, measured on the extent of semantic analysis, we will find that Interpreters lie roughly in the middle, Virtual Machines lie towards the side of low semantic analysis, and Semantic Evaluators lie towards the side of high semantic analysis. These evaluators are the portion of a language processing system which *understands* the meaning of a language, and provide its interpretation. The other execution techniques are independent of a particular level of semantic analysis.

3.6.1 Record Consumer For many kinds of input, especially that which might be found in a Record Language, even the most simple of "processing" methods is overkill. You are simply looking for a way to shovel data into an environment. An AST would clearly be overkill in this case, and in fact so would most primitive forms of parsing.

A Record Language *has a unique need for simplicity in processing in order to preserve the labor savings its use is intended to win.*

Imagine a bash script that removes batches of users from a system. It's input might be a file containing a list of said users, separated by newlines. All the script really does is consume a line at a time taking the string, the username, and rewriting it into a command that it then runs.

Such a script is a good example of a Record Consumer. Its input file is clearly a Record Language based on a Delimiter-separated Values list, it consumes one record at a time and it does so with very little processing.

Therefore, use Record Consumer *when using a* Record Lanugage *and actions are performed immediately, one record at a time.*

Many Record Consumers take an immediate action as they consume a record. They use it as a argument in a system command or a function call, they initialize a variable to that value, or they might create a data object based on the record. If anything but the most primitive of validity checking need be done perhaps you should not use a Record Consumer.

Record Consumers often serve to perform tasks such as reading in configuration records, but they may also be used as part of a more complex processing environment. Flat Intermediate Representation may be used if a minimal amount of aggregate processing is needed.

3.6.2 Immediate Execution You need a processing paradigm underneath a Record Consumer or similar mechanism. Your language is mostly atomic and directive in nature. No further complex processing of the language is required save to carry out a series of desired commands. The language is not forward referenced. Advanced semantic features such as side effects are either not present or constrained such that they permit serial evaluation. A source program in this simplest case can be recognized and executed directly.

Therefore, evaluate the source program directly, without translation to an intermediate form. Immediate Execution is often used in situations such as when records have a direct mapping to function calls within an API or some other mechanism that needs to be "driven". Input is parsed, and as soon as an executable portion is recognized, it is executed. BASIC and dc are prototypical examples of this pattern.

The action code will mostly consist of calls into a Runtime.

3.6.3 Virtual Machine You have an algorithmic language and need to evaluate it. Your desired language has features which are not easily realized by your implementation language. Your desired language has a well defined, fairly invariant processing model. This model is more well defined than that common in an Interpreter. Portability is also an important concern.

Therefore, use a Virtual Machine. A Virtual Machine is a simulated architecture realized within an implementation environment. It is much like an emulator except that the architecture it "emulates" may not exist, or even be possible.

By virtue of providing its own memory and processing environment, the Virtual Machine may be able to provide features not native to the implementation language. The Java VM is able to offer garbage collection to the Java language even though C does not implement this paradigm. Further, as a result of the virtual nature of the VM, environment code inside it is well insulated from the rest of the system. This may be a cognitive interface advantage, or a security and performance advantage as in sandboxes.

Virtual machines have a substantial cost. You should have a great deal of resources to commit to development. Platform portability should be important. VM's are good when complex features are required yet speed of language programs is still important. A Virtual Machine almost by definition is tied to some form of Runtime and may require a Semantic Evaluator, usually in the form of a compiler because VM's are often based on bytecode or some other form of atomic instruction set.

3.6.4 Interpreter The intermediate representation, typically a tree structure, is often sufficient for driving program execution. The most common case is when execution is dominated by time spent in the Runtime, rather than in the direct execution of the input language. For example, database query execution time is dominated by I/O, not language processing.

Therefore, *use an* Interpreter *to evaluate the language.* An interpreter treats the intermediate representation (IR) as the instructions for an execution engine, rather than translating the IR into another language. These instructions are not necessarily linear, but can take other forms. Interpreters are useful when portability is important, the developers do not have knowledge of the processor, or the problem domain of the language is straightforward and does not require a Semantic Evaluator.

This pattern was first described in [14] and elaborated on in [3]. However, in both cases, the definition of Interpreter is narrower than is usually meant in the language implementation community. The emphasis on ASTs in the pattern version slights other techniques, such as linear forms [29] for evaluating a language.

