A Project Report on

"MATHEMATICAL PROGRAMMING LANGUAGE"

Submitted in partial fulfillment of the requirement for Degree in Bachelor of Engineering (Computer Engineering)

By

Ms. Alfy Samuel
Mr. Rohit Jha
Ms. Ashmee Pawar

Guided by

Prof. M. Kiruthika

Department of Computer Engineering
Fr. Conceicao Rodrigues Institute of Technology
Sector 9A, Vashi, Navi Mumbai – 400703

University of Mumbai
2012-2013

CERTIFICATE

This is to certify that the project entitled

MATHEMATICAL PROGRAMMING LANGUAGE

Submitted By

Ms. Alfy Samuel	100916
Mr. Rohit Jha	100923
Ms. Ashmee Pawar	100940

In partial fulfillment of degree of **B.E.(Semester VIII)** in **Computer Engineering** for term work of the project is approved.

External Examiner	**Internal Examiner**
________________	________________
External Guide	**Internal Guide**
________________	________________
Head of the Department	**Principal**
________________	________________
Date: -	**College Seal**

ACKNOWLEDGEMENT

This project has been a combined effort of not only the members of the group, but also teachers, friends and their families.

We thank our Principal, Dr. Rollin Fernandes for giving us the opportunity and permission to undertake the project. We would like to thank the Head of Department, Computer Engineering, Prof. Harish Kumar Kaura, for allowing us to take up the study of this topic. We are extremely grateful to our project guide, Prof. M. Kiruthika, for guiding us throughout the project and giving us her valuable feedback and support. We would also like to express sincere gratitude to our project coordinators, Prof. Smita Dange and Prof. Sandhya Pati.

ABSTRACT

Our proposed programming language is a Preprocessed Domain Specific Language (DSL) that enables implementation of concepts of discrete mathematics. The language has data structures and flow control structures that are expected in a programming language of this domain. The language covers the areas of Mathematical Logic, Set Theory, Functions, Graph Theory, Combinatorics, Linear Algebra and Number Theory.

A library of data types and functions provides functionality which is frequently required by mathematicians and computer scientists. The preprocessor implemented is a syntactical preprocessor, translating the DSL program into equivalent base language representation. This program is then compiled into binary format, i.e. machine code, and can be executed by users.

The advantage of our DSL is that users are provided with a notation close to the actual representation used for concepts of discrete mathematics. As a result, this language is better suited for usage than a library for a General Programming Language (GPL).

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

A programming language is an artificial language that is used to communicate instructions to a machine, particularly a computer. Programming languages are used to create "programs" that control the behavior of a machine and/or express algorithms precisely.

1.1 Elements of a Programming Language

1. *Syntax* - The syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.
2. *Semantics* – The term 'semantics' refers to the interpretation meaning of the languages as opposed to their syntax/form. Various terms related to semantics of programming languages are Static/Dynamic semantics, Static/Dynamic and/or Weak/Strong Typed languages.

1.2 Standard Library

Most programming languages have an associated core library, conventionally made available by all implementations of the language. This library typically includes definitions for commonly used algorithms, data structures and mechanisms for input and output.

1.3 Implementation

Broadly, programming languages can be implemented in two ways – *compiled* and *interpreted*. Compiler languages make use of a 'compiler', which translates high-level source code to a machine code, which in turn is later executed. Such languages are typically faster and require lesser memory than interpreted languages, which are implemented through an 'interpreter', which converts the source code to another high-level language and executes the new code. Such implementations are slower, but their performance can be improved by techniques such as Just-In-Time compilation in Virtual Machines, which operate on the bytecode.

Programming languages can also be classified as *General-Purpose* and *Domain-Specific*. General-Purpose Languages (GPLs) can be used for writing software in a variety of application domains. For example, Ada, C, C++, C#, Java, Perl, Python, Ruby and Scala are GPLs. Domain-Specific Languages (DSLs), on the other hand, are dedicated to a particular problem domain. Examples of DSLs include HTML, Logo, Verilog, VHDL, Mathematica, SQL and YACC.

1.4 Quality Requirements

The final program developed, irrespective of the methodology, must have the following properties:

1. *Reliability*
2. *Robustness*
3. *Usability*
4. *Portability*
5. *Maintainability*
6. *Efficiency/Performance*

1.5 Domain-Specific Languages

1.5.1 Usage Patterns

- *Processing with standalone tools*, invoked via direct user operation, often on the command line or from a Makefile.
- *Implemented using programming language macro systems*, and which are converted or expanded into a host GPL at compile-time or read-time.
- *Embedded (or internal) domain-specific languages*, implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.)
- DSLs which are called (at run-time) *from programs written in general purpose languages* like C or Perl, to perform a specific function, often returning the results of operation to the "host" programming language for further processing; generally, an interpreter or virtual machine for the domain-specific language is embedded into the host application.
- DSLs which are *embedded into user applications* (e.g., macro languages within spreadsheets) and which are used to execute code that is written by users of the application, and/or dynamically generated by the application.

1.5.2 Design Goals

- DSLs are less comprehensive.
- DSLs are much more expressive in their domain.
- DSLs should exhibit minimum redundancy.

1.5.3 Advantages

- Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. The idea is that domain experts themselves may understand, validate, modify, and often even develop domain-specific language programs. However, this is seldom the case.
- Self-documenting code.
- Domain-specific languages enhance quality, productivity, reliability, maintainability, portability and re-usability.
- Domain-specific languages allow validation at the domain level. As long as the language constructs are safe any sentence written with them can be considered safe.

1.5.4 Disadvantages

- Cost of learning a new language vs. its limited applicability
- Cost of designing, implementing, and maintaining a domain-specific language as well as the tools required to develop with it (IDE)
- Potential loss of processor efficiency compared with hand-coded software
- Proliferation of similar non-standard domain specific languages
- Non-technical domain experts can find it hard to write or modify DSL programs by themselves
- Increased difficulty of integrating the DSL with other system components
- Low supply of experts in a particular DSL tends to raise labor costs
- Harder to find code examples

2. LITERATURE SURVEY

2.1 Domain-Specific Languages

2.1.1 Definition

A Domain-Specific Language (DSL) is a programming language that is targeted to a particular problem area [1]. By contrast, a General Programming Language (GPL) is used for developing software in a variety of application domains. For example, HTML (Hypertext Markup Language), Logo, CSS (Cascading Style Sheets), Verilog, SQL (Structured Query Language), AutoCAD and YACC (Yet Another Compiler Compiler).

2.1.2 Characteristics of DSLs

Following are some vital characteristics of DSLs [2]:

1. A central and well-defined domain

   Focusing on the jargon of a problem domain rather than on the jargon of a computer implementation is a pervasive characteristic of good DSLs.

2. Clear notation

   Part of the design of a DSL is finding a good notation, and for practical reasons of storage and processing it is often convenient to use symbols that are easy to enter using a keyboard, mouse, or similar input devices. DSLs are designed to be simple, in order to reduce the learning time.

3. Comprehensible informal meaning

   A key part of what makes notations work is that they have a clear meaning, shared by all those who use them to communicate.

4. Well-suited for implementation

   This feature distinguishes a DSL from jargon; it means being amenable to rigorous, formal treatment, and being well-suited for sensible implementation by a machine. In spite of an increased start-up cost, DSL-based methodology renders a lesser Total Software Cost, compared to conventional methodology.

2.1.3 Need for DSLs

The following reasons have led to the need to create and use DSLs [1]:

- Creating a DSL can be worthwhile if the language *allows particular types of problems or solutions to be expressed more clearly* than what existing languages would allow, and also when the type of problem in question reappears sufficiently often.
- In order to reduce development time, tools with reusable code libraries are required. *Repetitive tasks to be performed are readily defined in DSLs* with custom libraries whose scope are restricted to the domain and hence need not be written from scratch each time.

- There is need for a solution that empowers experts with the power to specify the logic of their applications and maintain it at the same time as and when requirements change. Domain specific languages provide such solutions that *help domain experts to easily comprehend and create code for their application*. The self-documenting feature of DSLs supports it further.
- It is difficult to map conceptual model of solution into mainstream programming language as most time is spent in finding ways to express natural language concepts in terms of programming level abstractions (e.g. classes, methods, loops, conditionals, etc.). The mapping to DSLs becomes much easier and straightforward because *DSLs make use of terms and concepts dealt in the specific domain* instead of being forced to translate ideas into notion that a GPL is able to understand.

2.1.4 Classification of DSLs

- *Internal/Embedded DSLs* - It uses the infrastructure of an existing programming language (also called the host language of the DSL) to build domain-specific semantics on top of it. For example, Rails is an internal DSL implemented on top of the Ruby programming language.
- *External DSL* - It is developed ground-up and has separate infrastructure for lexical analysis, parsing techniques, interpretation, compilation, and code generation. Developing an external DSL is similar to implementing a new language from scratch with its own syntax and semantics. Build tools like make, parser generators like YACC, and lexical analysis tools like LEX are examples of popular external DSLs [3].

2.1.5 DSLs vs. GPLs

The advantages of DSLs over GPLs are listed below [1]:

- The scope of a DSL is only up to a specific domain. It therefore allows any domain expert to use it, in contrast with general purpose language that requires core programming capabilities in order to develop applications.
- Domain specific languages are very expressive i.e. their syntax is readable and easily understandable.
- DSLs reduce complexity by screening away the internal complex operations of the system. GPLs would require manual coding of every detail that becomes cumbersome and time consuming. This leads to concise semantic rules.
- DSLs are more productive as they need lesser programming time compared to GPLs.
- Domain specific languages support standardization wherein the underlying implementation can be changed without the need to change the code. For example, HTML is browser independent and can work on all kinds of browsers.

2.2 Guidelines for implementing DSLs

Guidelines can be categorized as follows based on a development-phase oriented classification [4]:

- *Language Purpose* - Discusses design guidelines for the early activities of the language development process.
- *Language Realization* - Introduces guidelines which discuss how to implement the language.
- *Language Content* - Contains guidelines which focus on the elements of a language.
- *Concrete Syntax* - Concentrates on design guidelines for the readable (external) representation of a language.
- *Abstract Syntax* - Concentrates on design guidelines for the internal representation of a language.

2.2.1 Language Purpose

Guideline 1 -

An *early identification of the language uses* have strong influence on the concepts the language will allow to offer. The concepts can be designed and analyzed for feasibility once the uses have been identified.

Guideline 2 -

Once the uses of a language have been identified it is helpful to embed these forms of language uses into the overall software development process. *People/roles have to be identified* that develop, review, and deploy the involved programs and models. After this, the developers can question whether the language is too complex or if it captures all the necessary domain elements.

Guideline 3 -

Since DSLs are typically designed for a specific purpose, *each feature of a language should contribute* to this purpose, otherwise it should be omitted for the language to remain consistent.

2.2.2 Language Realization

Guideline 4 -

The end-user's preferences must be matched with the advantages and disadvantages of both *textual and graphical realization*. Textual realization have the advantage of faster development and are platform and tool independent, whereas graphical models provide better overview and understanding of models in some cases.

Guideline 5 -

The labor-intensive task of developing a new language can be made easier with *reusing existing languages*. The most general and useful form of language reuse is thus the unchanged embedding of an existing language into another language.

Guideline 6 -

If the language cannot be simply composed from some given language parts, it is a good idea to reuse existing language definitions as much as possible. Taking the definition of *a language as a starter* to develop a new one is better than creating a language from scratch. Both the concrete and the abstract syntax will benefit from this form of reuse. The new language might then retain a look-and-feel of the original, thus allowing the user to easily identify familiar notations.

Guideline 7 -

A language designer should *reuse existing type systems* to improve comprehensibility and to avoid errors that are caused by misinterpretations in an implementation.

2.2.3 Language Content

Guideline 8 -

While designing a language, *only those domain concepts* need to be reflected *that contribute to the tasks* the language shall be used for.

Guideline 9 -

Simplicity is one of the main targets in designing languages. If it is complex, it raises the barrier of introducing the language. Even when such a language is introduced, unnecessary complexity minimizes the benefit the language should have yielded.

Guideline 10 -

Designing only what is necessary facilitates a quick and successful introduction of the DSL in the domain.

Guideline 11 -

Limiting the number of language elements makes the DSL easier to understand. To include elaborated content, libraries can be used. This results in a flexible, extensible and extensive, yet simple language.

Guideline 12 -

Having several concepts at hand to describe the same fact allows users to model it differently and this redundancy is a constant source of problems, such as those found in C++ and Perl. Thus, one must *avoid conceptual redundancy*.

Guideline 13 -

In order to have an efficient execution, and given the higher level of abstraction to be provided, *inefficient language elements must be avoided*, that would lead to poor generated code.

2.2.4 Concrete Syntax

Guideline 14 -

Rather than inventing a new notation, it is useful to *adopt the existing formal notation* used by domain experts.

Guideline 15 -

A *descriptive notation* supports both learnability and comprehensibility of a language especially when reusing frequently-used terms and symbols of domain knowledge.

Guideline 16 -

Easily distinguishable representations of language elements are a basic requirement to support understandability.

Guideline 17 -

Introduction of *syntactic sugar* can help improve the language's expressiveness and efficiency, if used appropriately.

Guideline 18 -

In order to make models more understandable to other developers, *provision of comments* must be made. Preferably, they must be a widely accepted standard form, such as /* … */ or //.

Guideline 19 -

Providing *organizational structure* for models, such as modules and packages, to the language makes it desirable for users to understand and use.

Guideline 20 -

Comprehensibility of notation is important and must be balanced with *compactness* for the language to be effective.

Guideline 21 -

To increase understandability, the *same look-and-feel* should be used for all the elements within a language.

Guideline 22 -

Usage conventions can be used which describe more detailed regulations that can, but need not be enforced.

2.2.5 Abstract Syntax

Guideline 23 -

Given the concrete syntax, the abstract syntax and especially its structure should *follow closely to the concrete syntax* to ease automated processing, internal transformations and also presentation (pretty printing) of the model.

Guideline 24 -

A *good layout* should be preferred so that it does not affect translation from concrete to abstract syntax.

Guideline 25 -

Enabling modularity helps in developing complex systems.

Guideline 26 -

Interfaces between parts of a model help users to provide proper exchange of data.

2.3 Developing DSLs

DSL development generally involves the following phases [5]:

1. Decision
2. Analysis
3. Design
4. Implementation
5. Deployment

2.3.1 Decision

The decision phase corresponds to the “when”-part of DSL development. Deciding in favor of a new DSL is usually not easy. The investment in DSL development (including deployment) has to pay for itself by more economical software development and/or maintenance later on. In practice, short-term considerations and lack of expertise may easily cause indefinite postponement of the decision.

To aid in the decision process, we identify a number of decision patterns. These are common situations that potential developers find themselves in that might motivate the use of DSLs. Underlying these patterns are general, interrelated concerns such as

- improved software economics,
- enabling of end-user programming or end-user specification,
- enabling of domain-specific analysis, verification, optimization, and/or transformation.

Following are some commonly used Decision Patterns:

1. *Notation*

   The availability of appropriate (new or existing) domain-specific notations is the decisive factor. Domain-specific notation beyond the limited userdefinable operator notation offered by GPLs may be added to an existing application library.

2. *Task Automation*

   Programmers often spend time on GPL programming tasks that are tedious and follow the same pattern. In such cases, the required code can be generated automatically by an application generator for an appropriate DSL.

3. *Data Structure Representation*

   Data-driven code relies on initialized data structures whose complexity may make them difficult to write and maintain. These structures are often more easily expressed using a DSL.

4. *Data Structure Traversal*

   Traversals over complicated data structures can often be expressed better and more reliably in a suitable DSL.

5. *System Front-end*

   DSL based front-end may be used for handling a system's configuration and adaptation.

6. *Interaction*

   Text or menu based interaction with application software often has to be supplemented with an appropriate DSL for the specification of complicated or repetitive input.

7. *AVOT (Analysis-Verification-Optimization-Transformation)*

   Domain-specific analysis, verification, optimization, and transformation of application programs written in a GPL are usually not feasible, because the source code patterns involved are too complex or not well defined. Use of an appropriate DSL makes these operations possible.

2.3.2 Analysis

In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered. This requires input from domain experts and/or the availability of documents or code from which domain knowledge can be obtained. Most of the time, domain analysis is done informally, but sometimes domain analysis methodologies such as DARE (Domain Analysis and Reuse Environment), DSSA (Domain-Specific Software Architectures), FODA (Feature-Oriented Domain Analysis) or ODM (Organization Domain Modeling) are used.

The output of formal domain analysis varies widely, but is some kind of representation of the domain knowledge obtained. It may range from a feature diagram, which is a graphical representation of assertions (propositions, predicates) about software systems in a particular domain, to a domain implementation consisting of a set of domain-specific reusable components, or a full-fledged theory in the case of highly developed scientific domains.

2.3.3 Design

Approaches to DSL design can be characterized along two orthogonal dimensions: the relationship between the DSL and existing languages, and the formal nature of the design description.

The easiest way to design a DSL is to base it on an existing language. One possible benefit is familiarity for users, but this only applies if the domain users are also programmers in the existing language. Another approach is to take an existing language and extend it with new features that address domain concepts. In most applications of this pattern the existing language features remain available. The challenge is to integrate the domain-specific features with the rest of the language in a seamless fashion.

The DSL designer has to keep in mind both the special character of DSLs as well as the fact that users need not be programmers. Since ideally the DSL adopts established notations of the domain, the designer should suppress a tendency to improve them.

Once the relationship to existing languages has been determined, a DSL designer must turn to specifying the design before implementation. In an informal design the specification is usually in some form of natural language probably including a set of illustrative DSL programs. A formal design would consist of a specification written using one of the available semantic definition methods. The most widely used formal notations include regular expressions and grammars for syntax specifications, and attribute grammars, rewrite systems and abstract state machines for semantic specification. There are several tools available which automate these techniques for DSL developers.

Commonly used Design Patterns for DSLs are:

- Language Exploitation

  DSL is based on an existing language. Important special cases:
  - *Piggyback*: Existing language is partially used.
  - *Specialization*: Existing language is restricted.
  - *Extension*: Existing language is extended.

- Language Extension

  A DSL is designed from scratch with no commonality with existing languages.

- Informal

  A DSL is described informally

- Formal

  A DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite systems, or abstract state machines.

2.3.4 Implementation

When a DSL is designed, the most suitable implementation approach should be chosen. Some approaches are:

1. *Interpreter*

   DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This is appropriate for languages having a dynamic character or if execution speed is not an issue. The advantages of interpretation over compilation are greater control over the execution environment and easier extension.

2. *Compiler/application generator*

   DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/ specification. DSL compilers are often called application generators.

3. *Preprocessor*

   DSL constructs are translated to constructs in the base language. Static analysis is limited to that done by the base language processor. Important special cases:

   - *Source-to-source transformation*: DSL source code is transformed (translated) into source code of existing language (the base language).
   - *Pipeline*: Processors successively handling sublanguages of a DSL and translating them to the input language of the next stage. This pattern also includes examples where only simple lexical processing is required, without complicated tree-based syntax analysis.

4. *Embedding*

   In the embedding approach, a DSL is implemented by extending an existing GPL (the host language) by defining specific abstract data types and operators. Application libraries are the basic form of embedding.

5. *Extensible compiler/interpreter*

   GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.

6. *Commercial Off-The-Shelf (COTS)*

   Existing tools and/or notations are applied to a specific domain.

7. *Hybrid*

   A combination of the above approaches is used.

2.3.4.1 Implementation Trade-offs

Majority of the DSLs are implemented using either the Interpreted design or Embedding design. The advantages and disadvantages for these are listed below:

2.3.4.1.1 Interpreted Design

Advantages:

- DSL syntax can be close to notations used by domain experts,
- Good error reporting possible,
- Domain-specific analysis, verification, optimization, and transformation (AVOT) possible

Disadvantages:

- The development effort is high because a complex language processor must be implemented,
- The DSL is more likely to be designed from scratch, often leading to incoherent designs compared with exploitation of an existing language,
- Language extension is hard to realize because most language processors are not designed with extension in mind.

The disadvantages can be minimized or eliminated when:

- A language development system or toolkit is used so that much of the work of language processor construction is automated, and
- A modular and extensible formal method for DSL design is used so that new features can be added without significant modification to the processing of old features.

2.3.4.1.2 Embedding Design

Advantages:

- Development effort is modest because an existing implementation can be reused,
- Often produces a more powerful language than other methods since many features come for free,
- Reuse of host language infrastructure (development and debugging environments: editors, debuggers, tracers, profilers etc.),
- User training costs might be lower since many users may already know the host language.

Disadvantages:

- Syntax is far from optimal as most languages do not allow arbitrary syntax extension,
- Overloading existing operators can be confusing if the new semantics does not have the same properties as the old,
- Bad error reporting because messages are in terms of host language concepts instead of DSL concepts,
- Domain-specific optimizations and transformations are hard to achieve, so efficiency may be affected, particularly when embedding in functional languages.

