A Pattern Language For Parsing

Yun Mai and Michel de Champlain
Department of Electrical and Computer Engineering
Concordia University
{*y_mai, michel*}*@ece.concordia.ca*

Abstract

Parsing is the core of the front end of a compiler. The predictive recursive-descent parsing technology is most widely used in a traditional compiler design. It is straightforward and easy to implement. But since predictive recursive-descent parsing degrades into a structural programming, it results in a parser that is very hard to change, extend and maintain.

A pattern language is a set of related patterns that solve a common problem in a problem domain. This paper presents a pattern language for developing a framework for parsing in object-oriented compiler design based on the principle of the predictive recursive-descent parsing technology. It describes four patterns that address three design aspects in developing an object-oriented parser. Two alternative patterns are presented to provide alternative solutions to solve the recursion problem in the object-oriented software design. One is based on the Builder design pattern, and the other is based on the meta-programming technology. The parsers developed from this pattern language are easy to implement, easy to extend, and easy to maintain. This pattern language is intended to express a flexible and extensible design for parsing that can accommodate variations to its most extent.

Keywords: Parsing, Compiler, Framework, Design Pattern, Pattern Language, Object-Oriented Design, Reflection, Meta-programming.

1 Introduction

As the use of pattern has injected insight in the analysis of a problem and its solutions, pattern is increasingly important in software design and presentation. A pattern language is a set of related patterns that solve a common problem in a problem domain. It is particular effective at addressing certain recurring problems.

The syntactic analyzer, or the parser, is the core of the front end of the compiler. Its main task is to analyze the program structure and its components [4]. In general, the design of a parser is changing due to the changing of the target language's definition. However, for various compiled languages, all parsing processes share the major commonalty, that is, they follow the same operation pattern.

This paper presents a pattern language for developing a framework for parsing in object-oriented compiler design based on the principle of the predictive recursive-descent parsing technology. It contains four patterns, each is described in a pattern style, where its context, problem, forces, solution, etc, are discussed. The target audience is the framework designer who intends to develop an extensible architecture for parsing or the application developer who needs to better understand the framework in order to customize it for a specific application.

This pattern language contains four different patterns to address three aspects of a framework design for the syntactic analysis in a compiler. These patterns are:

- An analysis pattern: Parser Structure, which addresses the architectural aspect of a parser.
- A structural pattern: Language Structure, which addresses the static representation of the target language.
- Two creational patterns: ParserBuilder and MetaParser, which address the dynamic aspects of the parsing process.

Table 1 is the problem/solution summaries for the patterns presented in the paper. It can be used as a guidance and quick reference to the use of the patterns.

Table 1: Problem/Solution Summaries

Problem	Solution	Pattern Name
How to define an extensible architecture to maximize accommodation of various hot spots for the design of a parser?	Separate grammar rules from the language structure.	Parser Structure
How to represent the language structure to anticipate the changing formats of the target languages?	Organize the language structure with the COMPOSITE design pattern.	Language Structure
How to assemble the loose coupling components in the parser, while at the same time, allow it to be easily extended without modifying the existing code?	Define a common parsing interface with a hook method and let a concrete class implement this hook method and wrap the parsing process for the corresponding target language.	ParserBuilder
How to encapsulate the application logic and build a self-manageable and intelligent parsing processing mechanism?	Define the base-level for the application logic and the meta-level to reflect the base-level and control the parsing process.	MetaParser

2 Parser Structure

Context

You have decided to develop a framework for syntactic analysis.

Problem

How to define an extensible architecture to maximize accommodation of various hot spots?