3.6.5 Semantic Evaluator Constraint languages are composed of elements that are not directly mapped to a Von-Neuman architecture. For example, Prolog programs do not explicitly specify an execution order. *How do you provide impetus to constraint languages?*

Processing of **Constraint Languages** requires a Semantic Evaluator, as there are few instructions to drive either an Interpreter, or a Virtual Machine.

Therefore, *construct a* Semantic Evaluator *for your language, and place the code which must understand your language there.* As a result of parsing the input, a Semantic Evaluator builds data structures that represent the program in a form that is closer to a semantics of variable assignment, function calls, and explicit ordering. The function calls will include calls into a Runtime

that contains significant encoding of the semantics of the input language, i.e. translated towards the algorithmic language side of the definitional dichotomy. For example, in Prolog, a function call might be a call to a routine to unify variable assignments in a single term. In SQL, a function call might be made to find all rows of a table with a column matching a given value. These data structures may be evaluated using INTERPRETER or a translation into a more sequential form and evaluated by a VIRTUAL MACHINE. SQL evaluators use the INTERPRETER approach using a query plan, a tree structure that is evaluated bottom up. The VIRTUAL MACHINE approach is the most common execution technique for Prolog. The Warren Abstract Machine is the most used virtual machine model used by Prolog implementors[2].

3.7 Infrastructure

At the lowest level of a language implementation is a support system. For languages that are executed directly, a RUNTIME provides services to the running program. For languages that are translated to another source form, LANGUAGE OUTPUT provides a means of producing human-readable output.

3.7.1 Runtime *If a language is to be executed and there is not a one-to-one relationship from the elements of the input language to the elements of the implementation language, how should the source language elements be executed?* Sometimes, the missing elements form a low-level infrastructure, such as memory allocation handling or stack treatment. Another need is for providing encapsulation of higher-order abstractions such as file I/O or database access.

Therefore, *use a* RUNTIME *environment to support execution.* If possible, compose the RUNTIME from already available components such as garbage collectors and database toolkits. Implementing a RUNTIME can require a great deal of programmer expertise and time but they are one of the only methods for providing certain categories of language features such as very specialized memory management or unusual execution methods.

3.7.2 Language Output Rather than executing an input language, a language processor can produce a translation into another language. In such a situation, *use* LANGUAGE OUTPUT*, producing human-readable text from the language processor.*

LANGUAGE OUTPUT is frequently used in combination with LEXICAL TRANSFORMATION in order to realize EMBEDDED LANGUAGE. Many multi-stage compilers use LANGUAGE OUTPUT several times, some C compilers sometimes go from C (with macros) to C (without macros) to Assembly. In addition, the natural results of some systems, such as those which produce PostScript or PDF documents, are Language based.

If the output language supports some form of comment feature, add a note that the code was generated, and if possible, by what processing system, and from what input files. Also, try to produce language output that is readable by

```
public class Table {

    private String tableName;
    private String[] columnNames;
    private int[][] rows;
    private static Hashtable tables = new Hashtable();

    public static Table byName(String tableName) {

    public Table project(String[] colNames) throws Exception {

    public Table select(String colName, Op op, int value) throws Exception {

    public Table join(String selfColName, Table otherTable, String otherColName)
        throws Exception {

}
```

Fig. 4. RUNTIME Example: Table.java

generating properly formatted code and preserving useful information from the input, such as identifiers.

4 Extended Example

To gain a better understanding of this pattern language, we present here an extended example. This example gives the highlights of the implementation of a small subset of SQL, the relational database query language. This example covers the patterns RUNTIME, HAND-WRITTEN AST, GENERATED PARSER. GENERATED AST, SEMANTIC EVALUATOR, and INTERPRETER. Although we have presented this pattern language in processing order, i.e. from the initial input language to the description of the execution mechanisms, we present our example in the fashion in which most domain-specific languages are developed. We begin with the execution environment and work towards the language specification.