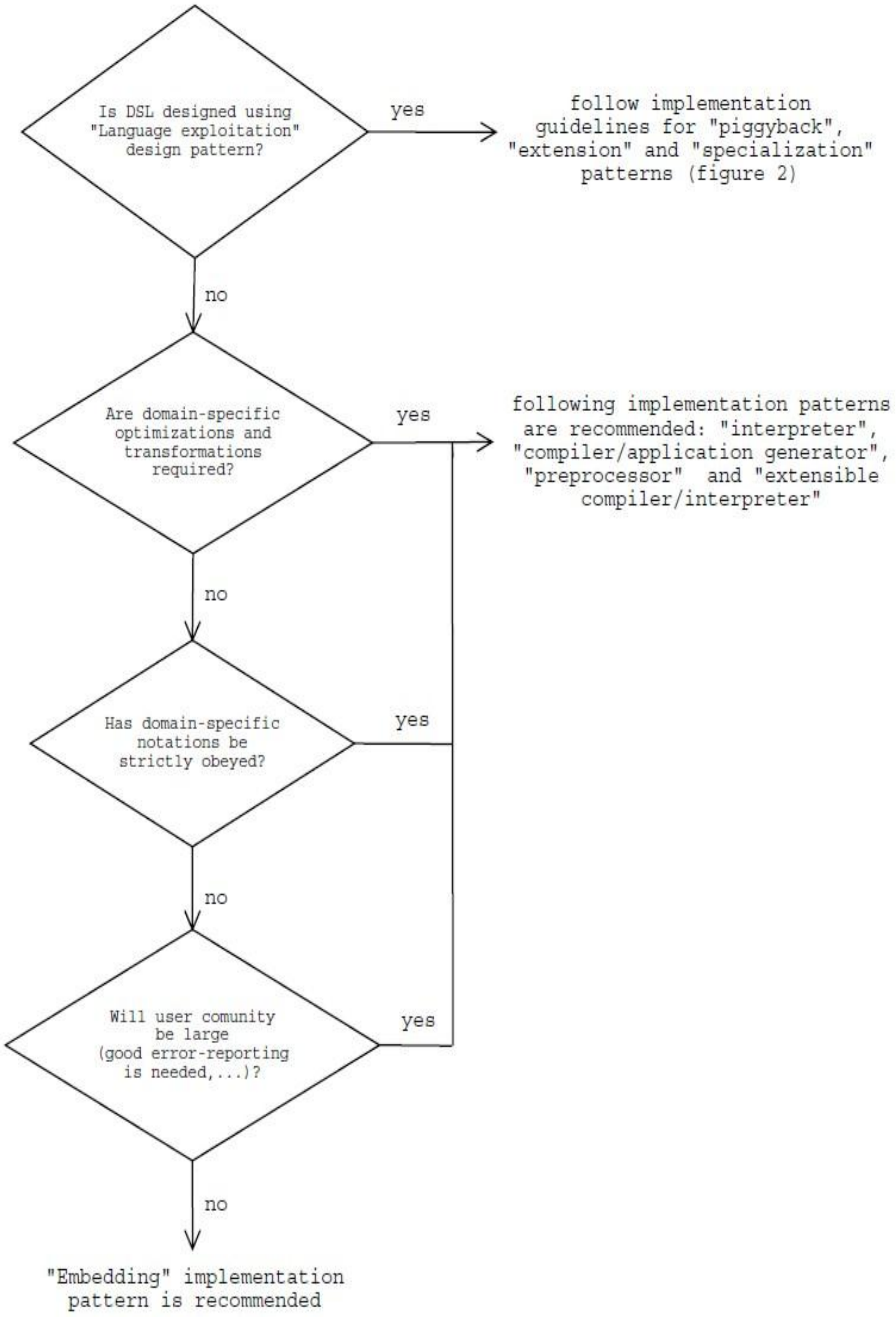

Is DSL designed using "Language exploitation" design pattern?
yes
follow implementation guidelines for "piggyback", "extension" and "specialization" patterns (figure 2)
no
Are domain-specific optimizations and transformations required?
yes
following implementation patterns are recommended: "interpreter", "compiler/application generator", "preprocessor" and "extensible compiler/interpreter"
no
Has domain-specific notations be strictly obeyed?
yes
no
Will user comunity be large (good error-reporting is needed,...)?
yes
no
"Embedding" implementation pattern is recommended

Fig. 1. Selecting an Implementation Design Pattern

2.3.5 Deployment

The mode of deployment of DSLs depends on the implementation design pattern selected. In case of an embedded DSL, the language can simply be deployed as a library for its base language. If the DSL is a preprocessed DSL, the preprocessor must be available to users as an executable file or installable module. If the implementation design pattern selected is either that of an interpreter of compiler/application generator, then a setup file would have to be created and be executed by users for installation.

2.3.6 DSL Design and Implementation Support

The DSL development process can be facilitated by using a language development system or toolkit. The available toolkits have widely different capabilities and are in widely different stages of development, but are based on the same general principle: they generate tools from language descriptions. Some of these systems support a specific DSL design methodology, while others have a largely methodology-independent character.

The input to these systems is a description of various aspects of the DSL to be developed in terms of specialized meta-languages. Depending on the type of DSL, some important language aspects are syntax, pretty-printing, consistency checking, execution, translation, transformation, and debugging. The meta-languages used for describing these aspects are themselves DSLs for the particular aspect in question. Some examples of such tools are:

- ASF + SDF
- AsmL
- Draco
- Eli
- Gem-Mex
- InfoWiz
- JTS
- Khepera
- Kodiyak
- LaCon
- LISA
- Metatool
- POPART
- smgn
- SPARK
- Sprint
- Stratego
- TXL

3. PROBLEM STATEMENT

A Domain-Specific Language proposed by us would aid users working with concepts of mathematics. The domain of this programming language is discrete mathematics. The basic library modules, which constitute the essential part of the language, include Set theory, Functions, Mathematical logic, Linear algebra and Number theory, while Combinatorics and Graph theory are the advanced modules.

As part of Set theory, the language supports concepts of Sets and Relations in the form of library modules. The library module for Graph theory provides data types, operations and functions on Graphs and Trees. Logical operators, namely NOT, AND, OR, NAND, NOR, XOR, XNOR, logical implication, logical equality and logical quantifiers (universal and existential) would be supported. Under Linear algebra, structures and functions for Matrices, Determinants and Vectors have been developed. Concepts relating to prime numbers, such as generation and testing, and multi-precision arithmetic would be included as part of Number theory. Support for Combinatorics would be in the form of functionality for calculating factorials, generating permutations and combinations of sets of elements. Our programming language would allow users to create and define their own functions. With these, a programmer can use complex functions as well.

4. SCOPE

The developed Domain Specific Language (DSL) has library modules for Set theory, Mathematical Logic, Graph theory, Combinatorics, Number theory, Linear algebra and Functions. In addition, it has a preprocessor that translates the DSL's syntax into equivalent base language syntax.

Besides aiding users working with structures of the aforementioned fields, the DSL would be useful in studying and describing objects and problems in branches of computer science, such as algorithms, programming languages, cryptography, automated theorem proving and software development. Such computer implementations are significant in applying ideas from discrete mathematics to real-world problems, such as in computer networks, operations research and social science.

For instance, Set theory is considered as a foundation for mathematical analysis, topology, abstract algebra, and discrete mathematics. Modern cryptography relies heavily on number theory. This is particularly true for public-key cryptography, which is employed for example in the SSL and TLS protocols. Furthermore, Graph theory finds applications in social networking, schedule development, design and analysis of computer networks, etc. Linear algebra is useful for solving Markov chains, which are probabilistic tools, used from biological population dynamics models and economics predictions, to traffic-flow models and incompressible fluid-flow dynamics. Combinatorics has many applications in optimization, computer science, analysis of algorithms, ergodic theory and statistical physics.

The syntax of our programming language is close to that used by domain experts for discrete mathematics. In this way, the language is easy to comprehend and learn for everyone.

5. SYSTEM DESIGN

5.1 Functional Programming Languages

Functional programming languages are a class of languages designed to reflect the way people think mathematically, rather than reflecting the underlying machine. The most commonly used functional languages are Standard ML, Haskell, and “pure” Scheme (a dialect of LISP), which, although they differ in many ways, share most of the properties of functional programming.

5.1.1 Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. It has its roots in lambda calculus to investigate function definition, function application, and recursion.

5.1.2 Benefits of Functional Programming

Following are the reasons why using functional programming is beneficial:

1. *Notation good for mathematical representation*

   It is possible to reason mathematically about functional programs in the same way one does in elementary algebra.

2. *Functions are first-class*

   A functional programming language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.

3. *Higher-order functions*

   They are functions that can either take other functions as arguments or return them as results. Higher-order functions enable partial application or currying, a technique in which a function is applied to its arguments one at a time, with each application returning a new function that accepts the next argument.

4. *Referential transparency*

   An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input). Unlike mathematics and programs in functional languages, programs in imperative languages lack referential transparency.

5. *Recursion*

   Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over.

6. *Good for structured programming*

   To make a program structured it is necessary to develop abstractions and split it into components which interface each other with those abstractions. Functional languages aid this by making it easy to create clean and simple abstractions.

7. *Short and easy to comprehend*

   Imperative programs tend to emphasize the series of steps taken by a program in carrying out an action, while functional programs tend to emphasize the composition and arrangement of functions, often without specifying explicit steps. This results in shorter codes.

8. *Ease of maintenance*

   The number of lines of code is a primary criteria for determining the ease of maintenance of programs. Since programs in functional programming languages are shorter, and are self-documenting in nature, they are easier to maintain than those written in imperative programming languages.

9. *High productivity*

   Shorter development time (due to reduced length of programs) and ease of maintenance increase programmers' productivity.

5.2 Haskell

Haskell is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing. It is named after logician Haskell Curry. In Haskell, "a function is a first-class citizen" of the programming language. As a functional programming language, the primary control construct is the function.

Haskell is unique for two reasons:

- It is purely functional. This means that in general, functions in Haskell do not have side effects. There is a distinct type for representing side effects, orthogonal to the type of functions. A pure function may return a side effect which is subsequently executed, modeling the impure functions of other languages.
- Haskell provides a very modern type system which incorporates features like typeclasses and generalized algebraic data types.

5.2.1 Advantages of Haskell

Following are the advantages of developing Domain-Specific Languages (DSLs) in Haskell:

- *Free and Open Source*

  Haskell has the benefit of being a free and open source software (FOSS) almost from the beginning. As a result, a variety of libraries, documentation and support is available for programmers. This also allows the language to continuously evolve over time.

- *Lazy Evaluation*

  Lazy evaluation or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing). The benefits of lazy evaluation include:

  - Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions
  - The ability to construct potentially infinite data structures
  - The ability to define control flow (structures) as abstractions instead of primitives

  In Haskell, an infinite-length list of natural numbers can be defined simply as:

  ```
  a = [1..]
  ```

  When this statement is executed, the entire infinite-length list is not loaded into the memory. Rather, when a particular element from the list is indexed, only then is it returned. The reference for index '5' is performed by the simple statement: `b = a !! 5`

- *Expressive type system*

  The use of algebraic data types and pattern matching makes manipulation of complex data structures convenient and expressive; the presence of strong compile-time type checking makes programs more reliable, while type inference frees the programmer from the need to manually declare types to the compiler.

- *Pure functional programming language*

  As a result of being a "pure" functional programming language, the notation of Haskell is closer to mathematical notations than other programming languages, functional or otherwise.

- *Very High Level Language (VHLL)*

  This results in providing users with good abstraction, aiding programmer productivity and enhancing maintainability of the programs. For instance, Quicksort in Haskell is:

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

This is much shorter than the equivalent program in imperative languages such as C or Java.

- *Composite functions*

  Provision of Higher Order Functions allows a programmer to work with composite functions, which are simply a combination of two or more first order functions. For Haskell, function composition can be explained by the following code:

```
desort = (reverse . sort)
  countdown = desort [2,8,7,10,1,9,5,3,4,6]
  -- output: [10,9,8,7,6,5,4,3,2,1]
```

  Here, the dot '.' operator is used for combining the two built-in functions 'sort' and 'reverse'. The argument of the first function is the value returned from the second.

- *Smart garbage collector*

  Haskell computations produce a lot of memory garbage - much more than conventional imperative languages. It's because data are immutable so the only way to store every next operation's result is to create new values. In particular, every iteration of a recursive computation creates a new value. But GHC (Glasgow Haskell Compiler) is able to efficiently manage garbage collection, so it's not uncommon to produce 1GB of data per second with most part being garbage collected immediately. Incidentally, GHC's efficiency in execution and memory management is second only to that of GCC.

- *Polymorphic types and functions*

  Most polymorphism in Haskell falls into one of two broad categories: parametric polymorphism and ad-hoc polymorphism. Parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types. Ad-hoc polymorphism refers to when a value is able to adopt any one of several types because it, or a value it uses, has been given a separate definition for each of those types. Polymorphism is defined for functions as well. This means that functions can take the same number of arguments as those of different data types. For example, a sorting function can take as input a list of integers, floating point numbers, strings or any other data type. A function signature such as `id :: a -> a` denotes that the function 'id' is defined for data 'a', which may be of any data type.

- *Extensible*

  Haskell was built keeping in mind the extensibility required for modern functional programming languages. This allows a provision of user defined functions, types, classes, modules, etc.

- *Very few reserved words*

  As a result, a programmer can have a multitude of names for variables and functions.

- *Flexible syntax*

  Haskell has a very flexible syntax, and offers higher-order functions. Therefore, we can often mimic the visual style of a particular domain directly within the language.

- *Syntactic sugar*

  Syntactic sugar is a computer science term that refers to syntax within a programming language that is designed to make things easier to read or to express. Specifically, a construct in a language is called syntactic sugar if it can be removed from the language without any effect on what the language can do: functionality and expressive power will remain the same.

- *Finite and infinite-precision integer arithmetic*

  Haskell has two integral types:
  - Int- limited-precision or single-precision integers
  - Integer - arbitrary-precision integers

5.3 Implementation Strategies

Using Haskell, the proposed Domain-Specific Language (DSL) for mathematics could be implemented in the following three ways:

5.3.1 Embedded DSL

An Embedded Domain Specific Language (EDSL), or Internal DSL, is a DSL that is defined as a library for a generic "host" programming language. The embedded DSL inherits the generic language constructs of its host language - sequencing, conditionals, iteration, functions, etc. - and adds domain-specific primitives that allow programmers to work at a much higher level of abstraction. There are two major degrees of embedding, shallow and deep.

- *Shallow Embedding*

  All Haskell operations immediately translate to the target language. E.g. the Haskell expression `a+b` is translated to a String like `"a + b"` containing that target language expression.

- *Deep Embedding*

  Haskell operations only build an interim Haskell data structure that reflects the expression tree. E.g. the Haskell expression `a+b` is translated to the Haskell data structure `Add (Var "a") (Var "b")`. This structure allows transformations like optimizations before translating to the target language.

For this style of programming to work well, the syntax of the generic language must be flexible and expressive enough to "get out of the way" of the embedded DSL. That usually means that the host language should have a very minimal syntax. Since Haskell provides us with all these, it is an ideal choice for a host programming language.

The advantage of building a DSL as an EDSL is that the development time is significantly reduced. Moreover, if users are satisfied with or used to the syntax of the host language, then they would face no problems in using the EDSL, since the syntax would remain identical.

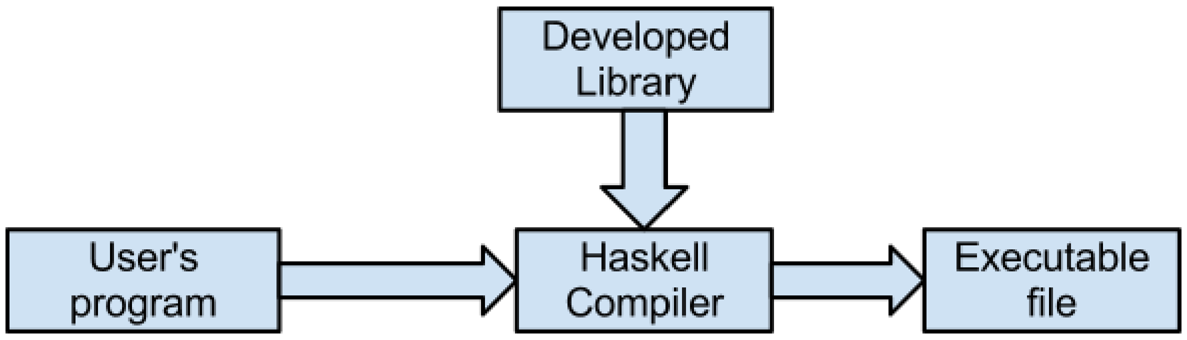


Fig. 2. Flow Diagram for Embedded DSL

The user’s program would be written in the new DSL. The library would have modules for functionality regarding Set theory, Mathematical logic, Combinatorics, etc. The Haskell compiler would take this program, along with the library, as input, and then produce an executable.

5.3.2 Preprocessed DSL

In situations where the syntax of a host language may be a limitation, DSLs can be developed by creating a preprocessor that translates the DSL’s syntax into the host language’s syntax, and then executes the resultant host language code. This type of preprocessor is classified as a Syntactic Preprocessor.

Programs for the language to be developed are operated upon by such a preprocessor, which not only translates the syntax into the host language’s syntax, but also imports required libraries for the generated code. This code is then be compiled and executed by the host language’s compiler to produce the output.

In this way, a preprocessor helps in

- Customizing syntax
- Extending a language
- Specializing a language

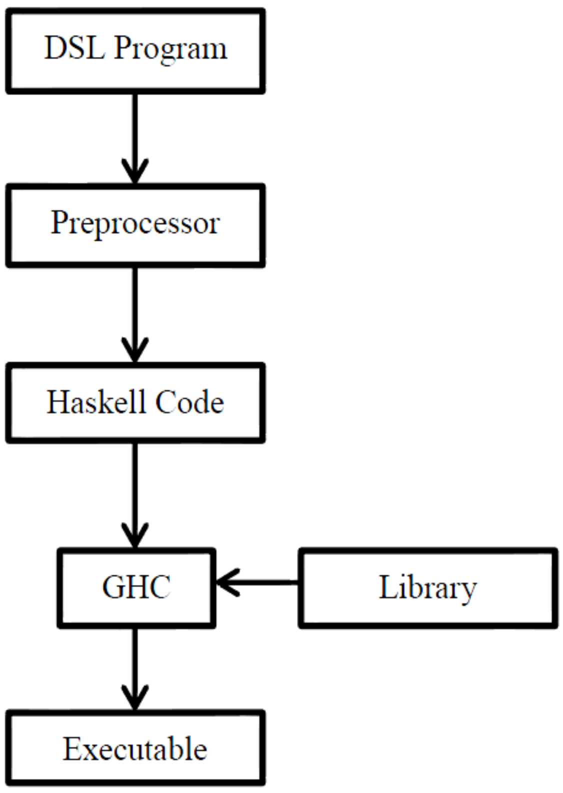


Fig. 3. Flow Diagram for Preprocessed DSL

The user's program would be written in the new DSL. The library would have modules for functionality regarding Set theory, Mathematical logic, Combinatorics, etc. The preprocessor would process the elements of the DSL and convert them to equivalent Haskell code. This code would then be compiled to produce an executable.

5.3.3 Interpreted DSL

An interpreter normally means a computer program that executes, i.e. performs, instructions written in a programming language. An interpreter may be a program that either

a) executes the source code directly
b) translates source code into some efficient intermediate representation (code) and immediately executes this
c) explicitly executes stored precompiled code made by a compiler which is part of the interpreter system

While interpreting and compiling are the two main means by which programming languages are implemented, these are not fully mutually exclusive categories, one of the reasons being that most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" merely mean that the canonical implementation of that language is an interpreter or a compiler; a high level language is basically an abstraction which is (ideally) independent of particular implementations.

Following are the advantages of using an interpreter for implementing a DSL:

- *Development cycle*

  A programmer using an interpreter does a lot less waiting, as the interpreter usually just needs to translate the code being worked on to an intermediate representation (or not translate it at all), thus requiring much less time before the changes can be tested.

- *Distribution*

  An interpreted program can be distributed as source code. It needs to be translated in each final machine, which takes more time but makes the program distribution independent of the machine's architecture.

The functioning of the interpreter can be depicted as:

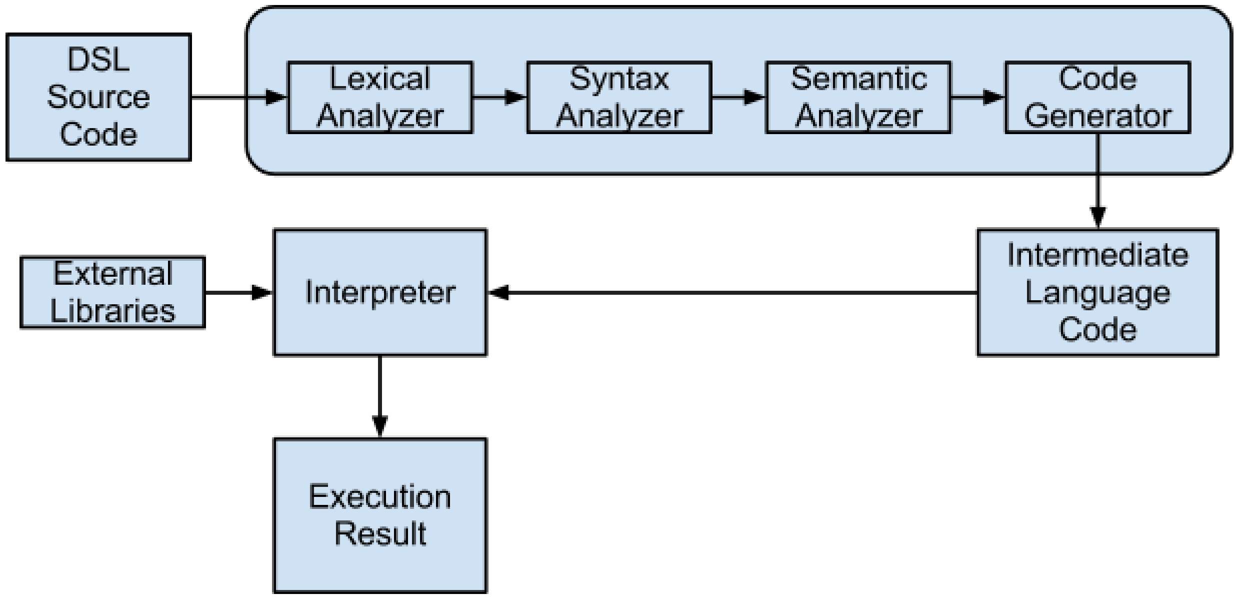


Fig. 4. Flow Diagram for Interpreted DSL

The DSL source code would be given as input to the developed interpretation system. This source code would be first passed to the Lexical Analyzer, which would tokenize the content and pass it to the Syntax Analyzer, and then to the Semantic Analyzer. On successful parsing, the Code Generator would produce an intermediate code representation. Combined with the external libraries, the Interpreter executes the Intermediate Language Code to give the execution result.

5.4 Selected Design

Of the three strategies mentioned previously, we have chosen to implement our programming language as a Preprocessed Domain-Specific Language (DSL). A preprocessor allows the DSL to have syntax different than the base language, which in this case is Haskell. Moreover, preprocessing is a pragmatic choice as efficiency of the generated code would not be affected if a Haskell compiler, such as GHC (Glasgow Haskell Compiler), were to operate on it by translating it to machine code.