Forces

- To anticipate the unanticipate is hard. The definition of the target language is vague when the framework is building.
- The grammar rules and the elements of the language structure are embedded in the language definition, which implies the parsing process. Any changes of the grammar rules or the language structure will cause the parsing process to change accordingly.
- A structure is easy to maintain if the code that is frequently changed is separated from that is not.
- The language definition contains so much information that it is too complex to handle. A number of simple problems are easy to solve than a complex one.
- To mix the processing of an object structure with its representation will make a system hard to understand and maintain.
- The user needs not understand the implementation details of a parser. A simple interface is always preferable than a complex one because the complexity of a system is hidden.
- Successful examples often inject insight into the solutions for a recurring problem. Reuse successful experience can minimize the potential risk.

Solution

Apply the ACCOUNTABILITY analysis pattern [3]. Separate grammar rules from the language structure and make the language structure stand alone. A grammar rule encapsulates the application logic and will drive the parsing process. It represents the dynamic aspect of the language definition. A language structure is only a representation of the target language. It represents the static aspect of the language definition.

Define a simple interface, **ParserHandler**, to simplify the use of the system. It provides the least and exact information that the user needs to know.

Structure

Figure 1 shows the structure of the PARSER STRUCTURE.

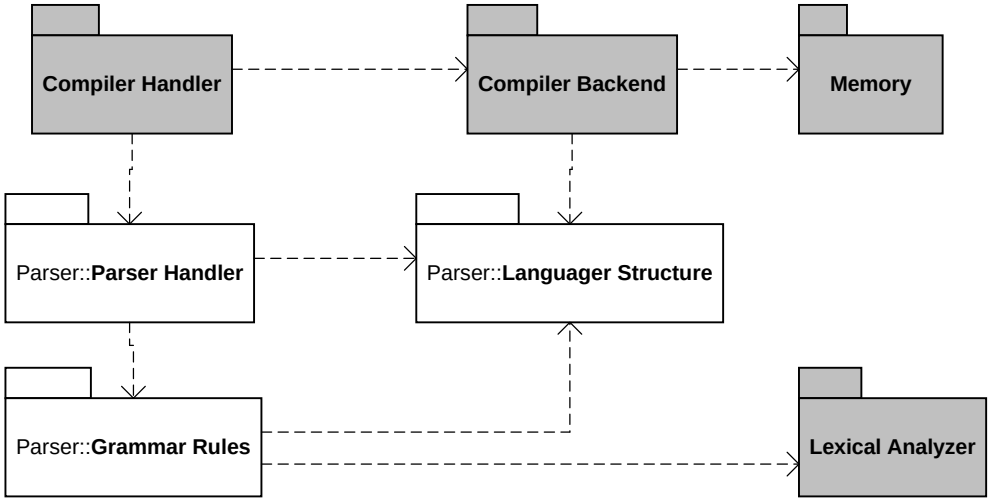

Figure 1: Structure for the Parser Structure

The PARSER STRUCTURE contains three packages: **Parser Handler**, **Grammar Rules**, and **Language Structure** . Note that the packages in gray do not belong to this pattern. But since they are parts of the compiler design, they have direct dependency relationships with the parser.

Participants

- **Parser Handler**
  Declares the interface for the syntactic analysis.
- **Grammar Rules**
  Encapsulates the grammar rules for the target languages and defines the execution sequence of the parsing process.
- **Language Structure**
  Defines the elements that make up of the target language and shows the static view of the relationships among the elements.

Consequences

The separation of the grammar rules from the language structure has the following implicit advantages:

- The static representation of the target language is separated from its potential processing. The grammar rules and the language structure have different roles to play and serve for different purpose. The architecture becomes less coupling and more cohesive.

- Both the grammar rules and the language structure are simple to handle than the one as a whole. The separation helps to reduce the complexity of the system.
- A loose coupling structure is easy to develop, extend, and maintain.

In addition, the **ParserHandler** provides a simple and stable interface to the user. The user is shielded from any potential changes of the grammar rules and the language structure.

Related Patterns

The ACCOUNTABILITY analysis pattern [3] provides similar solution to separate rules from the organization structure.

3 Language Structure

Context

You are defining the language structure and have applied the PARSER STRUCTURE.