In Figure 4 we see the declarations of a Java class for representing a table in a relational database. In addition to storing the data (`rows`) and metadata (`tables`, `tableName` and `columnNames`), we also support the basic relational operators on a table, `project`, `select`, and `join`. The goal of our effort is to ease the use of these operators by providing a constraint language for accessing these operations. The *project* relational operator selects all the rows of the input table and returns only those columns specified. The *select* operator chooses rows from a table based on satisfying some expression based on the values in a single table, taken a row at a time. The *join* operator merges two tables together when a column from one table matches the value of a column in another table. In our example, we implement only limited forms of these operators. Our *select* operator

```
public abstract class QueryPlan {
    protected QueryPlan qPlan;
    protected Table table = null;

    public abstract Table execute() throws Exception;
```

Fig. 5. HAND-WRITTEN AST Example: QueryPlan.java

```
public class Project extends QueryPlan {
    private String[] columnNames;

    public Table execute() throws Exception {
        Table result = table == null ? qPlan.execute() : table;
        return result.project(columnNames);
    }
```

Fig. 6. HAND-WRITTEN AST Example: Project.java

allows only a single selection comparing a column against a integer constant using the usual set of comparison operators. The *select* operator we implement supports only "natural joins", matching only when the two values are equal, not less-than, etc. Nothing in the instantiation of RUNTIME had dependencies on the language implementation, like parsers, etc. This is frequently the case, as a library will frequently exist before the notion of calling it from a language does.

The next step is to choose a means of representing the combination of operations to be performed as the result of a query. We use a simplified version of a *query plan*, which represents the operations as a tree. The tree is evaluated bottom up. We use the pattern HAND-WRITTEN AST, as there is no directly corresponding input language, and there are only three concrete classes. In Figure 5, we see the definition of the abstract superclass, `QueryPlan`. Every operation in the subclasses operates either on a table contained in the database (the `table` member) or on the result of calling `execute()` on the `qPlan` member.

For all of the concrete subclasses of `QueryPlan`, the constructor captures the data members displayed in Figures 6, 7, and 8. To execute a query such as `SELECT colOne FROM tableOne WHERE colOne > 1`, the corresponding query plan would consist of a tree with an instance of `Project` as the root and an instance of `Select` as its child. Calling `execute()` on the root to get the result makes this an instance of INTERPRETER.

Having worked backwards, we now go to the beginning of the processing pipeline to consider the specification of the input language. To keep the example simple, the input language will consist of only enough of SQL to allow for the specification of query that involves either a select on a single column against a constant or a natural join between two tables, and a general project. In Figure 9 we see the specification of a lexer, parser, and AST using the language for the tool SLY. [42] SLY takes the specification in file `SQL.sly` and generates the

```
public class Select extends QueryPlan {
    private String columnName;
    private Op op;
    private int value;

    public Table execute() throws Exception {
        Table result = table == null ? qPlan.execute() : table;
        return result.select(columnName, op, value);
    }
```

Fig. 7. HAND-WRITTEN AST Example: Select.java

```
public class Join extends QueryPlan {
    private String selfColName;
    private Table otherTable;
    private String otherColName;

    public Table execute() throws Exception {
        Table result = table == null ? qPlan.execute() : table;
        return result.join(selfColName, otherTable, otherColName);
    }
```

Fig. 8. HAND-WRITTEN AST Example: Join.java

lexer source for the tool JLex, the parser source for the tool Java Cup, and generates the Java source code for representing an AST. This is an instance of the application of GENERATED PARSER and GENERATED AST. In addition to combining the specification of the lexer and parser into a single file, SLY also has other features to note. The specification of the lexer is simplified to support the specific case of being used in conjunction with a parser generator and building ASTs. This is accomplished by providing support for alphabetic keywords, non-varying symbols, and varying value tokens, `keyword`, `symbolic`, and `lexer` and `ignore`, respectively. Parser rules are specified using `::=`, with the right-hand side consisting of sequences of terminals and non-terminals. The left-hand side serves two purposes—to define the "name" of the production and to optionally specify a superclass, e.g. `IntFilterExpression` specifies the name and `super WhereExpression` specifies the superclass of the AST node used to represent `IntFilterExpression`. The parser supports the direct specification of optional elements on the right-hand side of a production, e.g., `ColumnRecur` in the definition of `ColumnList`.