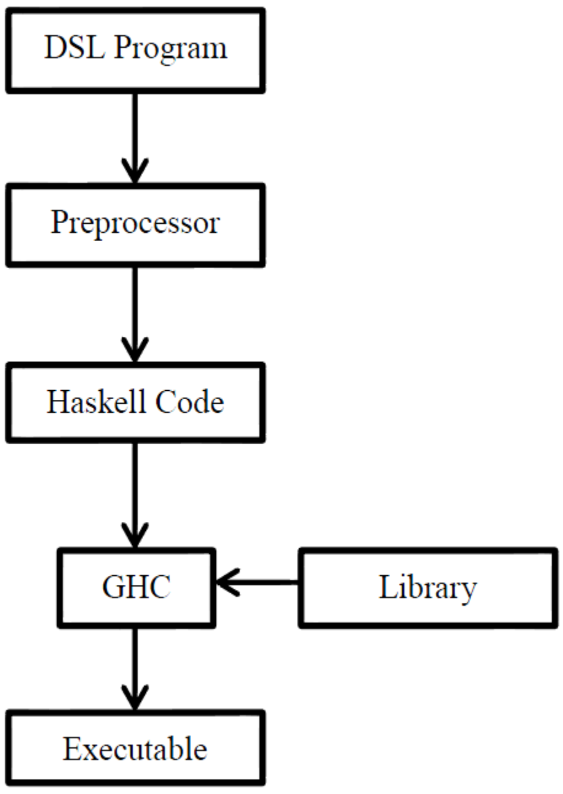


Fig. 5. The design selected from among the three possible ones

6. SYSTEM REQUIREMENTS

6.1 Hardware Requirements

To develop the Mathematical Programming Language, the system hardware requirements are as follows:

- *Development Platform Architecture:* 64-bit (x86-64/amd64)
- *Minimum Disk Space:* 200MB
- *Minimum Memory Required:* 64MB

6.2 Software Requirements

Following is a list of software required for the development of Mathematical Programming Language:

- *Development Platform*: Linux, x86-64/amd64
- *Operating System*: Fedora 17
- *Base Language*: Haskell
- *Haskell Compiler*: GHC (Glasgow Haskell Compiler), version 7.0.4

7. PROPOSED MODULES

The DSL's implementation required developing of a library of modules for discrete mathematics and a preprocessor. The details of the same are mentioned in the following sections.

7.1 Library Design

The library consists of modules containing data types and functions for the following fields of discrete mathematics:

- Mathematical Logic
- Set Theory
- Graph Theory
- Number Theory
- Linear Algebra
- Combinatorics

7.1.1 Mathematical Logic

Logic is a vital topic of discrete mathematics, with applications in foundations of mathematics, formal logic systems and proofs. Often, set theory, model theory and recursion theory are considered as subsections of logic. In the DSL, logical operators and quantifiers from propositional logic, Boolean algebra and predicate logic are supported. This includes operators such as negation (NOT), conjunction (AND), disjunction (OR), exclusive disjunction (XOR), inverse conjunction (NAND), inverse disjunction (NOR), inverse exclusive disjunction (XNOR), logical implication (if...then), logical equality (iff), universal quantifier (for all) and existential quantifier (there exists some) and parentheses – "(" and ")". Haskell provides a unary Boolean negation function (not) and binary operators for conjunction (&&) and disjunction (||), allowing development of other operators using these. Besides these, in Haskell, the universal and existential quantifiers are given by "forall" and „exists‟, respectively. The library module for mathematical logic also contains functions for applying the operations mentioned on lists of Boolean values.

7.1.2 Set Theory

According to Georg Cantor, the founder of set theory, a set is a gathering together into a whole of definite, distinct objects of our perception and of our thought - which are called elements of the set. The module currently focuses on naive set theory, operations on sets, relations, properties of relations and closures. Later, functionality would be added for groups, rings, fields, group-theoretic lattices and order-theoretic lattices, which find applications in cryptography and computational physics.

For sets, the module on set theory provides users with support for concepts such as checking for membership, empty/null set, subset, superset, generating power sets, finding cardinality, set difference, determining equality of sets, calculating Cartesian product, union of two sets, union of a list of sets, intersection of two sets, intersection of a list of sets, checking if two sets or a list of sets are disjoint, and mapping functions to sets. Working on sets is eased immensely with the provision of lists and list comprehension in Haskell.

Relations are sets of ordered pairs from elements of two sets, and are also called binary relations. The module for set theory in the library of the DSL contains functions for checking properties of relations. Important among these are those for checking if a relation is reflexive, symmetric, asymmetric, anti-symmetric, transitive, equivalence, partial order (weak or strict) and total order (weak or strict). With these as a base, functions for creating reflexive, symmetric and transitive closures are also developed and included in the library. As relations are essentially sets at their core, they can be combined by the operations of union, intersection, difference and composition. Composition also allows calculating powers of a relation and thus, the determination of transitive closures. The module also contains functions to check if a relation is a weak partial order, strong partial order, weak total order or strong total order.

7.1.3 Graph Theory

Considered the prime objects of study in discrete mathematics, and ubiquitous models for natural as well as man-made structures, graphs and trees are an important component of the DSL. This module provides support for users in computer science for studying networks, flow of computation, social network analysis, etc. In mathematics it would help users working with geometry, topology and group theory.

Graphs can be formally represented as the triple G = (V, E, ϕ), where V is a finite set of vertices, E is the finite set of edges and ϕ is the incidence function, with domain E and co-domain $P^2(V)$. Here, $P^2(V)$ represents the two-element subset of the power set P(V). For example, consider the graph represented as G = (V, E, ϕ), such that V = {A, B, C}, E = {a, b, c, d} and ϕ = {(A, B), (A, B), (A, C), (B, C)}. Graphs may also be directed, in which case, the co-domain of ϕ would become V*V. An example of a directed graph can be G = (V, E, ϕ), such that V = {A, B, C}, E = {a, b, c, d} and ϕ = {(A, B), (B, A), (A, C), (C, B)}.

Important graph operations such as finding in-degree and out-degree of vertices, finding nodes adjacent to a given node, checking for cycles, calculating union of graphs, determining if a graph is a subgraph of another, finding existence of Euler paths, Euler circuits, Hamiltonian paths and Hamiltonian circuits are included in the module for graph theory in the DSL‟s library as well. Apart from these operations, the library also contains algorithms for Dijkstra‟s shortest path, Prim‟s and Kruskal‟s Minimum Spanning Tree algorithms, Depth-First Search, and Breadth-First Search.

This library module also contains functionality for Trees, primarily in the form of Binary Trees. It also contains frequently used functions such as in-order, pre-order and post-order tree traversals, inserting nodes in a tree, finding total number of nodes, searching for a particular node using Binary Search, determining height of a tree, checking if a tree is balanced and calculating depth of a node.

7.1.4 Number Theory

Number theory is one of the oldest and largest branches of mathematics. It primarily deals with the study of integers, but it also involves studying prime numbers, rational numbers and equations. Some applications of concepts in number theory are finding solutions to simultaneous linear equations, numerical analysis, group theory, field theory and elliptic curve cryptography.

The module for number theory covers generation of prime numbers using Sieve of Eratosthenes, primality testing using trial division and Miller-Rabin test, prime factorization of integers and random number generation. This module also contains functions for Fibonacci numbers, including generating a list of Fibonacci terms and finding the *n*th term of the Fibonacci series.

Elementary number theory consists of base/radix operations and manipulations. Accordingly, the module provides support for handling bases ranging up from 1 to any integer. This includes operations of addition, subtraction, multiplication, division and exponentiation in all bases, apart from conversion of numbers from a particular base to another.

Another important part of number theory is modular arithmetic. The DSL's library supports solving linear congruence relations of the form $ax \equiv b \pmod{m}$ and also evaluation of modular operations such as addition, multiplication and exponentiation.

7.1.5 Linear Algebra

The branch of linear algebra deals with vector spaces and linear mappings between these spaces. These are used to represent systems of linear equations in multiple unknowns. Combined with calculus, linear algebra facilitates the solution of differential equations. Linear algebra is applied in quantum mechanics, systems using the Fourier series, and several fields where simultaneous linear equations need to be solved.

The module for linear algebra in the DSL's library contains data structures for Vectors and Matrices, which are the essence of linear algebra. Vectors are represented as *n*-valued tuples $<v_1,\ v_2\ \ldots\ v_n>$, and $n \times m$ Matrices as $[row_1,\ row_2\ \ldots\ row_n]$, where $row_i = [a_{i1},\ a_{i2}\ \ldots\ a_{im}]$ and a_{ij} is an element. For example, consider the examples of a vector used in three-dimensional Cartesian system: `Vector <3,2,-7>` and the third order unit matrix: `Matrix [[1,0,0], [0,1,0], [0,0,1]]`. Operations such as finding the order of a matrix, calculating trace, transpose, determinant, inverse, multiplication, division, addition, subtraction and power of matrices are frequently applied in matrix theory, and functions for the same have been included in the library module.

The module also contains functions for checking properties of a matrix or whether a matrix is of a certain type. Some of these include checking if a matrix is symmetric, skew-symmetric, orthogonal, involutory, 0/1, unit/identity matrix, a zero matrix or a one matrix. In addition, a mapping function for matrices allows the application of a single function to all the elements of a matrix. The module contains functions for generating unit matrices of order *n*, $m \times n$ zero and one matrices.

For vectors, functions are developed for addition, subtraction, multiplication (scalar/dot/inner product, vector product, scalar triple product and vector triple product), calculating magnitude of a vector, calculating angle between two vectors, mapping a function to a vector, checking if a vector is a unit vector, determining order of vectors and extracting an element or even a range of elements from a vector. The module also contains functions to find sum and difference of a list of Vectors.

7.1.6 Combinatorics

This branch of mathematics deals with the study of countable discrete structures. This involves counting the structures, determining criteria, and constructing and analyzing objects satisfying these criteria. In computer science, combinatorics is used frequently in analysis of algorithms to obtain estimates and formulas.

For users involved in computational combinatorics, this DSL would be helpful as it has a module consisting of frequently used functions such as those to find factorials, permutations and combinations, generate permutation and combination lists and also to generate random permutations using the Fisher-Yates/Knuth shuffle algorithm.

7.2 Preprocessor

The language is a Preprocessed DSL, wherein the Preprocessor is tasked with translating the language's syntax into equivalent Haskell representation. The Preprocessor is essentially a Bash script, 'preprocess.sh', which invokes another program, 'script', written in sed. The tool sed was selected since it provides excellent functionality for working with regular expressions. Since the preprocessor would not perform as much computation as a parser, scripts written in sed suffice. When the Bash script is called by GHC with users' programs written in the DSL as arguments, the sed script is executed over the programs and Haskell programs are generated for these files. This program is compiled by GHC to produce a binary executable. When executed, it generates the output.

The commands to compile and run a DSL program named 'myprogram.hs' are:

```
$ ghc -F -pgmF ./preprocess.sh myprogram.hs
$ ./myprogram
```

7.3 Deployment of the DSL

The DSL's library is written in Haskell, which is the base language, and can be packaged as an installable Haskell library using Cabal (Common Architecture for Building Applications and Libraries), which comes with the Haskell Platform. This package can be compressed in a gzipped tarball (.tar.gz) and uploaded on the web or community-repositories such as Hackage. A user need only download this file, extract the contents and setup the library like any other Haskell package, using the Setup.hs file. In addition, the Bash and sed scripts for Preprocessor can be downloaded and placed in a directory. Simply adding this directory to the OS's path and modifying the access privileges to add permissions for execution would allow completed use of the DSL.

8. IMPLEMENTATION AND RESULTS

This section describes modules from the DSL's library, including declaration of these modules and a few sample functions with results for every module. In addition, this section also contains details of applications developed using the DSL.

8.1 Mathematical Logic

The module for mathematical logic contains the following declaration for exporting functions to users' programs:

```
module MPL.Logic.Logic
(
     and',
     or',
     xor,
     xnor,
     nand,
     nor,
     equals,
     implies,
     (/\),
     (\/),
     (==>),
     (<=>),
     notL,
     andL,
     orL,
     xorL,
     xnorL,
     nandL,
     norL
)
where
```

Here, MPL.Logic.Logic is the module's name, indicating that the file is stored in the directory MPL/Logic and is named Logic.hs. This declaration is followed by definitions for each of the functions mentioned.

For example, consider the definition of the function for logical implication:

```
implies :: Bool -> Bool -> Bool
implies a b
     | (a == True)&&(b == False) = False
     | otherwise = True
```

In accordance with the objective of creating a notation close to the one actually used in discrete mathematics, an operator for logical implication is defined as follows:

```
(==>) :: Bool -> Bool -> Bool
a ==> b = implies a b
```

This provides syntactic sugar and improves readability. Now, the function for logical implication may be called by the user in any of the following three ways, all giving the same result - `False`:

```
implies True False
```

```
True `implies` False
```

```
True ==> False
```

As mentioned in 7.1.1, this module also defines functions which work on a list of Boolean values. The difference between the names of these functions and those of unary or binary functions is that they contain an additional 'L' as suffix, indicating that they operate on lists. A common operation is to find the XOR (Exclusive OR) of a list of values. Since the module already contains a function for finding the XOR of two values, it can be used to XOR the result of XOR of two values with the next value. Repeating this process for the length of the list gives a single final Boolean value. Such functions for lists of Boolean values are implemented using Haskell's `foldl1` function. The `xorL` function is defined as:

```
xorL :: [Bool] -> Bool
xorL a = foldl1 (xor) a
```

Here, a represents a list of Bool. An example of this function's usage is:

```
xorL [True, False, True, True, False]
```

This returns the `Bool` value `True`. The results of invoked functions and sample usage of operators are shown in Fig. 6.

8.2 Set Theory

Under set theory, the library contains modules for working on Sets and Relations.

8.2.1 Sets

The module for Sets is declared as:

```
module MPL.SetTheory.Set
(
      Set(..),
      set2list,
      union, unionL,
      intersection,
      intersectionL,
      difference,
      isMemberOf,
      cardinality,
      isNullSet,
      isSubset,
      isSuperset,
      powerSet,
      cartProduct,
      disjoint,
      disjointL,
      sMap
)
where
```

The function `union` is defined as:

```
union :: Ord a => Set a -> Set a -> Set a
union (Set set1) (Set set2)
     = Set $ (sort . nub) (set1 ++ [e | e <- set2, not (elem e
set1)])
```

This is based on the definition that the union of two sets is the set containing all elements from that first set, and all elements from the second set that are not in the first. In addition, duplicates from this set are removed and this resultant set is sorted. If this function is called as `union (Set {2,4,6}) (Set {1,2,3})`, the output would be the set `{1,2,3,4,6}`.

A common set operation is that of finding the Cartesian product of two sets. In the library module, it is defined as:

```
cartProduct :: Ord a => Set a -> Set a -> [(a,a)]
cartProduct (Set set1) (Set set2)
     = Set [(x,y) | x <- set1', y <- set2']
          where
               set1' = (sort . nub) set1
               set2' = (sort . nub) set2
```

This function may be called as `cartProduct (Set {1,2}) (Set {3,4})` to produce the result as the set `{(1,3),(1,4),(2,3),(2,4)}`.

In several conditions, it is requires to check if two sets are disjoint. For this, the module contains the function disjoint, and it is defined as:

```
disjoint :: Ord a => Set a -> Set a -> Bool
disjoint (Set s1) (Set s2) = isNullSet $ intersection (Set s1)
(Set s2)
```

If a user were to invoke this function as `disjoint (Set {1,3..10}) (Set {2,4..10})`, then he/she would get back `True` as the output. These results are shown in Fig. 7.

8.2.2 Relations

The module for Relations is declared as:

```
module MPL.SetTheory.Relation
(
     Relation(..),
     relation2list,
     getFirst,
     getSecond,
     elemSet,
     returnFirstElems,
     returnSecondElems,
     isReflexive,
     isIrreflexive,
     isSymmetric,
     isAsymmetric,
     isAntiSymmetric,
     isTransitive,
     rUnion,
     rUnionL,
     rIntersection,
     rIntersectionL,
     rDifference,
     rComposite,
     rPower,
     reflClosure,
     symmClosure,
     tranClosure,
     isEquivalent,
     isWeakPartialOrder,
     isWeakTotalOrder,
     isStrictPartialOrder,
     isStrictTotalOrder
)
where
```

Consider the definition for the `isTransitive` function:

```
isTransitive :: Eq a => Relation a -> Bool
isTransitive (Relation r)
= andL [(a,c) `elem` r | a <- elemSet r, b <- elemSet r, c <-
elemSet r, (a,b) `elem` r, (b,c) `elem` r]
```

This function may be called by the user as `isTransitive (Relation {(1,1),(1,2),(2,1)})`, which would return `False`. However, the call `isTransitive (Relation {(1,1),(1,2),(2,1),(2,2)})` would return `True`. This result is shown in Fig. 8.

The `symmClosure` function returns symmetric closure of the relation passed to it. It is defined as:

```
symmClosure :: Ord a => Relation a -> Relation a
symmClosure (Relation r) = rUnion (Relation r) (rPower
(Relation r) (-1))
```

This function uses the property that symmetric closure of a relation is the union of that relation with its inverse. Calling the function as `symmClosure (Relation {(1,1),(1,3)})` would give the result as the relation `{(1,1),(1,3),(3,1)}`.

8.3 Graph Theory

Under graph theory, the library contains modules for Graphs and Trees.

8.3.1 Graphs

Declaration for the module on graphs is:

```
module MPL.GraphTheory.Graph
(
     Vertices(..),
     vertices2list,
     Edges(..),
     edges2list,
     Graph(..),
     GraphMatrix(..),
     graph2matrix,
     getVerticesG,
     getVerticesGM,
     numVerticesG,
     numVerticesGM,
     getEdgesG,
     getEdgesGM,
     numEdgesG,
     numEdgesGM,
     convertGM2G,
     convertG2GM,
     gTransposeG,
     gTransposeGM,
     isUndirectedG,
     isUndirectedGM,
     isDirectedG,
     isDirectedGM,
     unionG,
     unionGM,
     addVerticesG,
     addVerticesGM,
     verticesInEdges,
     addEdgesG,
     addEdgesGM,
     areConnectedGM,
     numPathsBetweenGM,
     adjacentNodesG,
     adjacentNodesGM,
     inDegreeG,
     inDegreeGM,
     outDegreeG,
     outDegreeGM,
     degreeG,
     degreeGM,
     hasEulerCircuitG,
     hasEulerCircuitGM,
     hasEulerPathG,
     hasEulerPathGM,
     hasHamiltonianCircuitG,
     hasHamiltonianCircuitGM,
     countOddDegreeV,
     countEvenDegreeV,
     hasEulerPathNotCircuitG,
     hasEulerPathNotCircuitGM,
     isSubgraphG,
     isSubgraphGM
)
where
```

As stated in section 7.1.3, the module contains functions which work on graphs defined both formally and as matrices. Functions for the former have 'G' as suffix, while functions for the latter have 'GM' as suffix. The implementation of functions for both is made possible by the functions `convertG2GM` and `convertGM2G`, which convert between the formal and matrix representations.

Consider the function for determining if a graph is undirected:

```
isUndirectedGM :: Ord a => GraphMatrix a -> Bool
isUndirectedGM (GraphMatrix gm)
     = (GraphMatrix gm) == gTransposeGM (GraphMatrix gm)
```

When called as `isUndirectedGM (GraphMatrix [[0,5],[5,0]])`, `True` is returned. The invocation of functions for graphs is shown in Fig. 9.

Using the property of that a graph has an Euler circuit only if all vertices have even degree, the function `hasEulerCircuitG` is defined as:

```
hasEulerCircuitG :: Ord a => Graph a -> Bool
hasEulerCircuitG (Graph g)
= and [ even $ (degreeG (Graph g) (Vertices [v])) | v <-
vertices2list $ getVerticesG (Graph g)]
```

Thus, an invocation such as `hasEulerCircuitG (Graph (Vertices {1,2}, Edges {(1,2,4),(2,1,3)}))` would result in a return of `True`.

8.3.2 Trees

The module for trees has the following declaration:

```
module MPL.GraphTheory.Tree
(
     BinTree(..),
     inorder,
     preorder,
     postorder,
     singleton,
     treeInsert,
     treeSearch,
     reflect,
     height,
     depth,
     size,
     isBalanced
)
where
```

The functions `inorder`, `preorder` and `postorder` are functions for tree traversal. The definition for `preorder` is:

```
preorder :: BinTree a -> [a]
preorder Leaf = []
preorder (Node x t1 t2) = [x] ++ preorder t1 ++ preorder t2
```

If we consider the following `BinTree`:

```
tree =
     Node 4
          (Node 2
               (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf))
          (Node 7
               (Node 5
                    Leaf
                    (Node 6 Leaf Leaf))
               (Node 8 Leaf Leaf))
```

Then the function call, preorder tree, would generate the result `[4,2,1,3,7,5,6,8]`.

In essence, the `BinTree` data type is a Binary Search Tree. The function `treeSearch`, is an implementation of the Binary Search algorithm and has the following definition:

```
treeElem :: Ord a => a -> BinTree a -> Bool
treeElem x Leaf = False
treeElem x ( Node a left right )
     | x == a = True
     | x < a = treeElem x left
     | x > a = treeElem x right
```

The function `isBalanced` recursively checks if the height of all nodes at the same level are equal. The definition of this function makes use of the height function and is as follows:

```
isBalanced :: BinTree a -> Bool
isBalanced Leaf = True
isBalanced (Node x t1 t2) = isBalanced t1 && isBalanced t2 &&
(height t1 == height t2)
```

If this function is applied on tree as `isBalanced tree`, the output would be `False`. The results of functions for Trees are shown in Fig. 10.