Problem

How to represent the language structure to anticipate the changing formats of the target languages?

Forces

- To define a unified language structure for all potential target languages is hard and impossible. A reasonable representation of the language structure is a general abstraction of most frequently used target languages.
- An organized structure is easy to understand and maintain than a number of discrete objects. An organized structure offers a hierarchy that can benefit from some design techniques such as inheritance, which promotes software reuse and extensibility.
- A component of the language structure can be primitive or composite. To differentiate their processing is tedious and error-prone.
- The parsing output is a syntax tree. The representation of the language structure should allow the syntax tree to be easily built and processed.

Solution

Define an interface class **Language** to encapsulate the language abstraction. The language structure is organized using the COMPOSITE design pattern [1]. The syntax tree is represented as the object structure. It is a tree made up of objects of the language structure that are created at run-time.

Structure

Figure 2 shows the structure of the LANGUAGE STRUCTURE.

Participants

- **SyntaxTree**
  A composite object structure that can be used to enumerate its elements.
- **Language**
  An interface for all components of the target language.

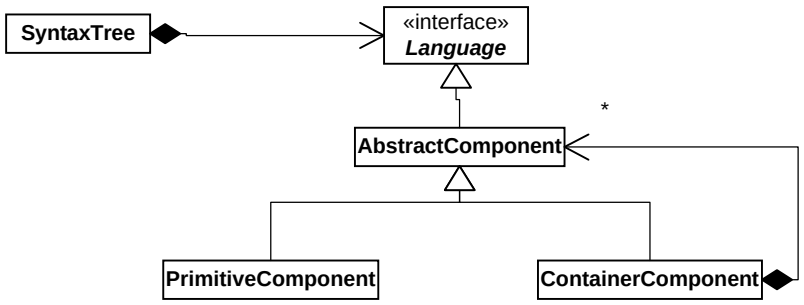

Figure 2: Structure for the Language Structure

- **AbstractComponent**
  A place holder to group the related components into a hierarchy according to their semantics. It lets the hierarchy to be easily extended.
- **PrimitiveComponent**
  Represents an atomic component that does not contain any other components.
- **ContainerComponent**
  Represents a component other than the primitive component. It can contain primitive components and even container components.

Consequences

- The use of the **Language** interface allows different target languages to extend and prevents the client code from changing.
- A composite element can be made up of primitive elements or composite elements. The **Abstract-Component** treats elements uniformly. The language structure is easy to extend through inheritance.
- The syntax tree can be used to easily enumerate its element objects without knowledge of their concrete types.

Related Patterns

The Composite design pattern [1] treats all primitive and composite objects uniformly and define a structure that is easy to extend.

The Reflective Visitor Pattern [8] or other variation of the Visitor patterns [7] can work with Language Structure to perform operations (for example, code generation) on the elements in the Language Structure.

4 ParserBuilder

Context

You are working towards the parsing process and you have applied the Language Structure.

Problem

How to assemble the loose coupling components in the parser, while at the same time, allow it to be easily extended without modifying the existing code?

Forces

- A structure that is hard to or is restricted to modify can be extended through inheritance.
- The rule set encapsulates the application logic. If the rule set is changed or new rules are added, the parsing process needs to be changed accordingly. A changing procedure is hard to maintain and evolve.
- A stable interface can hide the implementation details and allows the implementation to change without changing the client code.
- If the parser is tightly bounded to the rule set, the parser is only meaningful when the corresponding rule set is in use. This makes the system hard to change.

Solution

Define a common parsing interface with a hook method and let a concrete class implement this hook method and wrap the parsing process for the corresponding target language. Apply the BUILDER design pattern [1] to separate the parsing process from the representation of the target language. A hook method *parse* is defined in the interface class and will be overridden in the concrete builder class. Processing of each rule is defined as a method in the concrete builder class.

Structure

Figure 3 shows the structure of the PARSERBUILDER.