The most novel aspect of SLY is its support for automatic construction of abstract syntax trees, an instantiation of GENERATED AST. The method used is simple—a class is generated for every grammar production with a data member defined for every symbol on the right-hand side of the production. The names of the data members are the all lower-case letter version of the symbol name.

```
keyword ::= WHERE SELECT FROM ;

symbolic ::=
  GE ">="  EQ "="  LE "<="  GT ">"   NE "!=" LT "<"  Comma "," ;

lexer INT ::= {:-?[0-9]:};
lexer ID ::= {:[a-zA-Z]+:};
ignore ::= {:[ \t\n]:};
lexer error ::= {:.:};

/* extremely simplified SQL grammar */
SelectStatement super SQLAST ::= T_Keyword_SELECT Columns
                              T_Keyword_FROM Tables
                              T_Keyword_WHERE WhereExpression ;
    {:  public void accept(SQLASTvisitor visitor) { visitor.visit(this); } :}

abstract Columns super SQLAST
    ColumnList super Columns ::= [ ColumnRecur ] Column ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
    ColumnRecur super Columns ::= ColumnList T_Symbolic_Comma ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
;

Column ::= T_ID ;

abstract Tables super SQLAST
    TableList super Tables ::= [ TableRecur ] Table;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
    TableRecur super Tables ::= TableList T_Symbolic_Comma ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
;

Table ::= T_ID ;

abstract WhereExpression super SQLAST
    IntFilterExpression super WhereExpression ::= T_INT Binop Column ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
    ColumnFilterExpression super WhereExpression ::= Column Binop T_INT ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
    JoinExpression super WhereExpression ::= Column T_Symbolic_EQ RightColumn ;
        {:  public void accept(SQLASTvisitor visitor) {
                visitor.visit(this); } :}
;
```

Fig. 9. Generated Parser and Generated AST Example: SQL.sly

```
abstract public class SQLAST {
    public abstract void accept(SQLASTvisitor visitor);
}
```

Fig. 10. GENERATED AST Example: SQLAST.java

Productions are grouped together for the purposes of grammar definition using the `abstract` keyword. Typically, productions grouped together also form the concrete subclasses of the class created for the containing abstract production, e.g. `ColumnFilterExpression` is subclass of `WhereExpression`. To support the next phase of the implementation, an additional source file, `SQLAST.java` (q.v. Figure 10) is used to define the base of the class hierarchy of most of the classes in the generated AST, e.g. `WhereExpression`. Classes used as terminal nodes are not part of this hierarchy, as it simplifies the implementation in the next phase. This is slightly unusual, as all of the nodes of the AST typically form a single hierarchy in an AST implementation.

The next phase of implementation uses SEMANTIC EVALUATION and VISITOR[14] to generate the query plan. In Figures 11, and 12, we see the implementation of those patterns. The query plan is built by visiting the nodes of the `SQLAST`, and generating instances of the appropriate subclasses of `QueryPlan`. Of note is the transformation of the tree structures, which mirror the parsing process, into list structures needed by the `Table` runtime, e.g. `visit(ColumnList)` and `visit(ColumnRecur)` generate into `Vector colVec`, which is converted into an array by `Vector.toArray()` in `visit(SelectStatement)`.

In Figure 13, we close the remaining gap of an end-to-end processor by instantiating INTERPRETER. The steps followed are typical of end-to-end interpreters—initialize the runtime (`buildDatabase`), parse the input (`planFor`), execute the IR (`qPlan.execute()`), and produce the output (`Table.toString()`, which is implicit in the method call `System.out.println()`).

5 Related Work

This pattern language barely touches upon the vast literature related to implementing languages. There are several areas that this pattern language covers only in passing, if it all. This list of suggested readings should aid the beginning language implementor in solving design problems not covered by this pattern language.

Lexing and Parsing: Aho et al[1] cover the basic theory of lexing and parsing and give algorithms for building lexer and parser generators. Grune and Jacobs[17] give an extensive coverage of grammars. Watt and Brown [41] give a good exposition of hand-written parsers and object-oriented idioms for lexing, parsing, and AST representation.

Optimization: Aho et al[1] cover the rudiments of optimization for translation to machine code. Muchnick[33] gives a more advanced coverage. The spe-

cial issue of SIGPLAN Notices[32] is a selection of most influential papers from 1979 to 1999 presented at the "Programming Language Design and Implementation" (PLDI) conference.

Logic and Functional Languages Kamin[25] gives several educational implementations of functional languages and Prolog. Hassan Aït-Kaci[2] has a book length introduction to the Warren Abstract Machine (WAM) implementation of Prolog. Queinnec's book[35] is a step-wise implementation of LISP. Simon Peyton Jones'[24] book is still a major contribution to the implementation of lazy functional languages. Appel[4] discusses the use of continuations as a way of structuring the runtime of functional languages.