8.4 Number Theory

Under number theory, the library contains the following modules:

4.4.1 Base/Radix Manipulation

This module has the description:

```
module MPL.NumberTheory.Base
(
     toBase,
     fromBase,
     toAlphaDigits,
     fromAlphaDigits
)
where
```

The function `toBase` converts a decimal number into the equivalent form of a specified base/radix. It has the definition:

```
toBase :: Int -> Int -> [Int]
toBase base v = toBase' [] v
     where
          toBase' a 0 = a
          toBase' a v = toBase' (r:a) q
               where
                    (q,r) = v `divMod` base
```

When invoked as `toBase 8 37` or as ``37 `toBase` 8``, the result would be `[4,5]`, which is read as 45, octal for 37. The result of `toBase` is also shown in Fig. 11.

4.4.2 Fibonacci Series

The module on Fibonacci series contains two functions, `fib` and `fibSeries`. The function `fib` takes an integer as parameter and returns the term at that index in the Fibonacci series. It is defined as:

```
fib n = round $ phi ** fromIntegral n / sq5
     where
          sq5 = sqrt 5 :: Double
          phi = (1 + sq5) / 2
```

If called as `fib 10`, the output is 55.

The `fibSeries` function takes an integer as parameter and returns the Fibonacci series as a list of integers. The definition is:

```
fibSeries n = [fib i | i <- [1..n]]
```

If a user wants to obtain the first 10 numbers in the Fibonacci series, he/she has to call the function as `fibSeries 10`, which gives the result `[1,1,2,3,5,8,13,21,34,55]`. The sample usage of functions from this module is shown is Fig. 12.

4.4.3 Modular Arithmetic

This module has the description:

```
module MPL.NumberTheory.Modular
(
     modAdd,
     modSub,
     modMult,
     modExp,
     isCongruent,
     findCongruentPair,
     findCongruentPair1
)
where
```

The `modExp` function is the function for modular exponentiation. It takes the numbers *a*, *b* and *m* as parameters and computes the value of *ab* mod *m*. The definition is:

```
modExp a b m = modexp' 1 a b
     where
     modexp' p _ 0 = p
     modexp' p x b =
          if even b
          then modexp' p (mod (x*x) m) (div b 2)
          else modexp' (mod (p*x) m) x (pred b)
```

If invoked as `modExp 112 34 546`, the integer 532 is returned. The sample usage of functions defined in this module is shown in Fig. 13.

4.4.4 Prime Numbers

This module has the following description:

```
module MPL.NumberTheory.Primes
(
     primesTo,
     primesBetween,
     firstNPrimes,
     isPrime,
     nextPrime,
     primeFactors
)
where
```

The function `primesTo` generates all prime numbers less than or equal to the number passed as parameter, using the Sieve of Eratosthenes. Its definition is:

```
primesTo :: Integer -> [Integer]
primesTo 0 = []
primesTo 1 = []
primesTo 2 = [2]
primesTo m = 2 : sieve [3,5..m]
```

The invocation `primesTo 20` produces the output as `[2,3,5,7,11,13,17,19]`. The usage of this function, as well as of the other functions from this module is shown in Fig. 14.

8.5 Linear Algebra

Under linear algebra, the library has modules for Matrices and Vectors.

4.5.1 Matrices

The module for matrices has the following description:

```
module MPL.LinearAlgebra.Matrix
(
     Matrix(..),
     mAdd,
     mAddL,
     (|+|),
     mSub,
     (|-|),
     mTranspose,
     mScalarMult,
     (|*|),
     mMult,
     mMultL,
     (|><|),
     numRows,
     numCols,
     mat2list,
     determinant,
     inverse,
     mDiv,
     (|/|),
     extractRow,
     extractCol,
     extractRowRange,
     extractColRange,
     mPower,
     trace,
     isInvertible,
     isSymmetric,
     isSkewSymmetric,
     isRow,
     isColumn,
     isSquare,
     isOrthogonal,
     isInvolutory,
     isZeroOne,
     isZero,
     isOne,
     isUnit,
     zero,
     zero’,
     one,
     one’,
     unit,
     mMap
)
where
```

The `mMult` function performs multiplication of two matrices and returns the resultant matrix. Its definition is:

```
mMult :: Num a => Matrix a -> Matrix a -> Matrix a
mMult (Matrix m1) (Matrix m2) = Matrix $ [ map (multRow r) m2t
| r <- m1 ]
     where
     (Matrix m2t) = mTranspose (Matrix m2)
     multRow r1 r2 = sum $ zipWith (*) r1 r2
```

To add syntactic sugar, the module provides the operator `|><|` for multiplying two matrices. Thus, if a user wishes to multiply a Matrix, `m1`, which is defined as `Matrix [[1,0],[0,1]]` and a Matrix, `m2`, which is defined as `Matrix [[4.5,8],[(-10),6]]`, he/she can call either `mMult m1 m2` or `m1 |><| m2`, to get the output as `Matrix [[4.5,8.0],[(-10.0),6.0]]`. The usage and result is shown in Fig. 15.

Another common operation is to find inverse of a matrix. In this module, the function `inverse` is defined using the functions `cofactorM` and `determinant` as:

```
inverse (Matrix m) = Matrix $ map (map (* recip det)) $
mat2list $ cofactorM (Matrix m)
     where
          det = determinant (Matrix m)
```

If called as `inverse (Matrix [[1,1],[1,(-1)]])`, the result is `Matrix [[0.5,0.5],[0.5,(-0.5)]]`.

The module contains several functions to check for properties of a matrix. One of these is `isOrthogonal`, which is to check if a matrix is orthogonal. Using the functions `mTranspose` and `inverse` it is easily defined as:

```
isOrthogonal (Matrix m) = (mTranspose (Matrix m) == inverse
(Matrix m))
```

When it is used as `isOrthogonal (Matrix [[1,1],[1.2,(-1.5)]])`, the output is `False`.

4.5.2 Vectors

This module's description is:

```
module MPL.LinearAlgebra.Vector
```

```
(
     Vector(..),
     vDim,
     vMag,
     vec2list,
     vAdd,
     vAddL,
     (<+>),
     vSub,
     vSubL,
     (<->),
     innerProd,
     (<.>),
     vAngle,
     scalarMult,
     (<*>),
     isNullVector,
     crossProd,
     (><),
     scalarTripleProd,
     vectorTripleProd,
     extract,
     extractRange,
     areOrthogonal,
     vMap,
     vNorm
)
where
```

The `vAngle` function returns the angle between two Vectors. It has the definition:

```
vAngle :: Floating a => Vector a -> Vector a -> a
vAngle (Vector []) (Vector []) = 0
vAngle (Vector v1) (Vector v2) = acos ( (innerProd (Vector v1)
(Vector v2)) / ( (vMag (Vector v1)) * (vMag (Vector v2)))
```

As shown in Fig. 16, when invoked as `vAngle (Vector [1,1,1]) (Vector [0,1,0])`, the result is 0.9553166181245092 (radians).

The function `scalarTripleProduct` is based on the functions `innerProduct` and `crossProduct`. It is defined as:

```
scalarTripleProd a b c = innerProd a (crossProd b c)
```

To normalize a Vector, the `vNorm` function can be used. It has the definition:

```
vNorm (Vector v) = scalarMult (1/(vMag (Vector v))) (Vector v)
```

If called as `vNorm (Vector [1,2,3])`, the output is the Vector `<0.2672612419124244, 0.5345224838248488, 0.8017837257372732>`

8.6 Combinatorics

This module has the description:

```
module MPL.Combinatorics.Combinatorics
(
     factorial,
     c,
     p,
     permutation,
     shuffle,
     combination
)
where
```

The function definition for `factorial` is:

```
factorial :: Integer -> Integer
factorial n
     | (n == 0) = 1
     | (n > 0) = product [1..n]
     | (n < 0) = error "Usage - factorial n, where 'n' is non-
negative."
```

This function can return arbitrarily large integers since its return type is Integer. When `factorial 5` is called, the result 120 is returned.

The factorial function acts as a base for other functions in the module. For example, the function `p` returns the number of possible permutations of *r* objects from a set of *n* given by *n*P*r*. It is defined as:

```
p :: Integer -> Integer -> Integer
p n r = div (factorial a) (factorial (a-b))
     where
          a = max n r
          b = min n r
```

When this function is called as `p 10 5` or ``10 `p` 5``, 30240 is the output. Usage of functions of this module is shown in Fig. 17.

8.7 Applications

This section contains descriptions of the applications developed using the Mathematical Programming Language as a DSL for discrete mathematics. These include ciphers such as those of Caesar and Transposition, RSA encryption and decryption system, implementation of the Diffie-Hellman Key Exchange Protocol, solution to simultaneous linear equations and Mersenne prime numbers.

8.7.1 Caesar Cipher

The Caesar Cipher is based on the concept of enciphering by replacing each character of a string by the character three positions to its right in the alphabet. The process of deciphering is the reverse of enciphering, i.e. replacing each character by the character three positions to its left in the alphabet. The implementation of this cipher in the DSL is shown in Fig. 18.

8.7.2 Transposition Cipher

Enciphering of a text by Transposition Cipher involves changing the relative positions of the characters forming the text. The result after enciphering appears as the jumbled string, containing the same characters as the plain text. The working of this cipher's implementation in the DSL is shown in Fig. 19.

8.7.3 RSA Encryption and Decryption

Using the library modules MPL.NumberTheory.Primes, MPL.NumberTheory.Modular and MPL.NumberTheory.Base of the Mathematical Programming Language, the RSA system for encryption and decryption was easily implemented. For implementation, the RSA algorithm was followed. This was extended to a menu-driven program, the results of which can be seen in Fig. 20 and Fig. 21.

8.7.4 Diffie-Hellman Key Exchange

The Diffie-Hellman Key Exchange protocol was implemented in the DSL using the library modules of MPL.NumberTheory.Modular, MPL.NumberTheory.Primes and MPL.NumberTheory.Base. This algorithm involves selecting common primitive root and prime number, and then generation of a shared key by a party using the other's public key and its own private key. This algorithm's implementation is the DSL is shown in Fig. 22.

8.7.5 Simultaneous Linear Equation

The solution to simultaneous linear equations can be found very easily by using the extensive functionality of the module on matrices in the DSL's library. The command line usage is shown in Fig. 23. The usage of the DSL program in Eclipse is shown in Fig. 24. The program is:

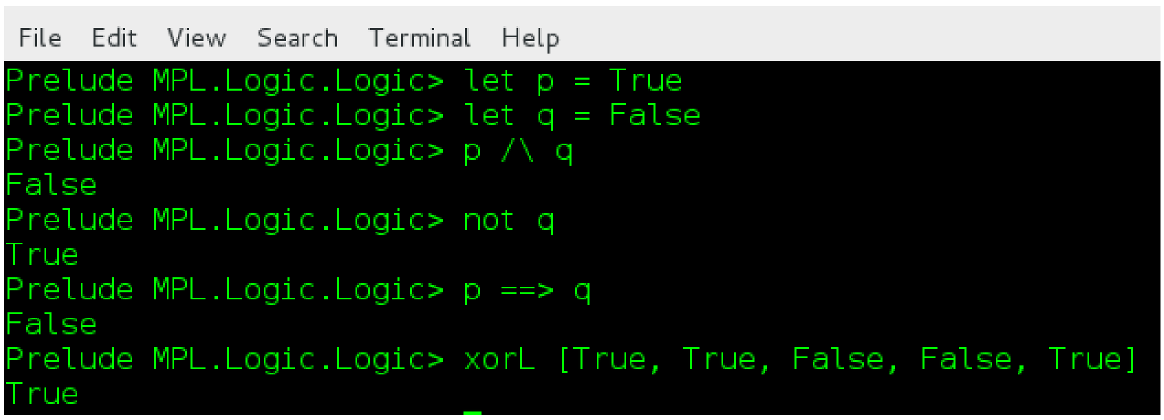
```
import MPL.LinearAlgebra.Matrix

solveEqns (Matrix coeff) (Matrix const) = (inverse (Matrix
coeff)) |><| (Matrix const)
```

8.7.6 Mersenne Prime Numbers

Mersenne Prime Numbers are prime numbers of the form $2^q - 1$, where q is also a prime number. Since the DSL's library on prime numbers, MPL.NumberTheory.Primes, provides efficient ways to deal with prime numbers, the finding of even large Mersenne prime numbers is a cinch. The results are shown in Fig. 25.

9. SCREENSHOTS

9.1 Mathematical Logic

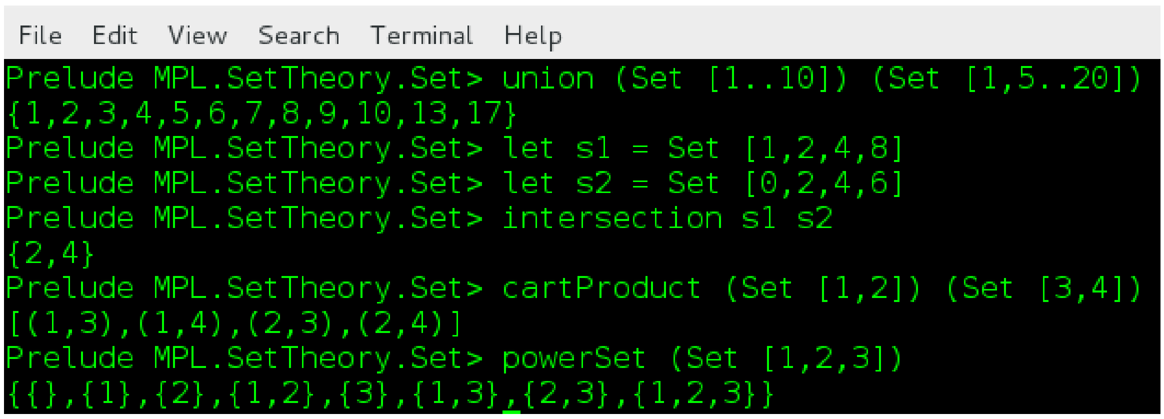


```
Prelude MPL.Logic.Logic> let p = True
Prelude MPL.Logic.Logic> let q = False
Prelude MPL.Logic.Logic> p /\ q
False
Prelude MPL.Logic.Logic> not q
True
Prelude MPL.Logic.Logic> p ==> q
False
Prelude MPL.Logic.Logic> xorL [True, True, False, False, True]
True
```

Fig. 6. Functions for Mathematical Logic in GHCi

9.2 Set Theory

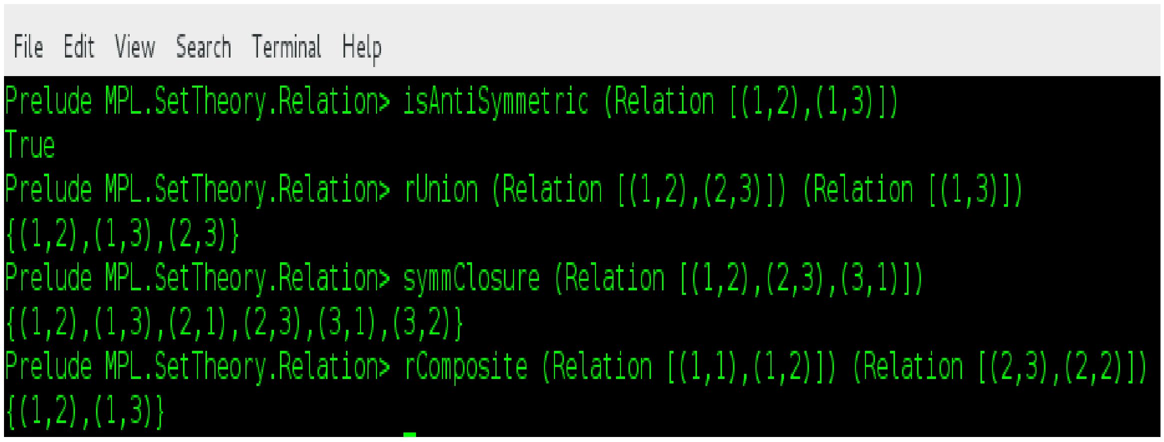
9.2.1 Sets



```
Prelude MPL.SetTheory.Set> union (Set [1..10]) (Set [1,5..20])
{1,2,3,4,5,6,7,8,9,10,13,17}
Prelude MPL.SetTheory.Set> let s1 = Set [1,2,4,8]
Prelude MPL.SetTheory.Set> let s2 = Set [0,2,4,6]
Prelude MPL.SetTheory.Set> intersection s1 s2
{2,4}
Prelude MPL.SetTheory.Set> cartProduct (Set [1,2]) (Set [3,4])
[(1,3),(1,4),(2,3),(2,4)]
Prelude MPL.SetTheory.Set> powerSet (Set [1,2,3])
{{},{1},{2},{1,2},{3},{1,3},{2,3},{1,2,3}}
```

Fig. 7. Functions for Sets in GHCi

9.2.2 Relations

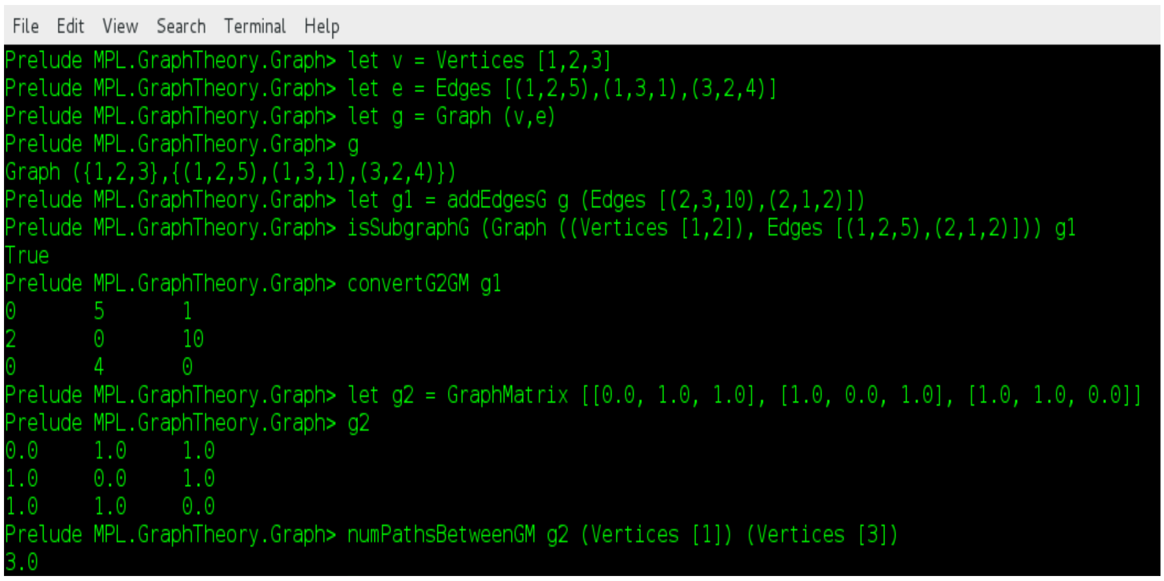


```
Prelude MPL.SetTheory.Relation> isAntiSymmetric (Relation [(1,2),(1,3)])
True
Prelude MPL.SetTheory.Relation> rUnion (Relation [(1,2),(2,3)]) (Relation [(1,3)])
{(1,2),(1,3),(2,3)}
Prelude MPL.SetTheory.Relation> symmClosure (Relation [(1,2),(2,3),(3,1)])
{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)}
Prelude MPL.SetTheory.Relation> rComposite (Relation [(1,1),(1,2)]) (Relation [(2,3),(2,2)])
{(1,2),(1,3)}
```

Fig. 8. Functions for Relations in GHCi

9.3 Graph Theory

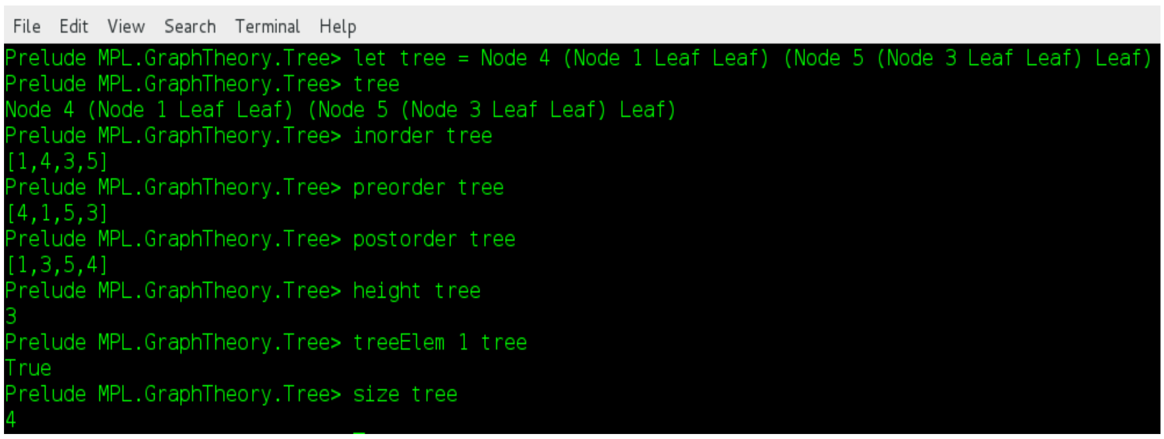
9.3.1 Graphs



```
Prelude MPL.GraphTheory.Graph> let v = Vertices [1,2,3]
Prelude MPL.GraphTheory.Graph> let e = Edges [(1,2,5),(1,3,1),(3,2,4)]
Prelude MPL.GraphTheory.Graph> let g = Graph (v,e)
Prelude MPL.GraphTheory.Graph> g
Graph ({1,2,3},{(1,2,5),(1,3,1),(3,2,4)})
Prelude MPL.GraphTheory.Graph> let g1 = addEdgesG g (Edges [(2,3,10),(2,1,2)])
Prelude MPL.GraphTheory.Graph> isSubgraphG (Graph ((Vertices [1,2]), Edges [(1,2,5),(2,1,2)])) g1
True
Prelude MPL.GraphTheory.Graph> convertG2GM g1
0       5       1
2       0       10
0       4       0
Prelude MPL.GraphTheory.Graph> let g2 = GraphMatrix [[0.0, 1.0, 1.0], [1.0, 0.0, 1.0], [1.0, 1.0, 0.0]]
Prelude MPL.GraphTheory.Graph> g2
0.0     1.0     1.0
1.0     0.0     1.0
1.0     1.0     0.0
Prelude MPL.GraphTheory.Graph> numPathsBetweenGM g2 (Vertices [1]) (Vertices [3])
3.0
```