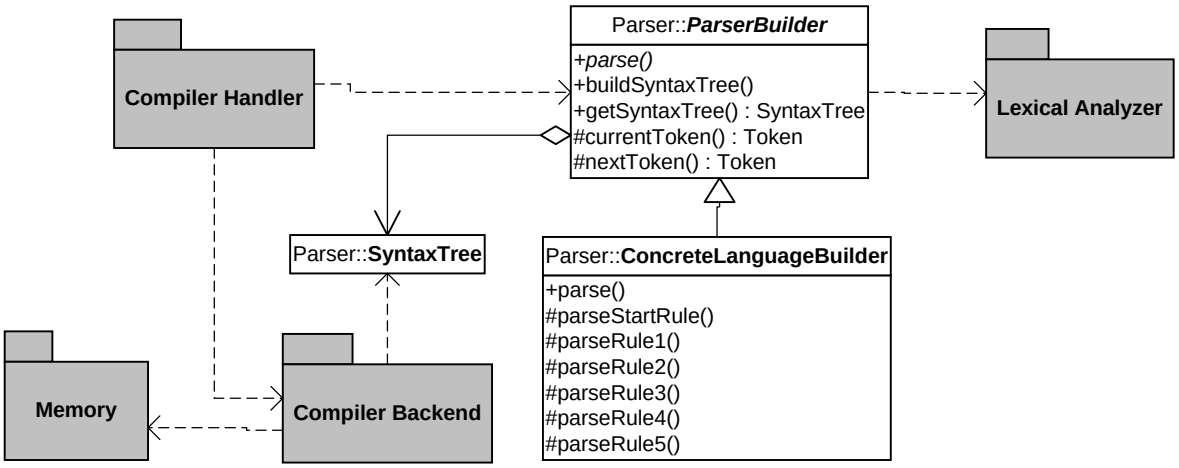

Figure 3: Structure for the ParserBuilder

Participants

- **ParserBuilder**
  A class that plays the role of the **Parser Handler** and defines a hook method *parse* that needs to be overridden by the **ConcreteLanguageBuilder** to perform the actual parsing.
- **ConcreteLanguageBuilder**
  Encapsulates the grammar rules and implements the *parse* method to perform the parsing in a sequence that determined by the rules.
- **SyntaxTree**
  A composite object structure that represents the parsing result and can be used to enumerate its element objects.

Collaborations

Figure 4 shows the sequence diagram for the parsing process in the ParserBuilder.

- An object of the **ConcreteLanguageBuilder** is created for a specific target language.
- The client compilerHandler invokes the *parse* method on an object of the **ConcreteLanguageBuilder** to start the parsing process.
- The parsing method for each grammar rule is recursively invoked. It may need to interact with the **Lexical Analyzer** package to get tokens.
- The parsing result is added to the syntax tree.