Object-oriented Languages The primary implementation concern for object-oriented languages is how to implement method lookup. The implementation choice is driven by whether or not the the language is statically typed. For coverage of implementation of dynamically typed object-oriented languages, see Deutsch and Schiffman[11] and Ungar[39] for Smalltalk and Holze[18] for Self. For statically typed OO languages, see Sun's HotSpot[38].

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley, Reading, Massachusetts, 1986.
2. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. published by the author, 1999.
3. Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.
4. Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
5. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
6. Jon L. Bentley, Lynn W. Jelinski, and Brian W. Kernighan. CHEM — a program for phototypesetting chemical structure diagrams. *Computers and Chemistry*, 11(4):281–297, 1987.
7. Elliot Berk. JLex: A lexical analyzer generator for Java. http://www.cs.princeton.edu/~appel/modern/Java/JLex/, 2000.
8. Berkeley Software Distribution (BSD) License. *Java Compiler Compiler^{TM} (JavaCC^{TM}) - The Java Parser Generator - Home Page*. https://javacc.dev.java.net.
9. Compiler Resources Inc's. Yacc++® and the language objects library. http://world.std.com/ compres/, 2003.
10. C. J. Date and Hugh Darwen. *A guide to the SQL standard: a user's guide to the standard database language SQL*. Addison-Wesley, Reading, MA, USA, fourth edition, 1997.
11. Peter Deutsch and Alan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, ACM, January 1984.
12. Crutcher Dunnavant, Trevor Jay, Joel Jones, Katrina Roan, Charles Ward, Nathan Wiegand, and Derek Woodham. Commodity transformation for dsls: Lpt with lex, yacc, and xslt. http://press.samedi-

studios.com/publications/2004/ dunnavant2004commodity_transformation/ dunnavant2004commodity_transformation.pdf.
13. Jeffrey E. F. Friedl. *Regular Expressions*. O'Reilly, 1998. ISBN: 3–930–67362–2.
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.
15. Gnome. The xml c parser and toolkit of gnome. http://www.xmlsoft.org/.
16. Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992.
17. Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Computers and their Applications. Ellis Horwood, Chichester, UK, 1990.
18. Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.
19. Scott Hudson. Java cup LALR parser generator for Java. http://www.cs.princeton.edu/~appel/modern/Java/CUP/, 1999.
20. TXL Software Research Inc. The TXL transformation system. http://www.txl.ca/, 2001.
21. Joel Jones. Tabular code generation: Write once, generate many. Pattern Languages of Program Design, 2002.
22. Joel Jones. Abstract syntax tree implementation idioms. Pattern Languages of Program Design, 2003.
23. Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.
24. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.
25. Samuel N. Kamin. *Programming languages: an interpreter-based approach*. Addison-Wesley Longman Publishing Co., Inc., 1990.
26. Brian W. Kernighan. PIC — A graphics language for typesetting user manual+. March 17 2001.
27. Brian W. Kernighan and Jon L. Bentley. Grap — A language for typesetting graphs tutorial and user manual. May 11 1991.
28. Brian W. Kernighan and Lorinda L. Cherry. Typesetting mathematics — user's guide (second edition). October 31 1997.
29. Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1984.
30. Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, USA, 1999.
31. John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 2nd edition, 1992.
32. Kathryn S. McKinley, editor. *Best of PLDI 1979-1999*, volume 39. ACM Press, April 2004.
33. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
34. Apache Organization. Xerces java parser readme. http://xml.apache.org/xerces-j/.
35. Christian Queinnec. *LISP in small pieces*. Cambridge University Press, Cambridge, UK, 1996.
36. Eric Steven Raymond. *The Art of UNIX Programming*. Addison-Wesley, Reading, MA, USA, 2004.