Fig. 9. Functions for Graphs in GHCi

9.3.2 Trees

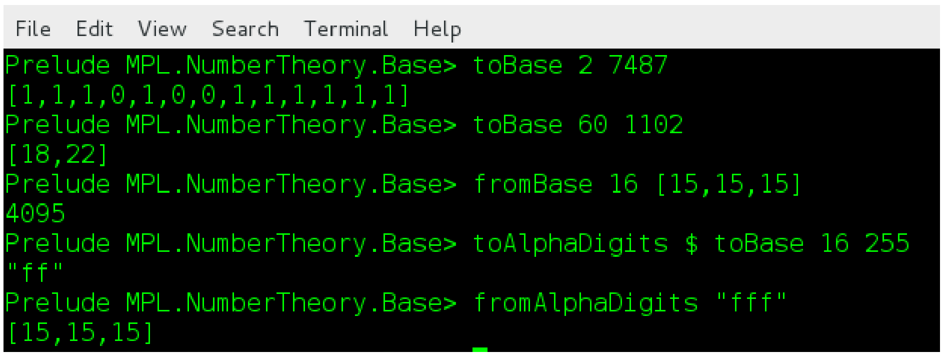


```
Prelude MPL.GraphTheory.Tree> let tree = Node 4 (Node 1 Leaf Leaf) (Node 5 (Node 3 Leaf Leaf) Leaf)
Prelude MPL.GraphTheory.Tree> tree
Node 4 (Node 1 Leaf Leaf) (Node 5 (Node 3 Leaf Leaf) Leaf)
Prelude MPL.GraphTheory.Tree> inorder tree
[1,4,3,5]
Prelude MPL.GraphTheory.Tree> preorder tree
[4,1,5,3]
Prelude MPL.GraphTheory.Tree> postorder tree
[1,3,5,4]
Prelude MPL.GraphTheory.Tree> height tree
3
Prelude MPL.GraphTheory.Tree> treeElem 1 tree
True
Prelude MPL.GraphTheory.Tree> size tree
4
```

Fig. 10. Functions for Trees in GHCi

9.4 Number Theory

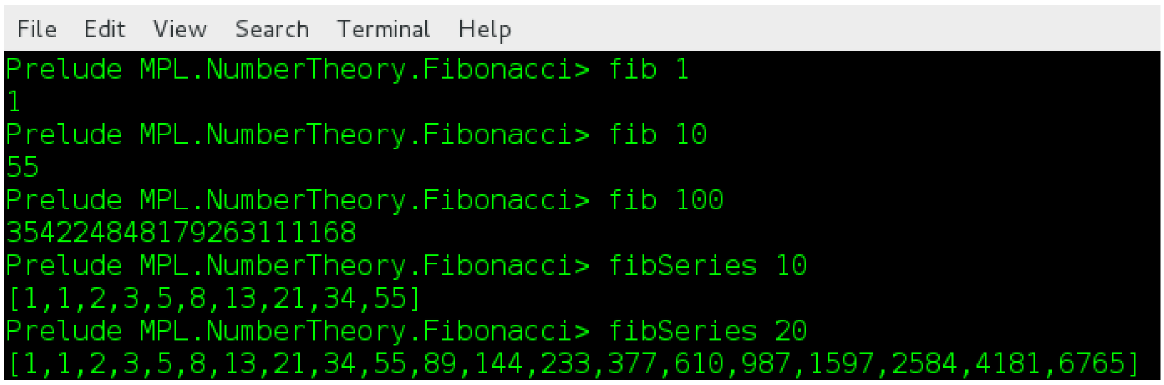
9.4.1 Base Manipulation



```
Prelude MPL.NumberTheory.Base> toBase 2 7487
[1,1,1,0,1,0,0,1,1,1,1,1,1]
Prelude MPL.NumberTheory.Base> toBase 60 1102
[18,22]
Prelude MPL.NumberTheory.Base> fromBase 16 [15,15,15]
4095
Prelude MPL.NumberTheory.Base> toAlphaDigits $ toBase 16 255
"ff"
Prelude MPL.NumberTheory.Base> fromAlphaDigits "fff"
[15,15,15]
```

Fig. 11. Functions for Base Manipulation in GHCi

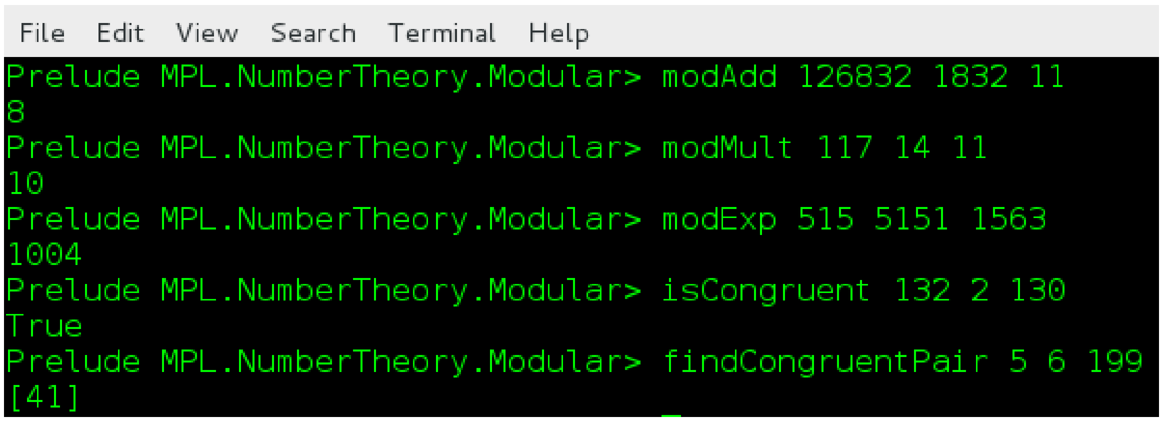
9.4.2 Fibonacci Sequence



```
Prelude MPL.NumberTheory.Fibonacci> fib 1
1
Prelude MPL.NumberTheory.Fibonacci> fib 10
55
Prelude MPL.NumberTheory.Fibonacci> fib 100
354224848179263111168
Prelude MPL.NumberTheory.Fibonacci> fibSeries 10
[1,1,2,3,5,8,13,21,34,55]
Prelude MPL.NumberTheory.Fibonacci> fibSeries 20
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

Fig. 12. Functions for Fibonacci Sequence in GHCi

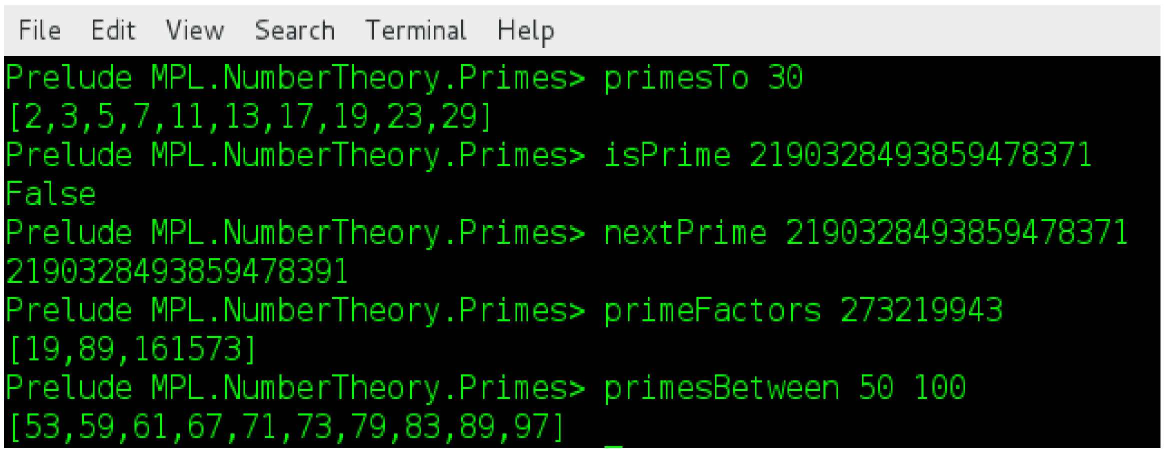
9.4.3 Modular Arithmetic



```
Prelude MPL.NumberTheory.Modular> modAdd 126832 1832 11
8
Prelude MPL.NumberTheory.Modular> modMult 117 14 11
10
Prelude MPL.NumberTheory.Modular> modExp 515 5151 1563
1004
Prelude MPL.NumberTheory.Modular> isCongruent 132 2 130
True
Prelude MPL.NumberTheory.Modular> findCongruentPair 5 6 199
[41]
```

Fig. 13. Functions for Modular Arithmetic in GHCi

9.4.4 Prime Numbers

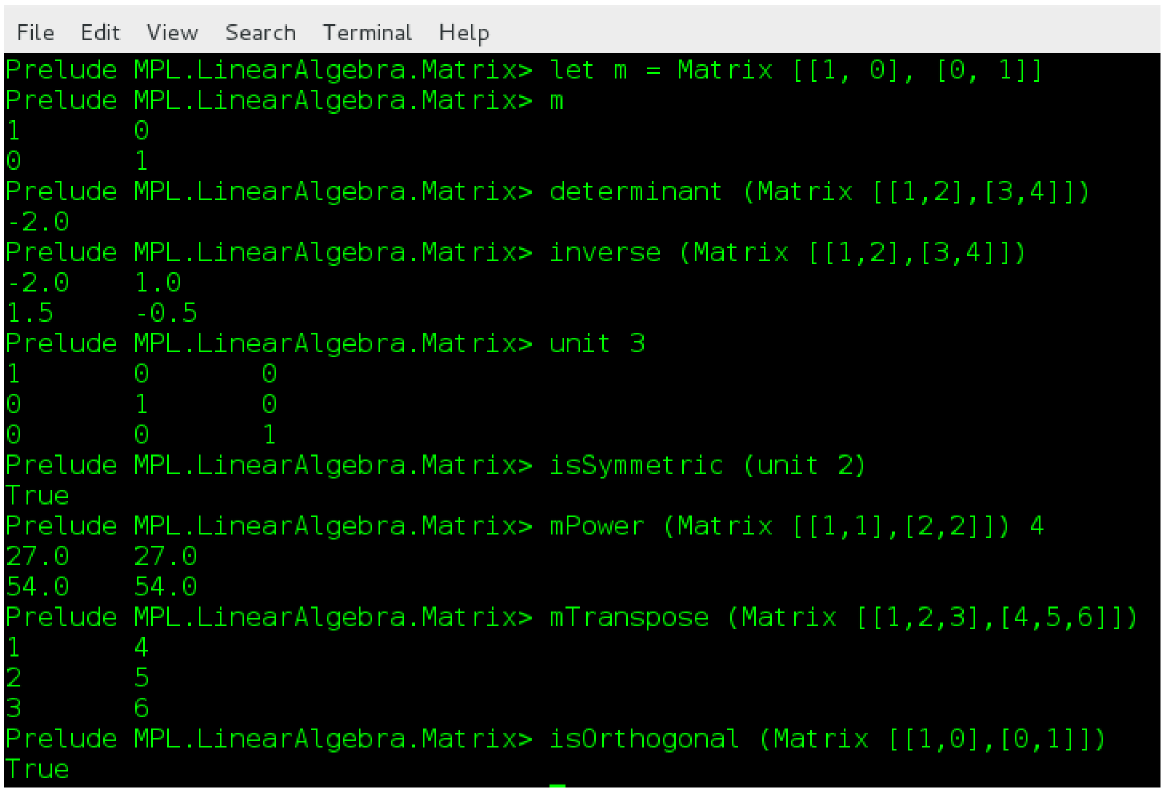


```
Prelude MPL.NumberTheory.Primes> primesTo 30
[2,3,5,7,11,13,17,19,23,29]
Prelude MPL.NumberTheory.Primes> isPrime 2190328493859478371
False
Prelude MPL.NumberTheory.Primes> nextPrime 2190328493859478371
2190328493859478391
Prelude MPL.NumberTheory.Primes> primeFactors 273219943
[19,89,161573]
Prelude MPL.NumberTheory.Primes> primesBetween 50 100
[53,59,61,67,71,73,79,83,89,97]
```

Fig. 14. Functions for Prime Numbers in GHCi

9.5 Linear Algebra

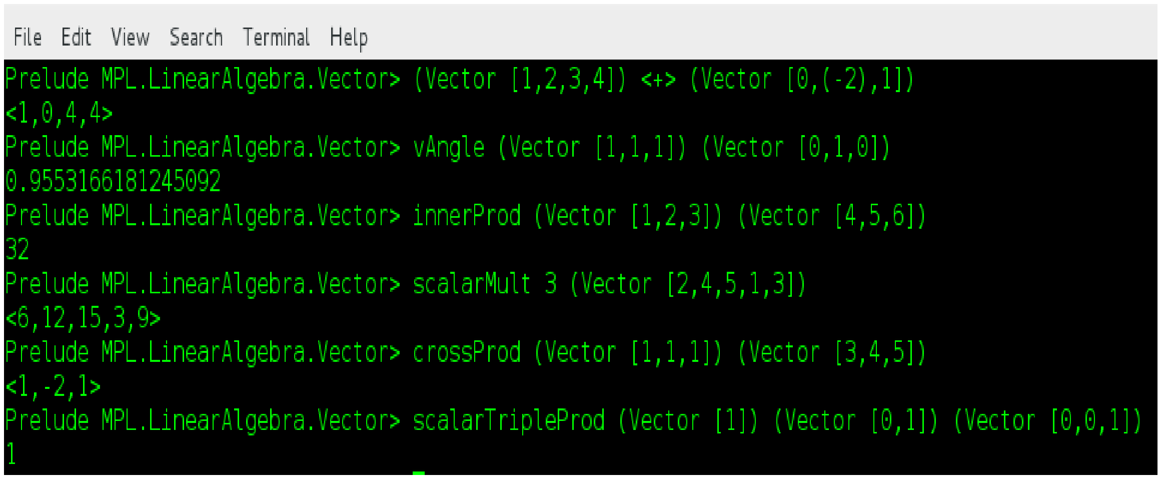
9.5.1 Matrices



```
Prelude MPL.LinearAlgebra.Matrix> let m = Matrix [[1, 0], [0, 1]]
Prelude MPL.LinearAlgebra.Matrix> m
1       0
0       1
Prelude MPL.LinearAlgebra.Matrix> determinant (Matrix [[1,2],[3,4]])
-2.0
Prelude MPL.LinearAlgebra.Matrix> inverse (Matrix [[1,2],[3,4]])
-2.0    1.0
1.5     -0.5
Prelude MPL.LinearAlgebra.Matrix> unit 3
1       0       0
0       1       0
0       0       1
Prelude MPL.LinearAlgebra.Matrix> isSymmetric (unit 2)
True
Prelude MPL.LinearAlgebra.Matrix> mPower (Matrix [[1,1],[2,2]]) 4
27.0    27.0
54.0    54.0
Prelude MPL.LinearAlgebra.Matrix> mTranspose (Matrix [[1,2,3],[4,5,6]])
1       4
2       5
3       6
Prelude MPL.LinearAlgebra.Matrix> isOrthogonal (Matrix [[1,0],[0,1]])
True
```

Fig. 15. Functions for Matrices in GHCi

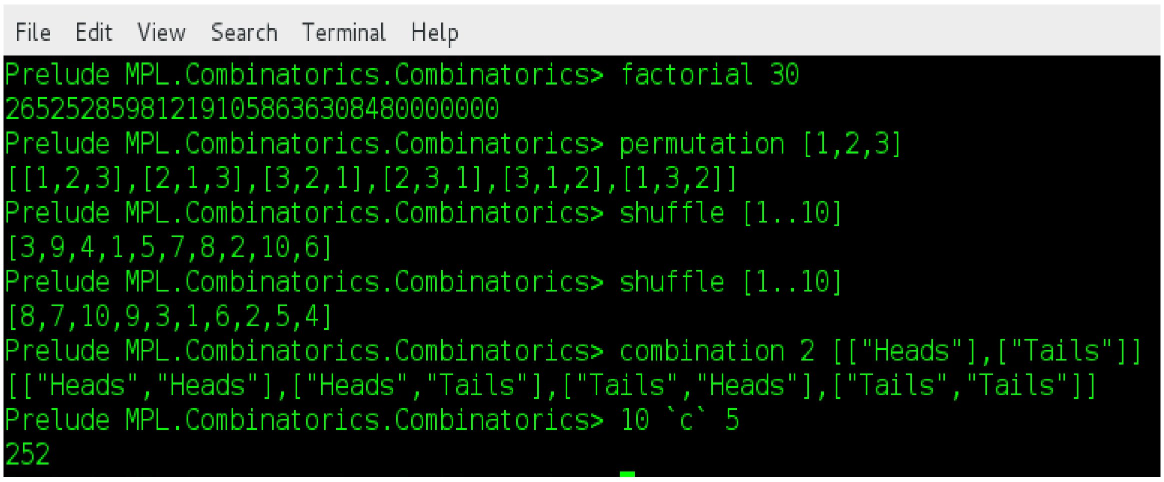
9.5.2 Vectors



```
Prelude MPL.LinearAlgebra.Vector> (Vector [1,2,3,4]) <+> (Vector [0,(-2),1])
<1,0,4,4>
Prelude MPL.LinearAlgebra.Vector> vAngle (Vector [1,1,1]) (Vector [0,1,0])
0.9553166181245092
Prelude MPL.LinearAlgebra.Vector> innerProd (Vector [1,2,3]) (Vector [4,5,6])
32
Prelude MPL.LinearAlgebra.Vector> scalarMult 3 (Vector [2,4,5,1,3])
<6,12,15,3,9>
Prelude MPL.LinearAlgebra.Vector> crossProd (Vector [1,1,1]) (Vector [3,4,5])
<1,-2,1>
Prelude MPL.LinearAlgebra.Vector> scalarTripleProd (Vector [1]) (Vector [0,1]) (Vector [0,0,1])
1
```

Fig. 16. Functions for Vectors in GHCi

9.6 Combinatorics

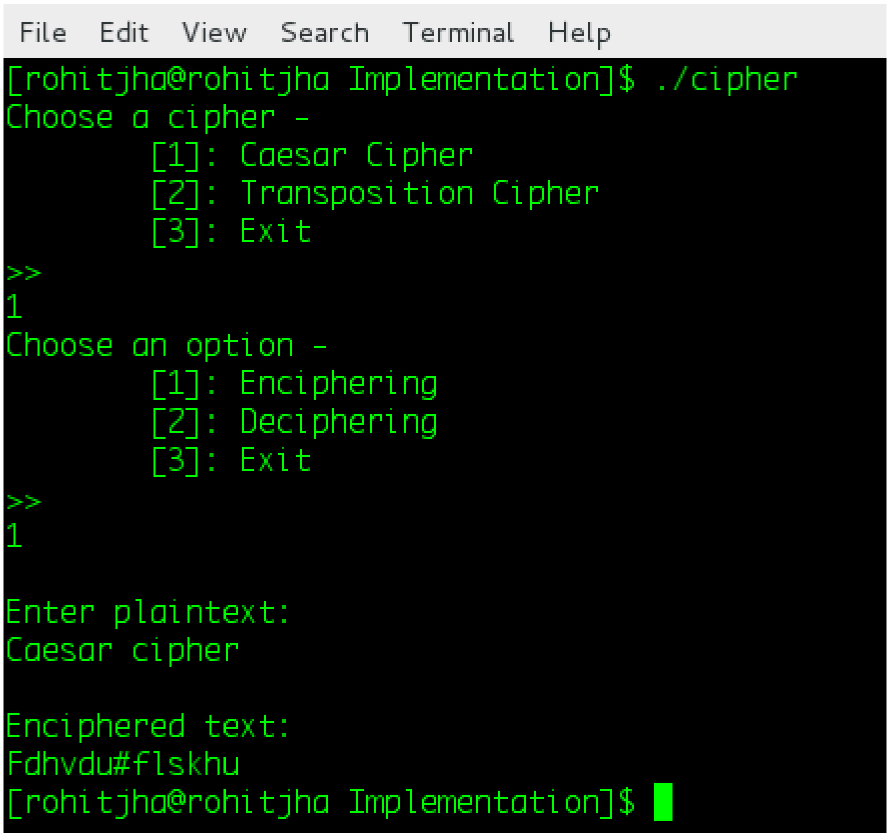


```
Prelude MPL.Combinatorics.Combinatorics> factorial 30
265252859812191058636308480000000
Prelude MPL.Combinatorics.Combinatorics> permutation [1,2,3]
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
Prelude MPL.Combinatorics.Combinatorics> shuffle [1..10]
[3,9,4,1,5,7,8,2,10,6]
Prelude MPL.Combinatorics.Combinatorics> shuffle [1..10]
[8,7,10,9,3,1,6,2,5,4]
Prelude MPL.Combinatorics.Combinatorics> combination 2 [["Heads"],["Tails"]]
[["Heads","Heads"],["Heads","Tails"],["Tails","Heads"],["Tails","Tails"]]
Prelude MPL.Combinatorics.Combinatorics> 10 `c` 5
252
```

Fig. 17. Functions for Combinatorics in GHCi

9.7 Applications

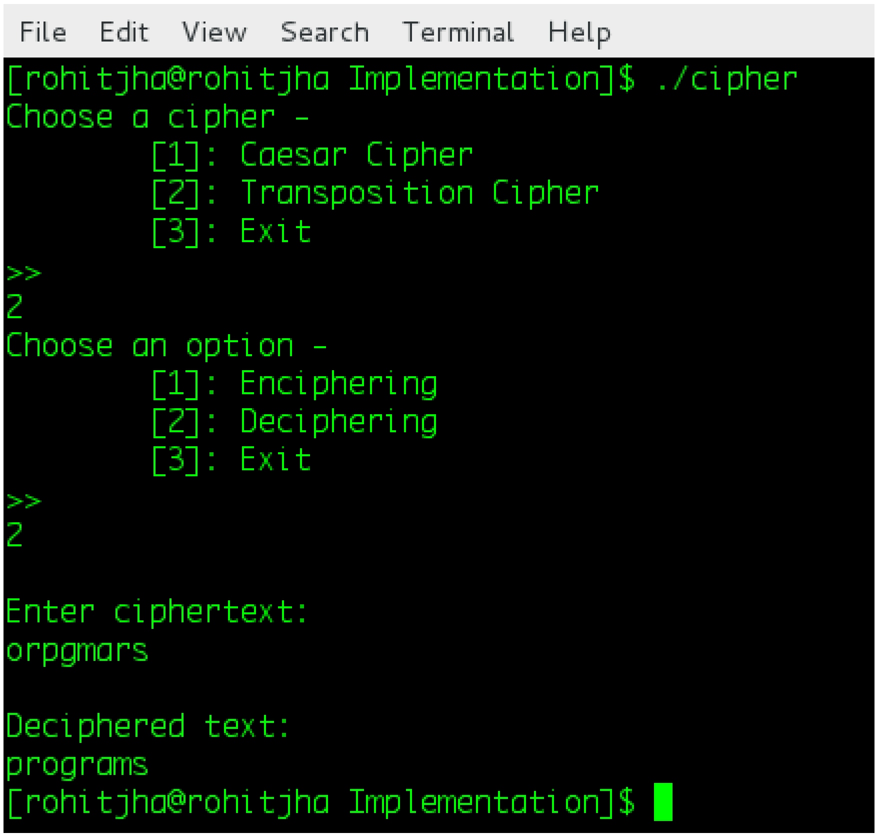
9.7.1 Caesar Cipher