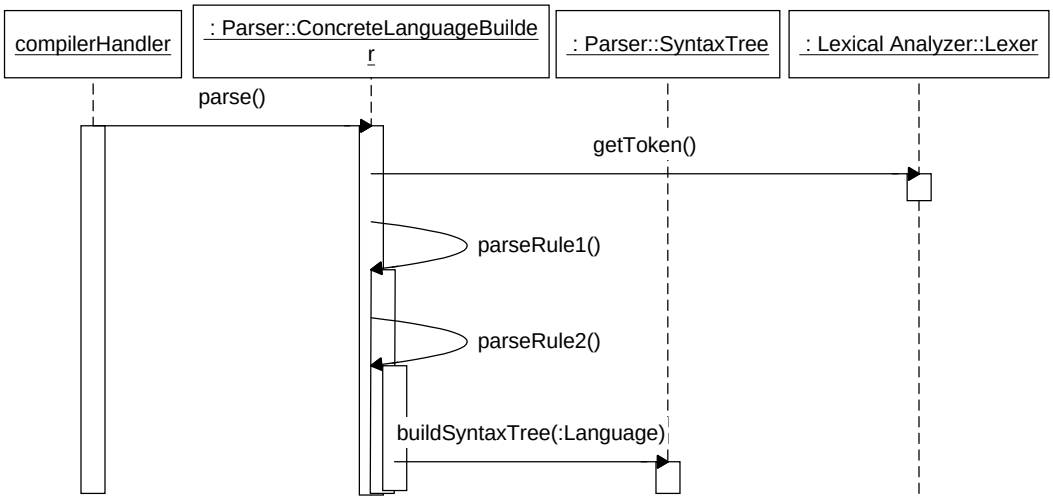

Figure 4: Sequence Diagram for the Parsing Process in the ParserBuilder

Consequences

- Because of the use of the hook method *parse* in the interface, the client is unaware of whatever changes that may be made to the rule set and its implementation.
- A rule is easy to change or add by subclassing the **ConcreteLanguageBuilder**. But removing a rule will cause its corresponding method obsolete and redundant.
- The **ConcreteLanguageBuilder** will become too complex to understand and maintain if the rule set becomes large.
- It is hard to debug and maintain the rule parsing methods due to the recursive invocations among them.

Related Patterns

The Builder design pattern [1] separates the construction process from the object structure so that the same construction process can create different representations of the same object structure.

The MetaPaser pattern that will be presented in Section 5 provides a more flexible structure for parsing.

5 MetaParser

Context

You are working towards the parsing process and you have applied the LANGUAGE STRUCTURE. You want a more flexible parser that supports its own modification at run-time.

Problem

How to encapsulate the application logic and build a self-manageable and intelligent parsing processing mechanism?

Forces

- The application logic encapsulates the changing rule set. A changing component will have limited impact on the rest of the system if it is wrapped into a separated component.
- When the rules are constantly added or are changed often, their relationships become unwieldy. A separate component may be necessary to control the spreading complexity.
- Changing software is error-prone and expensive. A desire result is to let the software actively control its own modification.
- Changes to rules vary according to the target language. A uniform handling mechanism can lead to a system that is easy to understanding and maintain.

Solution

Apply the REFLECTION pattern [2] and define two levels in the system. The base-level contains a set of classes, where each represents a grammar rule. The meta-level handles the complex relationships of the rules that are maintained in a hash table. Reflection technique is used to discover rules at run-time and determines the parsing order. The base-level delegates dynamic dispatch to a meta-level object.

Structure

Figure 5 shows the structure of the MATAPARSER pattern. The gray area represents the meta-level of the system. The packages in gray belong to a compiler design and have direct interaction with the parser.

Participants

- **Parser**
  A class that plays the role of the **Parser Handler**. The client can directly invoke its method *parse* to start the parsing process.
- **Rule**
  Defines a common interface for all grammar rules.
- **ConcreteRule**
  A concrete grammar rule defined in a target language. All grammar rules compose the rule library that can be reused over time.
- **MetaRule**
  Defines the properties of the rule. Each grammar rule class has a corresponding meta-object whose declare type is **MetaRule**.
- **ParsingEnvironment**
  Encapsulates the parsing related information used by the **Rule**. It is managed by the **MetaRule** and shared by all **MetaRule** objects.