37. Diomidis Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99, February 2001.
38. Sun Microsystems Incorporated. The Java HotSpot^{TM} performance engine architecture: A white paper about Sun's second generation performance technology, April 1999.
39. David Michael Ungar. The design and evaluation of A high performance smalltalk system. Technical Report CSD-86-287, University of California, Berkeley, February 1986.
40. Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In USENIX, editor, *Proceedings of the Conference on Domain-Specific Languages, October 15–17, 1997, Santa Barbara, California*, pages ??–??, Berkeley, CA, USA, 1997. USENIX.
41. David Anthony Watt and Deryck F. Brown. *Programming Language Processors in Java : Compilers and Interpreters*. Harlow, Inglaterra : Prentice-Hall, 2000.
42. Don Yessick and Joel Jones. Reinventing the wheel or not yet another compiler compiler compiler. In *Southeast ACM Conference*, 2002.

```
public class QueryPlanVisitor extends SQLASTvisitor {
    private Hashtable tableDictionary;
    private Hashtable col2Table;
    public QueryPlan qPlan;
    private Vector colVec = null;


    public void visit(SelectStatement selectStatement) {
        tableDictionary = new Hashtable();
        selectStatement.tables.accept(this);
        makeCol2Table();

        selectStatement.whereexpression.accept(this);

        colVec = new Vector();
        selectStatement.columns.accept(this);
        String[] colNames = (String[]) colVec.toArray((Object[]) new String[0]);
        qPlan = new Project(colNames, qPlan);
    }


    public void visit(ColumnList columnList) {
        visit(columnList.columnrecur);
        colVec.add(columnList.column.toString());
    }


    public void visit(ColumnRecur columnRecur) {
        if (columnRecur.isNullObject)
            return;
        visit(columnRecur.columnlist);
    }


    public void visit(TableList tableList) {
        visit(tableList.tablerecur);
        String tableName = tableList.table.toString();
        tableDictionary.put(tableName, sql.Table.byName(tableName));
    }


   public void visit(TableRecur tableRecur) {
        if (tableRecur.isNullObject)
            return;
        visit(tableRecur.tablelist);
    }
```

Fig. 11. Semantic Evaluation Example: QueryPlanVisitor.java

```
    private void makeCol2Table() {
        col2Table = new Hashtable();
        Enumeration e = tableDictionary.elements();
        while (e.hasMoreElements()) {
            Table table = (Table) e.nextElement();
            String[] colNames = table.getColumnNames();
            for (int i = 0; i < colNames.length; i++) {
                col2Table.put(colNames[i], table);
            }
        }
    }


   public void visit(ColumnFilterExpression exp) {
        Binop binOp = exp.binop;
        Table table = (Table) col2Table.get(exp.column.toString());
        String colName = exp.column.toString();
        String valAsString = exp.t_int.toString();
        Op op = Op.opFor(binOp.toString());
        int value = Integer.parseInt(valAsString);
        qPlan = new Select(colName, op, value, table);
    }


    public void visit(IntFilterExpression exp) {
        Binop binOp = exp.binop;
        Table table = (Table) col2Table.get(exp.column.toString());
        String colName = exp.column.toString();
        String valAsString = exp.t_int.toString();
        Op op = Op.opFor(binOp.toString());
        int value = Integer.parseInt(valAsString);
        qPlan = new Select(colName, op.invert(), value, table);
    }


    public void visit(JoinExpression joinExpression) {
        String selfColName = joinExpression.column.toString();
        String otherColName = joinExpression.rightcolumn.toString();
        Table otherTable = (Table) col2Table.get(otherColName);
        Table selfTable = (Table) col2Table.get(selfColName);
        qPlan = new Join(selfTable, selfColName, otherTable, otherColName);
    }

}
```

Fig. 12. Semantic Evaluation Example (cont.): QueryPlanVisitor.java

```
public class SQLInterpreter {
    public static SelectStatement selectStatement;

    public static void main(String[] args) {
        String query = parseArgs(args);
        buildDatabase();
        QueryPlan qPlan = planFor(query);
        Table result = null;
        try {
            result = qPlan.execute();
        } catch (Exception e) {
            System.err.println("Problem with query: " + query);
            e.printStackTrace();
            System.exit(1);
        }
        System.out.println(result);
    }

    private static QueryPlan planFor(String query) {
        StringReader sbf = new StringReader(query);
        Yylex lexer = new Yylex(sbf);

        parser myParser = new parser(lexer);
        try {
            myParser.parse();
        } catch (Exception ex) {
            System.err.println("syntax error: " + query);
            ex.printStackTrace();
        }

        QueryPlanVisitor visitor = new QueryPlanVisitor();
        selectStatement.accept(visitor);
        return visitor.qPlan;
    }
}
```

Fig. 13. INTERPRETER Example: SQLInterpreter.java