```
[rohitjha@rohitjha Implementation]$ ./cipher
Choose a cipher -
        [1]: Caesar Cipher
        [2]: Transposition Cipher
        [3]: Exit
>>
1
Choose an option -
        [1]: Enciphering
        [2]: Deciphering
        [3]: Exit
>>
1

Enter plaintext:
Caesar cipher

Enciphered text:
Fdhvdu#flskhu
[rohitjha@rohitjha Implementation]$
```

Fig. 18. Enciphering using Caesar Cipher

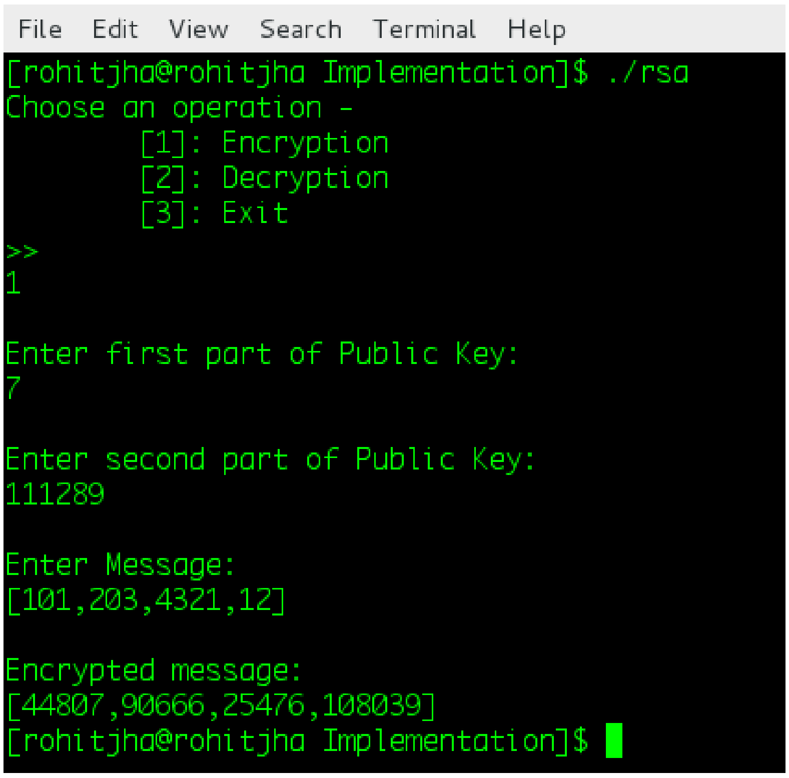
9.7.2 Transposition Cipher



```
[rohitjha@rohitjha Implementation]$ ./cipher
Choose a cipher -
        [1]: Caesar Cipher
        [2]: Transposition Cipher
        [3]: Exit
>>
2
Choose an option -
        [1]: Enciphering
        [2]: Deciphering
        [3]: Exit
>>
2

Enter ciphertext:
orpgmars

Deciphered text:
programs
[rohitjha@rohitjha Implementation]$
```

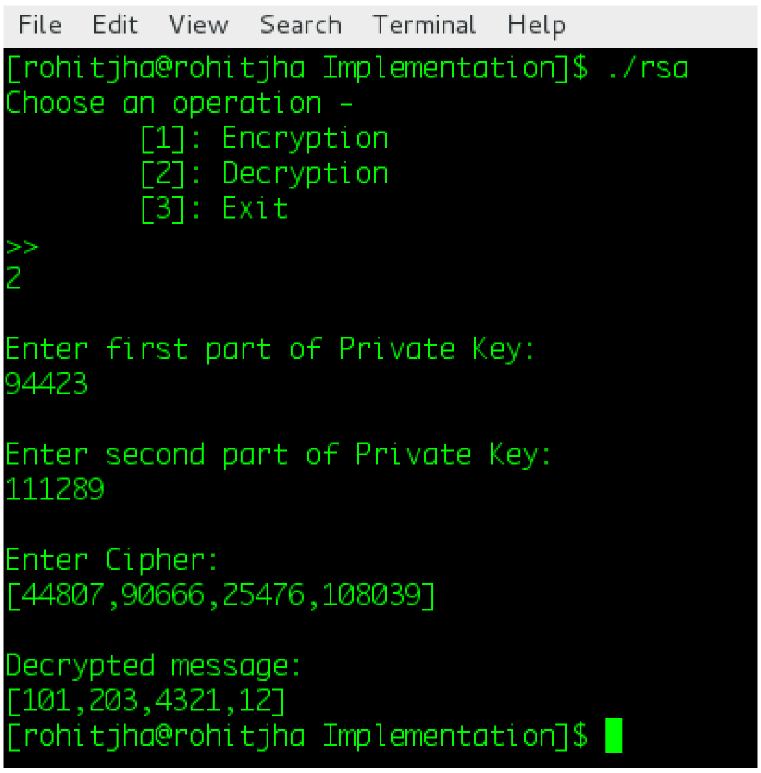
Fig. 19. Deciphering using Transposition Cipher

9.7.3 RSA Encryption and Decryption



```
[rohitjha@rohitjha Implementation]$ ./rsa
Choose an operation -
        [1]: Encryption
        [2]: Decryption
        [3]: Exit
>>
1

Enter first part of Public Key:
7

Enter second part of Public Key:
111289

Enter Message:
[101,203,4321,12]

Encrypted message:
[44807,90666,25476,108039]
[rohitjha@rohitjha Implementation]$
```

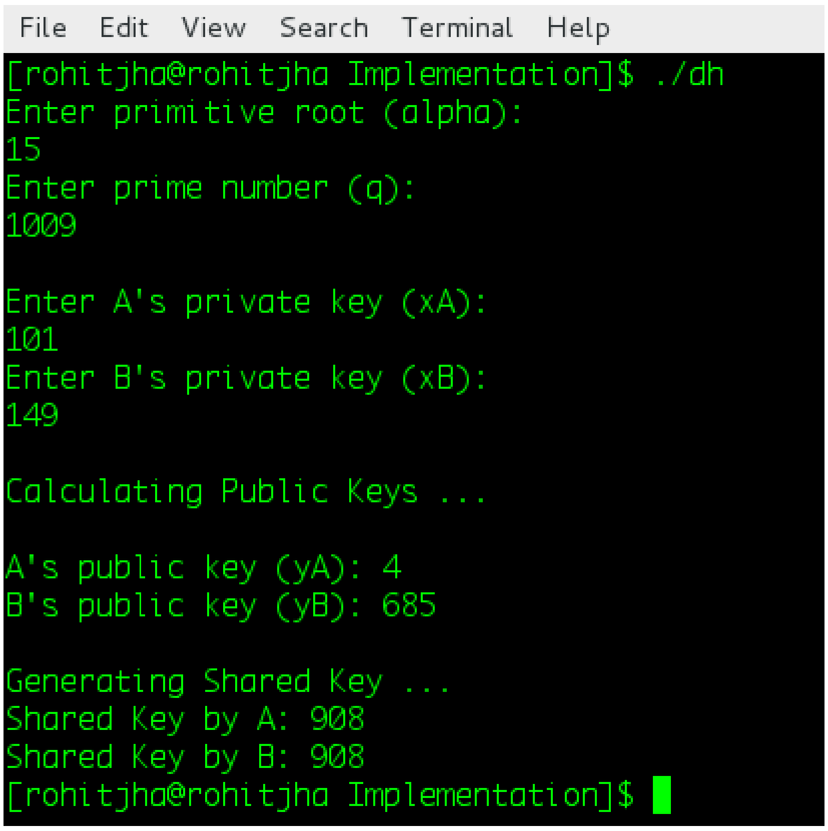
Fig. 20. Encryption using RSA



```
[rohitjha@rohitjha Implementation]$ ./rsa
Choose an operation -
        [1]: Encryption
        [2]: Decryption
        [3]: Exit
>>
2

Enter first part of Private Key:
94423

Enter second part of Private Key:
111289

Enter Cipher:
[44807,90666,25476,108039]

Decrypted message:
[101,203,4321,12]
[rohitjha@rohitjha Implementation]$
```

Fig. 21. Decryption using RSA

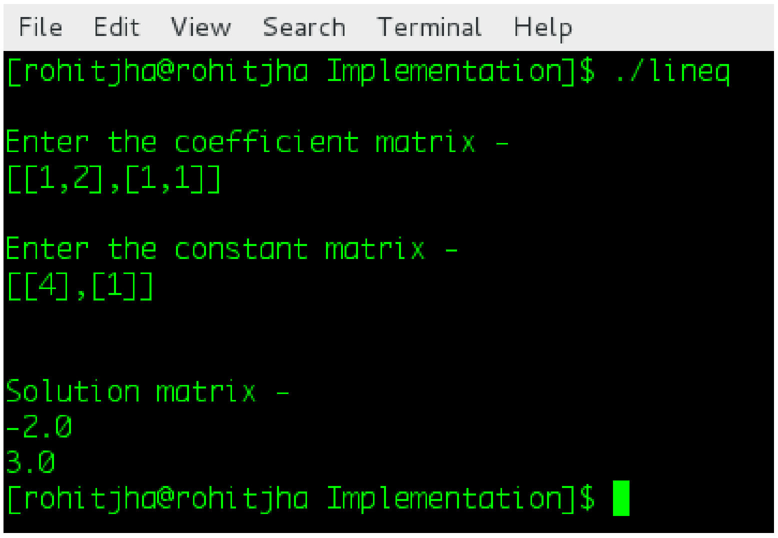
9.7.4 Diffie-Hellman Key Exchange Protocol



```
[rohitjha@rohitjha Implementation]$ ./dh
Enter primitive root (alpha):
15
Enter prime number (q):
1009

Enter A's private key (xA):
101
Enter B's private key (xB):
149

Calculating Public Keys ...

A's public key (yA): 4
B's public key (yB): 685

Generating Shared Key ...
Shared Key by A: 908
Shared Key by B: 908
[rohitjha@rohitjha Implementation]$
```

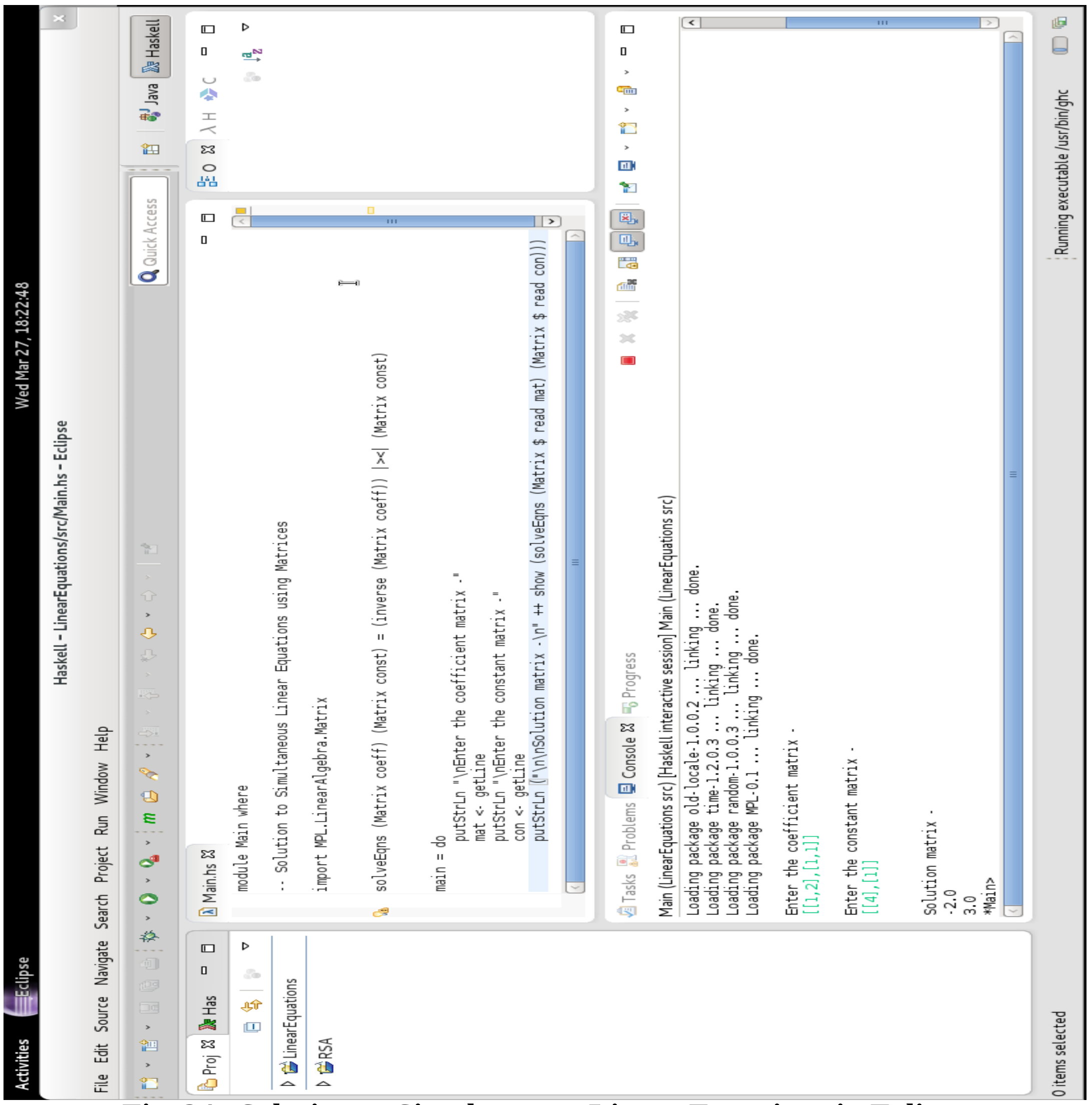
Fig. 22. Diffie-Hellman Key Exchange Protocol

9.7.5 Simultaneous Linear Equations



```
[rohitjha@rohitjha Implementation]$ ./lineq

Enter the coefficient matrix -
[[1,2],[1,1]]

Enter the constant matrix -
[[4],[1]]


Solution matrix -
-2.0
3.0
[rohitjha@rohitjha Implementation]$
```

Fig. 23. Solution to Simultaneous Linear Equations

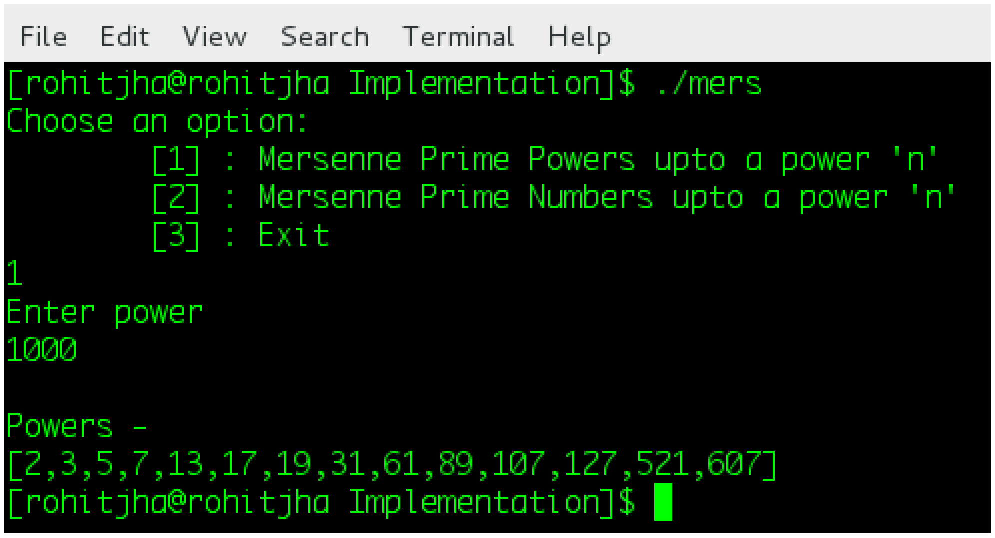


```haskell
module Main where

-- Solution to Simultaneous Linear Equations using Matrices

import MPL.LinearAlgebra.Matrix

solveEqns (Matrix coeff) (Matrix const) = (inverse (Matrix coeff)) |><| (Matrix const)

main = do
    putStrLn "\nEnter the coefficient matrix -"
    mat <- getLine
    putStrLn "\nEnter the constant matrix -"
    con <- getLine
    putStrLn ("\n\nSolution matrix -\n" ++ show (solveEqns (Matrix $ read mat) (Matrix $ read con)))
```

```
Main (LinearEquations src) [Haskell interactive session] Main (LinearEquations src)
Loading package old-locale-1.0.0.2 ... linking ... done.
Loading package time-1.2.0.3 ... linking ... done.
Loading package random-1.0.0.3 ... linking ... done.
Loading package MPL-0.1 ... linking ... done.

Enter the coefficient matrix -
[[1,2],[1,1]]

Enter the constant matrix -
[[4],[1]]


Solution matrix -
-2.0
3.0
*Main>
```

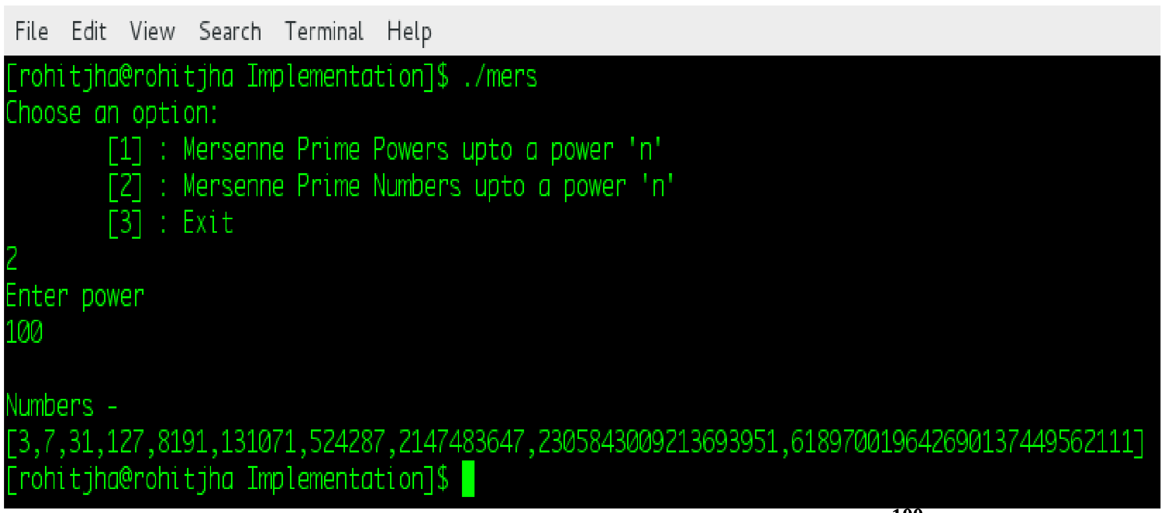
Fig. 24. Solution to Simultaneous Linear Equations in Eclipse

9.7.6 Mersenne Prime Numbers



```
[rohitjha@rohitjha Implementation]$ ./mers
Choose an option:
        [1] : Mersenne Prime Powers upto a power 'n'
        [2] : Mersenne Prime Numbers upto a power 'n'
        [3] : Exit
1
Enter power
1000

Powers -
[2,3,5,7,13,17,19,31,61,89,107,127,521,607]
[rohitjha@rohitjha Implementation]$
```

Fig. 25. List of Mersenne Prime Numbers' powers up to 1000



```
[rohitjha@rohitjha Implementation]$ ./mers
Choose an option:
        [1] : Mersenne Prime Powers upto a power 'n'
        [2] : Mersenne Prime Numbers upto a power 'n'
        [3] : Exit
2
Enter power
100

Numbers -
[3,7,31,127,8191,131071,524287,2147483647,2305843009213693951,618970019642690137449562111]
[rohitjha@rohitjha Implementation]$
```

Fig. 26. List of Mersenne Prime Numbers up to 2^{100}-1

10. CONCLUSION AND FUTURE SCOPE

The Mathematical Programming Language has been successfully implemented as a Preprocessed Domain-Specific Language (DSL) for Discrete Mathematics.

The DSL contains two major components – a library and the preprocessor. Haskell is the base language for the library and the preprocessor translates the DSL programs into equivalent Haskell programs, which are then compiled by GHC after importing the required library modules.This DSL is available as an installable package for all platforms on which Haskell is supported.

Since discrete mathematics is a vast area of study, it is not possible to include all topics in the library during the initial stages of development. In the future, modules for group theory, information theory, geometry, topology and theoretical computer science can be added. Additionally, the preprocessor can be constantly updated to handle new modules and new features in existing ones. Apart from this, based on feedback and suggestions from users, the syntax of this DSL can be improved to suit their needs.

11. REFERENCES

[1] A. Raja, D. Lakshmanan, “Domain Specific Languages”, International JournalofComputer Applications, vol 1, no. 21, 2010.

[2] W. Taha, “Domain Specific Languages”, IEEE International Conference on Computer Engineering and Systems (ICESS), 2008.

[3] D. Ghosh, “Part 1: Introducing Domain Specific Languages”, in *DSLs in Action*, Manning Publication Co., 2011, ch. 1, sec. 5, pp. 17-20.

[4] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Volkel, "Design Guidelines for Domain Specific Languages", Proc. *DSM 2009*, 2009.

[5] M.Mernik, J.Heering, A. M. Sloane, “When and How to Develop Domain-Specific Languages”, ACM Computing Surveys (CSUR), 2005.

[6] M. Fowler, “Domain-Specific Languages”, Addison-Wesley Professional, 2010.

[7] J. Hughes, "Why Functional Programming Matters", The Computer Journal, 1989.

[8] B. Goldberg, “Functional Programming Languages”, ACM Computing Surveys, Vol. 28, No. 1, March 1996.

[9] P. Hudak, "Building domain-specific embedded languages", ACM Computing Surveys, December 1996.

12. APPENDIX

12.1 DSL Library Modules

12.1.1 Mathematical Logic

The Haskell code for this module is:

```
module MPL.Logic.Logic
(
      and',
      or',
      xor,
      xnor,
      nand,
      nor,
      equals,
      implies,
      (/\),
      (\/),
      (==>),
      (<=>),
      notL,
      andL,
      orL,
      xorL,
      xnorL,
      nandL,
      norL
)
where

-- Binary XOR Function
xor :: Bool -> Bool -> Bool
xor a b
      | a == b = False
      | otherwise = True


-- Binary XNOR Funtion
xnor :: Bool -> Bool -> Bool
xnor a b = not (xor a b)


-- Binary NAND Function
nand :: Bool -> Bool -> Bool
nand a b = not (a && b)


-- Binary NOR Function
nor :: Bool -> Bool -> Bool
nor a b = not (a || b)


-- Binary Logical Equality
equals :: Bool -> Bool -> Bool
equals a b = a == b


-- Binary Logical Implication
implies :: Bool -> Bool -> Bool
implies a b
      | (a == True) && (b == False) = False
      | otherwise = True


-- Binary and Operator
(/\) :: Bool -> Bool -> Bool
a /\ b = a && b


-- Binary or Operator
(\/) :: Bool -> Bool -> Bool
a \/ b = a || b


-- Binary implication Operator
(==>) :: Bool -> Bool -> Bool
a ==> b = implies a b


-- Binary equality Operator
(<=>) :: Bool -> Bool -> Bool
a <=> b = a == b


-- unary not Operator on a list of Bool
notL :: [Bool] -> [Bool]
notL a = map not a


-- Binary and Operator on a list of Bool
andL :: [Bool] -> Bool
andL a = foldl1 (&&) a


-- Binary or Operator on a list of Bool
orL :: [Bool] -> Bool
orL a = foldl1 (||) a


-- Binary xor Operator on a list of Bool
xorL :: [Bool] -> Bool
xorL a = foldl1 (xor) a


-- Binary nand Operator on a list of Bool
nandL :: [Bool] -> Bool
nandL a = foldl1 (nand) a


-- Binary nor Operator on a list of Bool
```

```
norL :: [Bool] -> Bool
norL a = foldl1 (nor) a


-- Binary xnor Operator on a list of Bool
xnorL :: [Bool] -> Bool
xnorL a = foldl1 (xnor) a
```

12.1.2 Sets

The Haskell code for this module is:

```
module MPL.SetTheory.Set
(
      Set(..),
      set2list,
      union,
      unionL,
      intersection,
      intersectionL,
      difference,
      isMemberOf,
      cardinality,
      isNullSet,
      isSubset,
      isSuperset,
      powerSet,
      cartProduct,
      disjoint,
      disjointL,
      natural,
      natural',
      whole,
      whole',
      sMap
)
where


-- Union of Sets
union :: Ord a => Set a -> Set a -> Set a
union (Set set1) (Set set2)
      = Set ((L.sort . L.nub) (set1 ++ [e | e <- set2, not (elem e
set1)]))


-- Union of a list of Sets
unionL :: Ord a => [Set a] -> Set a
unionL s = foldl1 (union) s


-- Intersection of Sets
intersection :: Ord a => Set a -> Set a -> Set a
intersection (Set set1) (Set set2)
      = Set ((L.sort . L.nub) [e | e <- set1, (elem e set2)])
```

```
-- Intersection of a list of Sets
intersectionL :: Ord a => [Set a] -> Set a
intersectionL s = foldl1 (intersection) s


-- Set difference
difference :: Ord a => Set a -> Set a -> Set a
difference (Set set1) (Set set2)
     = Set ((L.sort . L.nub) [e | e <- set1, not (elem e set2)])


-- Membership
isMemberOf :: Eq a => Element a -> Set a -> Bool
isMemberOf a (Set []) = False
isMemberOf a (Set set) = a `elem` set


-- Cardinality
cardinality :: Eq a => Set a -> Int
cardinality (Set set) = (L.length . L.nub) set


-- Empty/Null Set verification
isNullSet :: Eq a => Set a -> Bool
isNullSet (Set set)
     | cardinality (Set set) == 0 = True
     | otherwise = False


-- Subset verification
isSubset :: Ord a => Set a -> Set a -> Bool
isSubset (Set set1) (Set set2) = null [e | e <- set1', not (elem e
set2')]
     where set1' = (L.sort . L.nub) set1
           set2' = (L.sort . L.nub) set2


-- Superset verification
isSuperset :: Ord a => Set a -> Set a -> Bool
isSuperset (Set set1) (Set set2) = null [e | e <- set2', not (elem e
set1')]
     where set1' = (L.sort . L.nub) set1
           set2' = (L.sort . L.nub) set2


-- Power set
powerSet :: Ord a => Set a -> Set (Set a)
powerSet (Set s) = Set $ map (\xs -> (Set xs)) (L.subsequences $
set2list (Set s))


powerSet' :: Ord a => Set a -> Set (Set a)
powerSet' (Set xs) = Set $ L.sort (map (\xs -> (Set xs)) (powerList
xs))
```

```
powerList :: Ord a => [a] -> [[a]]
--powerList :: [a] -> [[a]]
powerList [] = [[]]
powerList (x:xs) = L.sort $ (powerList xs) ++ (map (x:) (powerList
xs))


-- Cartesian product
cartProduct :: Ord a => Set a -> Set a -> [(Element a,Element a)]
cartProduct (Set set1) (Set set2) = [(x,y) | x <- set1', y <- set2']
      where set1' = (L.sort . L.nub) set1
            set2' = (L.sort . L.nub) set2


-- Checking if two sets are disjoint
disjoint :: Ord a => Set a -> Set a -> Bool
disjoint (Set set1) (Set set2) = isNullSet $ intersection (Set set1)
(Set set2)


-- Checking if all Sets in a list are disjoint
disjointL :: Ord a => [Set a] -> Bool
disjointL s = isNullSet $ intersectionL s


-- Set of natural numbers
natural = [1,2..]


-- Set of natural numbers upto n
natural' n = [1,2..n]


-- Set of whole numbers
whole = [0,1..]


-- Set of whole numbers upto n
whole' n = [0,1..n]


-- Mapping a function to a Set
sMap f (Set s) = list2set $ map f s
```

12.1.3 Relations

```
module MPL.SetTheory.Relation
(
      Relation(..),
      relation2list,
      getFirst,
      getSecond,
      elemSet,
      returnFirstElems,
      returnSecondElems,
      isReflexive,
      isIrreflexive,
      isSymmetric,
      isAsymmetric,
      isAntiSymmetric,
      isTransitive,
      rUnion,
      rUnionL,
      rIntersection,
      rIntersectionL,
      rDifference,
      rComposite,
      rPower,
      reflClosure,
      symmClosure,
      tranClosure,
      isEquivalent,
      isWeakPartialOrder,
      isWeakTotalOrder,
      isStrictPartialOrder,
      isStrictTotalOrder
)
where


import qualified Data.List as L


-- Relation data type
newtype Relation a = Relation [(a,a)] deriving (Eq)

instance (Show a) => Show (Relation a) where
      showsPrec _ (Relation s) str = showRelation s str

showRelation [] str = showString "{}" str
showRelation (x:xs) str = showChar '{' (shows x (showl xs str))
      where
            showl [] str = showChar '}' str
            showl (x:xs) str = showChar ',' (shows x (showl xs str))


-- Converting a relation to list
relation2list (Relation r) = r


elemSet r = L.union (getFirst (Relation r)) (getSecond (Relation r))


-- Returns list of all 'a' where (a,b) <- Relation
getFirst (Relation r) = L.nub [fst x | x <- r]


-- Returns list of all 'b' where (a,b) <- Relation
getSecond (Relation r) = L.nub [snd x | x <- r]
```

```
-- Returns list of all 'a' where (a,b) <- Relation and 'b' is
specified
returnFirstElems (Relation r) x = L.nub [fst (a,x) | a <- getFirst
(Relation r), (a,x) `elem` r]


-- Returns list of all 'b' where (a,b) <- Relation and 'a' is
specified
returnSecondElems (Relation r) x = L.nub [snd (x,b) | b <- getSecond
(Relation r), (x,b) `elem` r]


-- Checks if a relation is reflexive or not
isReflexive (Relation r) = and [(a,a) `elem` r | a <- elemSet r]


isIrreflexive (Relation r) = not $ isReflexive (Relation r)


-- Checks if a relation is symmetric or not
isSymmetric (Relation r) = and [((b,a) `elem` r) | a <- elemSet r, b
<- elemSet r, (a,b) `elem` r]


-- Checks if a relation is asymmetric or not
isAsymmetric (Relation r) = and [not ((b,a) `elem` r) | a <- elemSet
r, b <- elemSet r, (a,b) `elem` r]


isAntiSymmetric (Relation r) = and [ a==b | a <- elemSet r, b <-
elemSet r, (a,b) `elem` r, (b,a) `elem` r]


-- Checks if a relation is transitive or not

isTransitive (Relation r) = and [(a,c) `elem` r | a <- elemSet r, b
<- elemSet r, c <- elemSet r, (a,b) `elem` r, (b,c) `elem` r]


-- Returns union of two relations
rUnion (Relation r1) (Relation r2) = Relation ((L.sort . L.nub) (r1
++ [e | e <- r2, not (elem e r1)]))


-- Returns union of list of relations
rUnionL r = foldl1 (rUnion) r


-- Returns intersection of two relations

rIntersection (Relation r1) (Relation r2) = Relation ((L.sort .
L.nub) [e | e <- r1, (elem e r2)])
```

```
-- Returns intersection of a list of relations
rIntersectionL r = foldl1 (rIntersection) r


-- Returns difference of two relations
rDifference (Relation r1) (Relation r2) = Relation ((L.sort . L.nub)
[e | e <- r1, not (elem e r2)])


-- Returns composite of two relations
rComposite (Relation r1) (Relation r2) = Relation $ L.nub [(a,c) | a
<- elemSet r1, b <- elemSet r1, b <- elemSet r2, c <- elemSet r2,
(a,b) `elem` r1, (b,c) `elem` r2]


-- Returns power of a relation
rPower (Relation r) pow =
      if (pow == (-1))
      then Relation [(b,a) | (a,b) <- r]
      else
            if (pow == 1)
            then (Relation r)
            else rComposite (rPower (Relation r) (pow-1)) (Relation
r)


-- Reflexive closure
reflClosure (Relation r) = rUnion (Relation r) (delta (Relation r))
      where
            delta (Relation r) = Relation [(a,b) | a <- elemSet r, b
<- elemSet r, a == b]


-- Symmetric closure
symmClosure (Relation r) = rUnion (Relation r) (rPower (Relation r)
(-1))


-- Transitive closure
tranClosure (Relation r) = foldl1 (rUnion) [ (rPower (Relation r) n)
| n <- [1 .. length (elemSet r) ]]


isEquivalent (Relation r) = isReflexive (Relation r) && isSymmetric
(Relation r) && isTransitive (Relation r)


isWeakPartialOrder (Relation r) = isReflexive (Relation r) &&
isAntiSymmetric (Relation r) && isTransitive (Relation r)


isWeakTotalOrder (Relation r) = isWeakPartialOrder (Relation r) &&
(and [ ((a,b) `elem` r) || ((b,a) `elem` r) | a <- elemSet r, b <-
elemSet r ] )
```

```
isStrictPartialOrder (Relation r) = isIrreflexive (Relation r) &&
isAsymmetric (Relation r) && isTransitive (Relation r)
```

```
isStrictTotalOrder (Relation r) = isStrictPartialOrder (Relation r)
&& (and [ ((a,b) `elem` r) || ((b,a) `elem` r) || a==b | a <-
elemSet r, b <- elemSet r ] )
```

12.1.4 Graphs

```
module MPL.GraphTheory.Graph
(
      Vertices(..),
      vertices2list,
      Edges(..),
      edges2list,
      first,
      second,
      third,
      Graph(..),
      GraphMatrix(..),
      graph2matrix,
      getVerticesG,
      getVerticesGM,
      numVerticesG,
      numVerticesGM,
      getEdgesG,
      getEdgesGM,
      numEdgesG,
      numEdgesGM,
      convertGM2G,
      convertG2GM,
      gTransposeG,
      gTransposeGM,
      isUndirectedG,
      isUndirectedGM,
      isDirectedG,
      isDirectedGM,
      unionG,
      unionGM,
      addVerticesG,
      addVerticesGM,
      verticesInEdges,
      addEdgesG,
      addEdgesGM,
      areConnectedGM,
      numPathsBetweenGM,
      adjacentNodesG,
      adjacentNodesGM,
      inDegreeG,
      inDegreeGM,
      outDegreeG,
      outDegreeGM,
      degreeG,
      degreeGM,
      hasEulerCircuitG,
      hasEulerCircuitGM,
      hasEulerPathG,
      hasEulerPathGM,
      hasHamiltonianCircuitG,
      hasHamiltonianCircuitGM,
      countOddDegreeV,
      countEvenDegreeV,
      hasEulerPathNotCircuitG,
      hasEulerPathNotCircuitGM,
      isSubgraphG,
      isSubgraphGM
)
where

import qualified Data.List as L


-- Data type for vertices
newtype Vertices a = Vertices [a] deriving (Eq)

instance (Show a) => Show (Vertices a) where
      showsPrec _ (Vertices s) str = showVertices s str

showVertices [] str = showString "{}" str
showVertices (x:xs) str = showChar '{' (shows x (showl xs str))
      where
            showl [] str = showChar '}' str
            showl (x:xs) str = showChar ',' (shows x (showl xs str))

vertices2list (Vertices v) = v


-- Data types for edges
newtype Edges a = Edges [(a,a,Int)] deriving (Eq)

instance (Show a) => Show (Edges a) where
      showsPrec _ (Edges s) str = showEdges s str

showEdges [] str = showString "{}" str
showEdges (x:xs) str = showChar '{' (shows x (showl xs str))
      where
            showl [] str = showChar '}' str
            showl (x:xs) str = showChar ',' (shows x (showl xs str))

edges2list (Edges a) = a

first (a,b,c) = a
second (a,b,c) = b
third (a,b,c) = c


-- Data type for Graph
newtype Graph a = Graph (Vertices a, Edges a) deriving (Eq, Show)


-- Data type for Graph as matrix
newtype Matrix a = Matrix [[a]] deriving (Eq)

instance Show a => Show (Matrix a) where
      show (Matrix a) = L.intercalate "\n" $ map (L.intercalate "\t"
. map show) a


newtype GraphMatrix a = GraphMatrix [[a]] deriving (Eq)

instance Show a => Show (GraphMatrix a) where
      show (GraphMatrix a) = L.intercalate "\n" $ map (L.intercalate
"\t" . map show) a

graph2matrix (GraphMatrix gm) = gm


-- Get vertices of a Graph
--getVertices :: Num a => Graph a -> Vertices a
getVerticesG (Graph g) = fst g


-- Number of vertices of Graph
--numVertices :: Num a => Graph a -> Integer
numVerticesG (Graph g) = fromIntegral $ length $ vertices2list $
getVerticesG (Graph g)


-- Number of edges of a Graph
numEdgesG (Graph g) = fromIntegral $ length $ edges2list $ getEdgesG
(Graph g)
```

```
-- Get edges of a Graph
--getEdges :: Num a => Graph a -> Edges a
getEdgesG (Graph g) = snd g


-- Get vertices of a GraphMatrix
--getVerticesGM :: Num a => GraphMatrix a -> Vertices a
getVerticesGM (GraphMatrix gm) = Vertices [1 .. fromIntegral $
length gm]


-- Get number of vertices of GraphMatrix
numVerticesGM (GraphMatrix gm) = length gm


-- Find weight of edge between nodes i and j
weight (GraphMatrix gm) i j = fromIntegral $ ((graph2matrix
(GraphMatrix gm))!!i)!!j


-- Get edges of a GraphMatrix
--getEdgesGM :: Num a => GraphMatrix a -> Edges a
getEdgesGM (GraphMatrix gm) = Edges [(i+1,j+1,(w i j)) | i <- [0 ..
fromIntegral $ ((length (graph2matrix (GraphMatrix gm)))-1)], j <-
[0 .. fromIntegral $ ((length (graph2matrix

(GraphMatrix gm)))-1)], ((weight (GraphMatrix gm) i j) /= 0)]
     where
          w i j = fromIntegral $ weight (GraphMatrix gm) i j


-- Number of edges in a GraphMatrix
numEdgesGM (GraphMatrix gm) = fromIntegral $ length $ edges2list $
getEdgesGM (GraphMatrix gm)


-- Convert GraphMatrix to Graph
--convertGM2G :: Num a => GraphMatrix a -> Graph a
convertGM2G (GraphMatrix gm) = Graph ((getVerticesGM (GraphMatrix
gm)), (getEdgesGM (GraphMatrix gm)))



-- Convert Graph to adjacency GraphMatrix
--convertG2GM' :: Num a => Graph a -> [a]
getLastVertex (Graph g) = (L.reverse $ L.sort $ vertices2list $
getVerticesG (Graph g)) !! 0
```

```
--convertG2GM' (Graph g) = [(f i j) | i <- [1 .. (numVerticesG (Graph g))], j <- [1 .. (numVerticesG (Graph g))]]
convertG2GM' (Graph g) = [(f i j) | i <- vertices2list $ (getVerticesG (Graph g)), j <- vertices2list $ (getVerticesG (Graph g))]
     where
          edgeList = [(first e, second e) | e <- edges2list (getEdgesG (Graph g))]

          f i j =
               if ((i),(j)) `elem` edgeList
               then third (w i j (Graph g))
               else 0

          w i j (Graph g) =
               [(i,j,k) | k <- [0 .. (maxWeight (Graph g))], (i,j,k) `elem` (edges2list (snd g))] !! 0

          maxWeight (Graph g) = fromIntegral $ ((L.reverse . L.sort) [third x | x <- edges2list (snd g)]) !! 0


chunk' n = takeWhile (not.null) . map (take n) . iterate (drop n)


convertG2GM (Graph g) = GraphMatrix $ chunk' (numVerticesG (Graph g)) (convertG2GM' (Graph g))


-- Transpose of a graph (GraphMatrix)
gTransposeGM (GraphMatrix []) = (GraphMatrix [])
gTransposeGM (GraphMatrix [[]]) = (GraphMatrix [[]])
gTransposeGM (GraphMatrix xs) = GraphMatrix $ foldr (zipWith (:)) (repeat []) xs


-- Transpose of a graph (Graph)
gTransposeG (Graph g) = convertGM2G $ gTransposeGM $ (convertG2GM (Graph g))


-- Checking if a GraphMatrix is undirected
isUndirectedGM (GraphMatrix gm) = (GraphMatrix gm) == gTransposeGM (GraphMatrix gm)


-- Checking if a Graph is undirected
```

```
isUndirectedG (Graph g) = isUndirectedGM (convertG2GM (Graph g))


-- Checking if a GraphMatrix is directed
isDirectedGM (GraphMatrix gm) = not $ isUndirectedGM (GraphMatrix
gm)


-- Checking if a Graph is directed
isDirectedG (Graph g) = isDirectedGM $ (convertG2GM (Graph g))


-- Union of Graphs
unionG (Graph g1) (Graph g2) = Graph (
     Vertices $ L.sort (L.union (vertices2list $ getVerticesG
(Graph g1)) (vertices2list $ getVerticesG (Graph g2))),
     Edges $ L.sort (L.union (edges2list $ getEdgesG (Graph g1))
(edges2list $ getEdgesG (Graph g2)))
     )


-- Union of GraphMatrices
unionGM (GraphMatrix gm1) (GraphMatrix gm2) = convertG2GM $ (unionG
(convertGM2G (GraphMatrix gm1)) (convertGM2G (GraphMatrix gm2)))


-- Adding vertices to Graph
addVerticesG (Graph g) (Vertices v) = Graph (
     Vertices (L.union (vertices2list $ getVerticesG (Graph g))
(vertices2list $ Vertices v)),
     getEdgesG (Graph g))


-- Adding vertices to GraphMatrix
addVerticesGM (GraphMatrix gm) (Vertices v) = convertG2GM $
addVerticesG (convertGM2G (GraphMatrix gm)) (Vertices v)


-- Extracting all vertices in Edges
verticesInEdges (Edges e) = L.union (L.nub [first edge | edge <-
edges2list (Edges e)]) (L.nub [second edge | edge <- edges2list
(Edges e)])


-- Adding edges to Graph
addEdgesG (Graph g) (Edges e) =
     if (and [v `elem` vertices2list (getVerticesG (Graph g)) | v
<- (verticesInEdges (Edges e))])
```

```
      then Graph (
      getVerticesG (Graph g),
      Edges (L.union (edges2list $ getEdgesG (Graph g)) (edges2list
$ Edges e))
      )
      else error "Vertices in the edge(s) are not in the graph's set
of vertices."