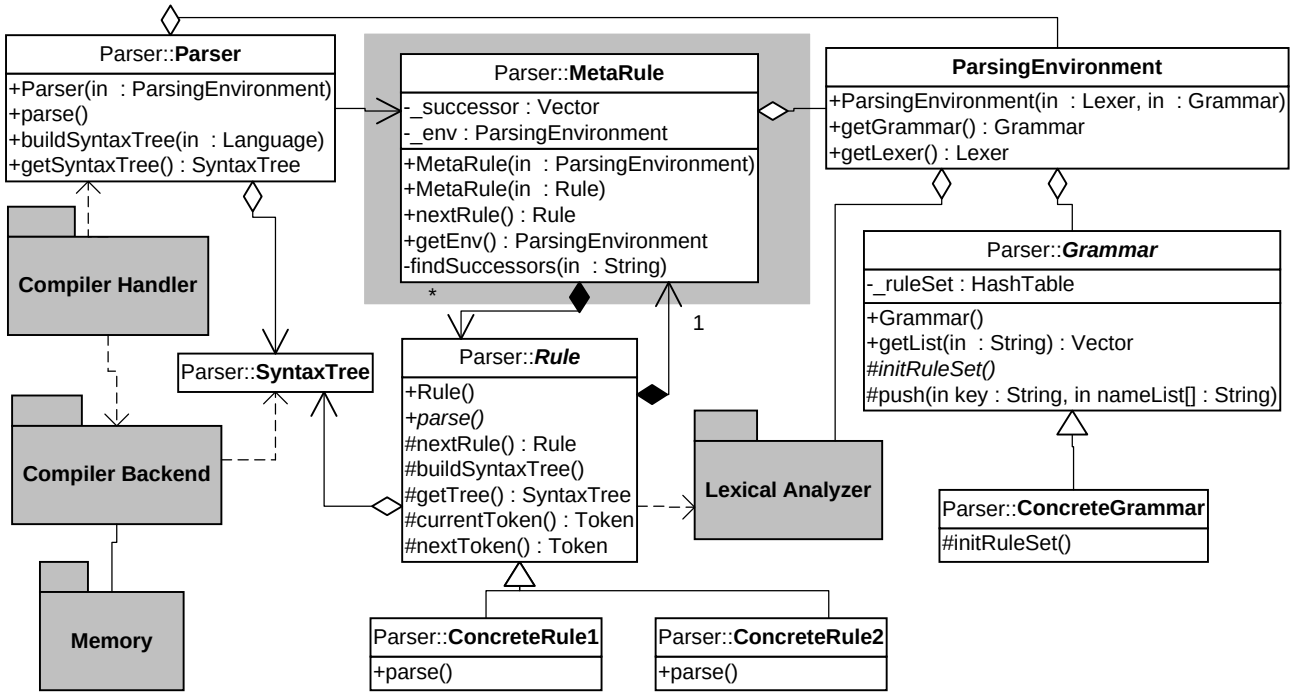

Figure 5: Structure for the MetaParser Pattern

- **Grammar**
  Defines a common interface for all grammar rules in the potential target languages. It contains a hash table that defines the relationships of the grammar rules.
- **ConcreteGrammar**
  Represents a grammar rule in the target language. It needs to initialize the hash table by specifying the actual grammar rules in use and their relationships.
- **SyntaxTree**
  A composite object structure that represents the parsing result.

Collaborations

Figure 6 shows the sequence diagram for the rule execution.

- The client invokes the *parse* method on the **Parser** to start the parsing process.
- The **Parser** initializes the **MetaRule** with the **ParsingEnvironment** object and invokes the *nextRule* method on its own **MetaRule** object to start the parsing. This **MetaRule** object then searches the hash table defined in the **ConcreteGrammar** to locate the start rule and creates the corresponding meta-object for the start.
- Once the **Parser** get the start **Rule** object from its **MetaRule**. It calls the *parse* method on that **Rule** object.
- When a rule is executed, it asks its own **MetaRule** object for the successors by invoking the *nextRule* method on this **MetaRule** object. The **MetaRule** object searches the **ConcreteGrammar** for the **Rule**'s successors. A *parse* method is then called on the successors.