-- Adding edges to GraphMatrix
addEdgesGM (GraphMatrix gm) (Edges e) = convertG2GM $ addEdgesG
(convertGM2G (GraphMatrix gm)) (Edges e)


-- Checking if two vertices in GraphMatrix are connected
areConnectedGM (GraphMatrix g) (Vertices v1) (Vertices v2) =
      if ((mat2list' $ (mPower' (Matrix $ graph2matrix (GraphMatrix
g)) (numVerticesGM (GraphMatrix g)))) !! ((v1!!0)-1) !! ((v2!!0))-1)
/= 0
      then True
      else False


-- Finding number of paths between two vertices in a GraphMatrix
numPathsBetweenGM (GraphMatrix g) (Vertices v1) (Vertices v2) =
      (((mat2list' $ (mPower' (Matrix $ graph2matrix (GraphMatrix
g)) (numVerticesGM (GraphMatrix g)))) !! (((vertices2list (Vertices
v1))!!0) - 1)) !! (((vertices2list (Vertices v2))!!0)

- 1))


-- Finding nodes adjacent to a node in a Graph
adjacentNodesG (Graph g) (Vertices v) = Vertices $ L.union [ second
x | x <- edges2list $ getEdgesG (Graph g), (first x) == (v!!0) ] [
first y | y <- edges2list $ getEdgesG (Graph g),

(second y) == (v!!0) ]


-- Finding nodes adjacent to a node in a GraphMatrix
adjacentNodesGM (GraphMatrix gm) (Vertices v) = adjacentNodesG
(convertGM2G (GraphMatrix gm)) (Vertices v)


-- In-degree of a vertex in a directed Graph
inDegreeG (Graph g) (Vertices v) = length $ [ first y | y <-
edges2list $ getEdgesG (Graph g), (second y) == (v!!0) ]
```

```
-- In-degree of a vertex in a directed GraphMatrix
inDegreeGM (GraphMatrix gm) (Vertices v) = inDegreeG (convertGM2G
(GraphMatrix gm)) (Vertices v)


-- Out-degree of a vertex in a directed Graph
outDegreeG (Graph g) (Vertices v) = length $ [ second y | y <-
edges2list $ getEdgesG (Graph g), (first y) == (v!!0) ]


-- Out-degree of a vertex in a directed GraphMatrix
outDegreeGM (GraphMatrix gm) (Vertices v) = outDegreeG (convertGM2G
(GraphMatrix gm)) (Vertices v)


-- Degree of a vertex in an undirected Graph
degreeG (Graph g) (Vertices v) = (inDegreeG (Graph g) (Vertices v))
+ (outDegreeG (Graph g) (Vertices v))


-- Degree of a vertex in an undirected GraphMatrix
degreeGM (GraphMatrix gm) (Vertices v) = (inDegreeGM (GraphMatrix
gm) (Vertices v)) + (outDegreeGM (GraphMatrix gm) (Vertices v))


-- Finding if a Graph contains a Euler Circuit
hasEulerCircuitG (Graph g) = and [ even $ (degreeG (Graph g)
(Vertices [v])) | v <- vertices2list $ getVerticesG (Graph g)]


-- Finding if a GraphMatrix contains a Euler Circuit
hasEulerCircuitGM (GraphMatrix gm) = hasEulerCircuitG (convertGM2G
(GraphMatrix gm))


-- Finding if a Graph contains a Euler Path
hasEulerPathG (Graph g) = hasEulerCircuitG (Graph g)


-- Finding if a GraphMatrix contains a Euler Path
hasEulerPathGM (GraphMatrix gm) = hasEulerCircuitGM (GraphMatrix gm)


-- Finding number of vertices with odd degree
countOddDegreeV (Graph g) = sum [ 1 | v <- vertices2list $
(getVerticesG (Graph g)), odd $ (degreeG (Graph g) (Vertices [v])) ]
```

```
-- Finding number of vertices with even degree
countEvenDegreeV (Graph g) = sum [ 1 | v <- vertices2list $
(getVerticesG (Graph g)), even $ (degreeG (Graph g) (Vertices [v]))
]


-- Finding if a Graph conatains a Euler Path but not a Euler circuit
hasEulerPathNotCircuitG (Graph g) = countOddDegreeV (Graph g) == 2


-- Finding if a GraphMatrix contains a Euler Path but not a Euler
circuit
hasEulerPathNotCircuitGM (GraphMatrix gm) = hasEulerPathNotCircuitG
(convertGM2G (GraphMatrix gm))


-- Finding if a Graph contains a Hamiltonian Circuit
hasHamiltonianCircuitG (Graph g) = and [(degreeG (Graph g) (Vertices
[v])) >= ((numVerticesG (Graph g)) `div` 2) | v <- vertices2list $
getVerticesG (Graph g), (numVerticesG (Graph g)) >= 3]


-- Finding if a GraphMatrix contains a Hamiltonian Circuit
hasHamiltonianCircuitGM (GraphMatrix gm) = hasHamiltonianCircuitG
(convertGM2G (GraphMatrix gm))


-- Checking if a Graph is a subgraph
isSubgraphG (Graph g1) (Graph g2) = (e1 `isSubset` e2) && (v1
`isSubset` v2)
      where
           isSubset set1 set2 = null [e | e <- (L.sort . L.nub)
set1, not (elem e ((L.sort . L.nub) set2))]
           e1 = edges2list $ getEdgesG (Graph g1)
           e2 = edges2list $ getEdgesG (Graph g2)
           v1 = vertices2list $ getVerticesG (Graph g1)
           v2 = vertices2list $ getVerticesG (Graph g2)


-- Checking if a GraphMatrix is a subgraph
isSubgraphGM (GraphMatrix gm1) (GraphMatrix gm2) = isSubgraphG
(convertGM2G $ (GraphMatrix gm1)) (convertGM2G $ (GraphMatrix gm2))
```

12.1.5 Prime Numbers

```
module MPL.NumberTheory.Primes
(
      primesTo,
      primesBetween,
      nPrimes,
      primesTo100,
      trialDivision,
      primesTo10000,
      isTrialDivisionPrime,
      isStrongPseudoPrime,
      isMillerRabinPrime,
      isPrime,
      nextPrime,
      primeFactors
)
where


-- internal functions ----------------------
d `divides` n = n `mod` d == 0


n `splitWith` p = doSplitWith 0 n
      where doSplitWith s t
            | p `divides` t = doSplitWith (s+1) (t `div` p)
            | otherwise     = (s, t)


power (idG,multG) x n = doPower idG x n
      where
            doPower y _ 0 = y
            doPower y x n =
                  let y' = if odd n then (y `multG` x) else y
                      x' = x `multG` x
                      n' = n `div` 2
                  in doPower y' x' n'


minus (x:xs) (y:ys) = case (compare x y) of
            LT -> x : minus  xs  (y:ys)
            EQ ->     minus  xs     ys
            GT ->     minus (x:xs)  ys
minus  xs     _     = xs


primePowerFactors :: Integer -> [(Integer,Int)]
primePowerFactors n | n > 0 = takeOutFactors n primesTo10000
      where
            takeOutFactors n (p:ps)
                  | p*p > n   = finish n
                  | otherwise =
                        let (s,n') = n `splitWith` p
```

```
                    in if s > 0 then (p,s) : takeOutFactors n' ps
else takeOutFactors n ps
          takeOutFactors n [] = finish n
          finish 1 = []
          finish n =
               if n < 100000000 || isMillerRabinPrime n
               then [(n,1)]
               else error ("primePowerFactors: unable to factor "
++ show n)


sieve (p:xs)
      | p*p > (last xs)   = p : xs
      | otherwise = p : sieve (xs `minus` [p*p, p*p+2*p..])
------------------------------------------


-- Generate list of primes upto specified limit by using Sieve of
Eratosthenes
primesTo :: Integer -> [Integer]
primesTo 0 = []
primesTo 1 = []
primesTo 2 = [2]
primesTo m = 2 : 3 : sieve [3,5..m]


-- Generate all primes between two numbers (upper limit inclusive)
primesBetween :: Integer -> Integer -> [Integer]
primesBetween m n
     | (m <= 2) = primesTo n
     | otherwise = (primesTo n) `minus` (primesTo m)--(nextPrime m)
: sieve [((nextPrime m)+1) .. n]


-- Generate list of first 'n' primes
nPrimes n = take n (sieve [2..])
     where sieve (p:ns) = p : sieve (filter (notdiv p) ns)
           notdiv p n = n `mod` p /= 0


-- List of prime numbers under 100
primesTo100 :: [Integer]
primesTo100                                                  =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89
,97]


-- Trial division
trialDivision ps n = doTrialDivision ps
     where doTrialDivision (p:ps) = let (q,r) = n `quotRem` p in if
r == 0 then False else if q < p then True else doTrialDivision ps
           doTrialDivision [] = True


-- List of prime numbers under 10000
```

```
primesTo10000 = primesTo100 ++ filter (trialDivision primesTo100)
[101,103..9999]


-- Determine primality using trial division
isTrialDivisionPrime 2 = True
isTrialDivisionPrime n = trialDivision (primesTo10000 ++
[10001,10003..]) n


-- Check if number is a pseudo-prime (probable)
isStrongPseudoPrime :: Integer -> (Int,Integer) -> Integer -> Bool
isStrongPseudoPrime n (s,t) b =
      let b' = power (1, \x y -> x*y `mod` n) b t
      in if b' == 1 then True else doSquaring s b'
      where
            doSquaring 0 x = False
            doSquaring s x
                  | x == n-1  = True
                  | x == 1    = False
                  | otherwise = doSquaring (s-1) (x*x `mod` n)


-- Check if number is prime using Miller-Rabin primality test
isMillerRabinPrime :: Integer -> Bool
isMillerRabinPrime n
      | n < 100   = n `elem` primesTo100
      | otherwise = all (isStrongPseudoPrime n (s,t)) primesTo100
            where (s,t) = (n-1) `splitWith` 2


-- Primality Checking which uses appropriate test according to the
given number
isPrime :: Integer -> Bool
isPrime n
      | n < 2          = False
      | n < 500000000  = isTrialDivisionPrime n
      | n >= 500000000 = isMillerRabinPrime n


-- Generate the next prime greater than or equal to the given number
nextPrime :: Integer -> Integer
nextPrime n = head [p | p <- [n..], isPrime p]


-- Prime factorization of a number
primeFactors :: Integer -> [Integer]
primeFactors n = concat (map (\(p,a) -> replicate a p)
(primePowerFactors n))
```

12.1.6 Matrices

```
module MPL.LinearAlgebra.Matrix
(
      Matrix(..),
      mAdd,
      mAddL,
      (|+|),
      mSub,
      (|-|),
      mTranspose,
      mScalarMult,
      (|*|),
      mMult,
      mMultL,
      (|><|),
      numRows,
      numCols,
      mat2list,
      determinant,
      inverse,
      mDiv,
      (|/|),
      extractRow,
      extractCol,
      extractRowRange,
      extractColRange,
      mPower,
      trace,
      isInvertible,
      isSymmetric,
      isSkewSymmetric,
      isRow,
      isColumn,
      isSquare,
      isOrthogonal,
      isInvolutive,
      isZeroOne,
      isZero,
      isOne,
      isUnit,
      mMap,
      zero,
      zero',
      one,
      one',
      unit
)
where

import qualified Data.List as L


newtype Matrix a = Matrix [[a]] deriving (Eq)
```

```
instance Show a => Show (Matrix a) where
      show (Matrix a) = L.intercalate "\n" $ map (L.intercalate "\t"
. map show) a


-- Matrix addition
mAdd :: Num a => Matrix a -> Matrix a -> Matrix a
mAdd (Matrix a) (Matrix b) = Matrix $ zipWith (zipWith (+)) a b


(|+|) (Matrix a) (Matrix b) = mAdd (Matrix a) (Matrix b)


-- Adding a list of matrices

mAddL :: Num a => [Matrix a] -> Matrix a
mAddL m = foldl1 (mAdd) m


-- Matrix subtraction
mSub :: Num a => Matrix a -> Matrix a -> Matrix a
mSub (Matrix a) (Matrix b) = Matrix $ zipWith (zipWith (-)) a b


(|-|) (Matrix a) (Matrix b) = mSub (Matrix a) (Matrix b)


-- Subtracting a list of matrices
mSubL :: Num a => [Matrix a] -> Matrix a
mSubL m = foldl1 (mSub) m


-- Matrix transposition
mTranspose :: Matrix a -> Matrix a
mTranspose (Matrix []) = (Matrix [])
mTranspose (Matrix [[]]) = (Matrix [[]])
mTranspose xs = Matrix $ foldr (zipWith (:)) (repeat []) (mat2list
xs)


-- Multiplication by a scalar
mScalarMult :: Num a => a -> Matrix a -> Matrix a
mScalarMult x (Matrix m) = Matrix $ map (map (x*)) m


(|*|) x (Matrix m) = mScalarMult x (Matrix m)


-- Matrix multiplication
mMult :: Num a => Matrix a -> Matrix a -> Matrix a
mMult (Matrix m1) (Matrix m2) = Matrix $ [ map (multRow r) m2t | r
<- m1 ]
      where
            (Matrix m2t) = mTranspose (Matrix m2)
            multRow r1 r2 = sum $ zipWith (*) r1 r2
```

```
(|><|) (Matrix a) (Matrix b) = mMult (Matrix a) (Matrix b)


-- Multiplying a list of Matrices
mMultL :: Num a => [Matrix a] -> Matrix a
mMultL m = foldl1 (mMult) m


-- Finding number of rows
numRows :: Num a => Matrix a -> Int
numRows (Matrix a) = length a


-- Finding number of columns
numCols :: Num a => Matrix a -> Int
numCols (Matrix a) = numRows (mTranspose (Matrix a))


-- Finding coordinates/position of an element
coords :: Num a => Matrix a -> [[(Int, Int)]]
coords (Matrix a) = zipWith (map . (,)) [0..] $ map (zipWith const
[0..]) a

delmatrix :: Num a => Int -> Int -> Matrix a -> Matrix a
delmatrix i j (Matrix a) = Matrix $ dellist i $ map (dellist j) a
      where
            dellist i xs = take i xs ++ drop (i + 1) xs

-- Converting a Matrix into a list
mat2list :: Matrix a -> [[a]]
mat2list (Matrix m) = m


-- Calculating determinant of a matrix
--determinant :: [[Double]] -> Double
--determinant :: Matrix a -> Double
determinant (Matrix m)
      | numRows (Matrix m) == 1 = head (head m)
      | otherwise    = sum $ zipWith addition [0..] m
      where
            addition i (x:_) =  x * cofactor i 0 (Matrix m)


-- Calculating cofactor
cofactor :: Int -> Int -> Matrix Double -> Double
cofactor i j (Matrix m) = ((-1.0) ** fromIntegral (i + j)) *
determinant (delmatrix i j (Matrix m))


-- Calculating minors
cofactorM :: Matrix Double -> Matrix Double
cofactorM (Matrix m) = Matrix $ map (map (\(i,j) -> cofactor j i
(Matrix m))) $ coords (Matrix m)
```

```
-- Matrix inversion
inverse :: Matrix Double -> Matrix Double
inverse (Matrix m) = Matrix $ map (map (* recip det)) $ mat2list $
cofactorM (Matrix m)
      where
            det = determinant (Matrix m)


-- Matrix division
mDiv :: Matrix Double -> Matrix Double -> Matrix Double
mDiv (Matrix a) (Matrix b) = mMult (Matrix a) (inverse (Matrix b))


(|/|) (Matrix a) (Matrix b) = mDiv (Matrix a) (Matrix b)


-- Extract particular row of a matrix
extractRow :: Matrix a -> Int -> [a]
extractRow (Matrix m) n = m !! n


-- Extract particular column of a matrix
extractCol :: Matrix a -> Int -> [a]
extractCol (Matrix m) n = (mat2list (mTranspose (Matrix m))) !! n


-- Extract range of rows from a matrix
extractRowRange :: Matrix a -> Int -> Int -> Matrix a
extractRowRange (Matrix m) a b = Matrix [extractRow (Matrix m) i | i
<- [a..b]]


-- Extract range of columns from a matrix
extractColRange :: Matrix a -> Int -> Int -> Matrix a
extractColRange (Matrix m) a b = Matrix [extractCol (Matrix m) i | i
<- [a..b]]


-- Power of a matrix
--mPower :: Num a => Matrix a -> Int -> Matrix a
mPower (Matrix matrix) exp =
      if (exp < 0)
      then mPower (inverse (Matrix matrix)) (-exp)
      else
            if(exp == 0)
            then error "Exponent must be non-zero."
            else
                  if (exp == 1)
                  then (Matrix matrix)
                  else
                        mMult (Matrix matrix) (mPower (Matrix matrix)
(exp-1))


-- Trace of a matrix
```

```
trace (Matrix m) = sum [(extractRow (Matrix m) r) !! c | r <- [0 ..
((numRows (Matrix m)) - 1)], c <- [0 .. ((numCols (Matrix m)) - 1)],
r == c]


-- Invertibility
--isInvertible :: Num a => Matrix a -> Bool
isInvertible (Matrix m) = (determinant (Matrix m)) /= 0


-- Is it symmetric?
--isSymmetric :: Eq a => Matrix a -> Bool
isSymmetric (Matrix m) = (Matrix m) == mTranspose (Matrix m)


-- Is it anti/skew-symmetric?
--isSkewSymmetric :: Eq a => Matrix a -> Bool
isSkewSymmetric (Matrix m) = (Matrix m) == mScalarMult (-1)
(mTranspose (Matrix m))


-- Is it a row matrix?
--isRow :: Ord a => Matrix a -> Bool
isRow (Matrix m) = (numRows (Matrix m) == 1)


-- Is it a column matrix?
--isColumn :: Ord a => Matrix a -> Bool
isColumn (Matrix m) = (numCols (Matrix m) == 1)


-- Is it a square matrix?
--isSquare :: Ord a => Matrix a -> Bool
isSquare (Matrix m) = (numRows (Matrix m) == numCols (Matrix m))


-- Orthogonality
--isOrthogonal :: Eq a => Matrix a -> Bool
isOrthogonal (Matrix m) = (mTranspose (Matrix m) == inverse (Matrix
m))


-- Is it an involutive matrix?
--isInvolutive :: Eq a => Matrix a -> Bool
isInvolutive (Matrix m) = ((Matrix m) == inverse (Matrix m))


-- Is it a 0/1 Matrix?
--isZeroOne :: Ord a => Matrix a -> Bool
isZeroOne (Matrix m) = and [(((m!!r)!!c) == 0) || (((m!!r)!!c) == 1)
| r <- [0..((numRows (Matrix m)) - 1)], c <- [0..((numCols (Matrix
m)) - 1)]]


-- Is it a zero matrix?
--isZero :: Ord a => Matrix a -> Bool
```

```
isZero (Matrix m) = and [(m!!r)!!c == 0 | r <- [0..((numRows (Matrix
m)) - 1)], c <- [0..((numCols (Matrix m)) - 1)]]


-- Is it a one matrix?
--isOne :: Ord a => Matrix a -> Bool
isOne (Matrix m) = and [(m!!r)!!c == 1 | r <- [0..((numRows (Matrix
m)) - 1)], c <- [0..((numCols (Matrix m)) - 1)]]


-- Is it a unit matrix?
--isUnit :: Eq a => Matrix a -> Bool
isUnit (Matrix [[]]) = False
isUnit (Matrix [[1]]) = True
isUnit (Matrix m) = and ([isSquare (Matrix m)] ++ [isOrthogonal
(Matrix m)] ++ [isSymmetric (Matrix m)] ++ [trace (Matrix m) ==
fromIntegral (numRows (Matrix m))])


-- Mapping a function to a matrix
mMap f (Matrix m) = Matrix $ map (map f) m


-- Generate special matrices

-- Temp function (converts list to n-row matrix)
chunk' n = takeWhile (not.null) . map (take n) . iterate (drop n)


-- NxN 0 matrix
zero n = Matrix $ chunk' n (take (n*n) $ repeat 0)


-- MxN 0 matrix
zero' m n = Matrix $ chunk' m (take (m*n) $ repeat 0)


-- NxN 1 matrix
one n = Matrix $ chunk' n (take (n*n) $ repeat 1)


-- MxN 1 matrix
one' m n = Matrix $ chunk' m (take (m*n) $ repeat 1)


-- NxN unit matrix
unit n = Matrix $ chunk' n (L.intercalate (take n $ repeat 0)
(mat2list (one' 1 n)))
```

12.1.7 Combinatorics

```
module MPL.Combinatorics.Combinatorics
(
      factorial,
      c,
      p,
      permutation,
      shuffle,
      combination
)
where

import Data.List as L
import System.Random
import Control.Applicative


-- Factorial function
factorial :: Integer -> Integer
factorial n
      | (n == 0) = 1
      | (n > 0) = product [1..n]
      | (n < 0) = error "Usage - factorial n, where 'n' is non-
negative."


-- nCr
c :: Integer -> Integer -> Integer
c n r = (factorial a) `div` ( (factorial b) * (factorial (a-b)) )
      where
      a = max n r
      b = min n r


-- nPr
p :: Integer -> Integer -> Integer
p n r = (factorial a) `div` (factorial (a-b))
      where
      a = max n r
      b = min n r


-- Permutation generation function
permutation :: [a] -> [[a]]
permutation x = L.permutations x


-- Random permutation generation - Fisher-Yates shuffle algorithm
shuffle :: [a] -> IO [a]
shuffle l = shuffle' l []
      where
            shuffle' [] acc = return acc
            shuffle' l acc =
                  do
```

```
                      k <- randomRIO (0, (length l) - 1)
                      let (lead, x:xs) = splitAt k l
                      shuffle' (lead ++ xs) (x:acc)


-- Generating combination (n at a time, with repetition)

prod as bs = (++) <$> as <*> bs

combination n as = foldr1 prod $ replicate n as
```

12.2 Preprocessor

12.2.1 Bash Script

```
#! /usr/bin/sh
# preprocess.sh

sed -f script $2 > $3
```

12.2.2 sed Script

```
# sed script for substituting text according to MPL's syntax
# Filename: script
# Author  : Rohit Jha
# Version : 0.1 (24 Jan 2013)

s/Set[\n\t ]{/Set [/g;
s/Relation[\n\t ]{/Relation [/g;
s/Vector[\n\t ]</Vector [/g;
s/Edges[\n\t ]{/Edges [/g;
s/Vertices[\n\t ]{/Vertices [/g;

s/[\n\t ]}/]/g;
s/[\n\t ]>/]/g;
```