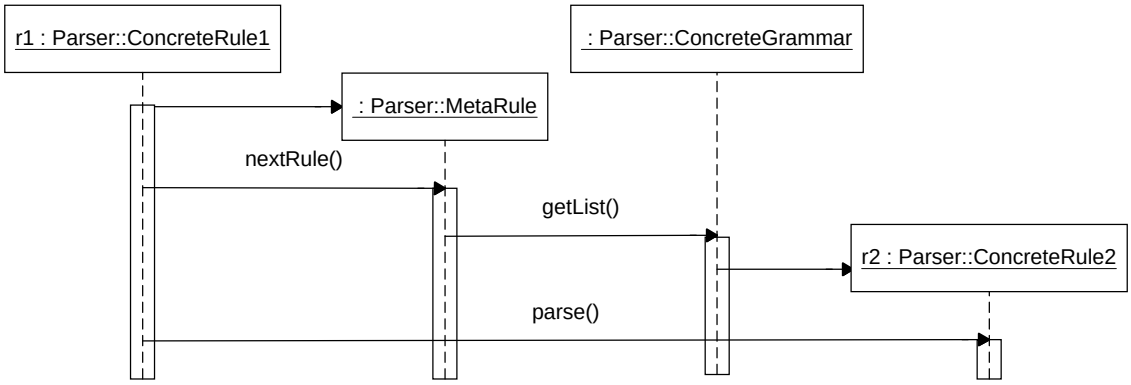

Figure 6: Sequence Diagram for the Parsing Process in the MetaParser

Consequences

- There is no need to explicitly modify the source code. Any potential changes are implicitly handled by the meta-level.
- The complexity of the system is reduced because the many-to-many relationships among the rules are changed to many-to-one relationship between the rules and the meta-level.
- The hash table that encapsulates the relationships of the rules can be modified or extended, the corresponding parsing logic and priority are then changed dynamically.
- A pool of grammar rules can be created and maintained, and optionally selected by the meta-level at run-time. The design promotes the reuse of the grammar rules even if they are defined for different target languages.
- A graded meta objects can be created to accommodate a graded complexity of the application logic. It is especially useful in incremental system development and testing.
- The design is more extensible and flexible. The grammar rules can be easily changed or extended without changing the existing classes. The hash table is free to add, delete, or modify an entry. The debug and test become easier. Any combination of the grammar rules can be set up in the hash table for different debugging purpose.
- There two major liabilities in the design. One is that the run-time efficiency is low due to the use of the reflection technique. The other is that the increased number of classes because each rule needs be represented as an individual class.

Related Patterns

The Reflection pattern [2] is used to discover the grammar rules at run-time.

The Accountability analysis pattern [3] defines a knowledge level (meta-level) and an operational level (base-level) to reduce the complexity of the system.

6 Conclusion

This paper intends to address the extensibility of the parser. The patterns presented can be easily used to build an extensible parser framework. The authors have used them to build a compiler framework [6], which

is implemented in Java. These patterns were also used in an extensible one-pass assembler developed by the authors [5]. This assembler is based on a virtual micro assembly language under a simple virtual processor (SVP) system and is implemented in Java. We agree that there exist different implementations of a parser in the compiler community, such as the table-driven parser, etc. Considering the recursive-descent parser is the one that is the most frequently used and the one that is the most difficult to extend in today's compiler design, we limit our discussion to the design of such a system to address its extensibility.

This pattern language is by no means complete. As long as experience is accumulated in the parser development, this language can be enriched when more and more patterns are added.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[3] Martin Fowler. *Analysis Pattern*. Addison-Wesley, 1997.

[4] Thomas W. Parsons. *Introduction To Compiler Construction*. , 1992.

[5] Yun Mai and Michel de Champlain. An Extensible One-Pass Assembler Framework. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, June 2000.

[6] Yun Mai and Michel de Champlain. Design A Compiler Framework in Java. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, November 2000.

[7] Yun Mai and Michel de Champlain. A Pattern Language to the Visitor Pattern. *In preparation to be submitted to 8th Conference PLoP '2001, Monticello, Illinois, USA*, September 2001.

[8] Yun Mai and Michel de Champlain. Reflective Visitor Pattern. *Submitted and accepted for EuroPLoP '2001 Writer's workshop, Irsee, Germany*, July 2001.