Jebr

DOCUMENTATION

By:

'Umar A.Abu Bakr, Fahed N.Shehab

Jebr Documentation

-- This page was intentionally left blank --

Jebr Documentation

‘Umar A.Abu Bakr
Fahed N.Shehab

For the soul of
Abu Abdullah Muhammad ibn Musa Al Khwarizmi

Foreword

"When I first learned how to program in high school, I didn't realize the importance of mathematics in such field, but when I went to college, things started to change.I realized that mathematics is the only science that lies behind every field we, humans, face, it is the engine of our civilizations and it is the greatest achievement we have ever achieved. Mathematics is an incredible science, a science that gives abstract entities, values and names and make them unique, measurable and understandable. And that's the reason why I chose Algebra to be the primary domain for Jebr.

Our values call upon us to care for those who we will never meet and to care for the education of those who will come after us. Thus, Jebr tries to establish a small base for those young scientists to examine, modify and execute a code that was written by two students, which they are having the same knowledge that those two had.

Passion and love are the main emotions that I have when I work on, talk about or even hear of Jebr, it combines all of my favorite subjects and it's the essence of what we learned in the computer science department."

--*'Umar Ahmad Abu Bakr*

"I am a fan of math, and this is why I study computer science. When Omar told me about the idea of the project, a programming language for linear algebra, I was very excited. It is an excellent idea to help students of mathematics, and it's also a good idea for students of computer science to study the language analysis and implementation. It is also a great way to make students like what they study, and that also encouraged me to this idea, I've done a simplified programming language compiler for the C language when I was taking the Compiler Design course and I liked the idea so much. In the course of programming languages, I learned the basics of designing programming languages, and the picture was clear in all senses to start the project and we were and still are excited to work on this idea."

--*Fahed Naif Shehab*

Table Of Contents:

-- This page was intentionally left blank --

Preface

Historical Background:

A programming language is a formal constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

'Umar A,Abu Baker and Fahed N.Shehab initiated Project Jebr in July 2014. After taking a linear algebra class and solving complex problems manually, 'Umar had the idea to create a simple but yet powerful programming language that allows linear algebra students to solve their problems through programming logic, speed up and increase the accuracy of linear algebra calculations. Then, he proposed the idea to Fahed, who agreed to be part of the team. The language was initially called MilkShake. Later the project was finally renamed to Jebr.

The core of Jebr was born from the integration of two projects: “Meow: Minimized Educational Open Windowing IDE” by 'Umar A.Abu Bakr and “Cyclone: Compiler” by Fahed N.Shehab.

Weaknesses in the Current Platforms:

MATLAB, Python and R offer a great linear algebra support, so one may wonder why are we making a new linear algebra programming language? To answer that we must take a look at these platforms' advantages and weaknesses. Python is a very easy dynamic programming language that offers a great environment that supports linear algebra coding, but there’s already a strong tendency in the language to require you to know everything before you can do anything. On the other hand, we have MATLAB, which is the best in this field. In fact, the Linear Algebra course from MIT open courseware is based on MATLAB and half of the linear algebra books published by SIAM use MATLAB. However, MATLAB's price is very high for students and updates costs even more. On the contrary, R language is open, free and easy-to-use language, but the main weakness in it is the lack of documentation and tutorials that can be found online.

Introduction

Need Statement:

Jebr is a static, imperative, object-oriented and domain-specific programming language that supports linear-algebra oriented operations and matrix manipulation to allow users to solve mathematical issues using programming logic. The language supports multiple programming[1] paradigms and the WORA “Write Once, Run Anywhere” ideology.

Jebr difference from other linear-algebra-oriented programming languages springs from the fact that it's is a Free-software project that is simple but yet powerful, supports the WORA coding ideology, and it's free to examine and edit the code or any other language entity.

The project is considered to be the first domain-specific programming language developed entirely by An-Najah National University students. It is intended to serve in many fields; as a programming language to help students solving linear algebra problems through programming logic, as well as an example for them to analyze and develop in both “Programming Languages” and “Compiler Design” courses

Language design:

Jebr follows static languages style but makes many changes aimed at simplicity, safety, readability, writability, orthogonality, variety and compatibility. The language also desires to keep the language specification simple enough to hold in a programmer's head, in part by omitting features common to similar languages such as not supporting type inheritance, method and operator overloading, pointers and genric programming. The language also adds some basic types not present in any other similar language like vector, space and matrix.

Intended Audience:

A domain-specific language (DSL) is a computer language specialized to a particular application domain and is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside it. Thus, The intended audience is considered to be linear-algebra and computer science students in the first place, as it offers a powerful but yet simple environment to work with and analyze. The language is also intended to help instructors to solve complicated problems quickly during lectures' time.

Programming languages also can be used to create programs to control the behavior of a machine or to express algorithms. Thus, the project can serve as an in-house development project, that allows computer science students to examine the code of the IDE and the interpreter, examine the BNF and also edit them.

[1] *(for more info about Programming Paradigms, take a look at Appendix A).*

Glossary

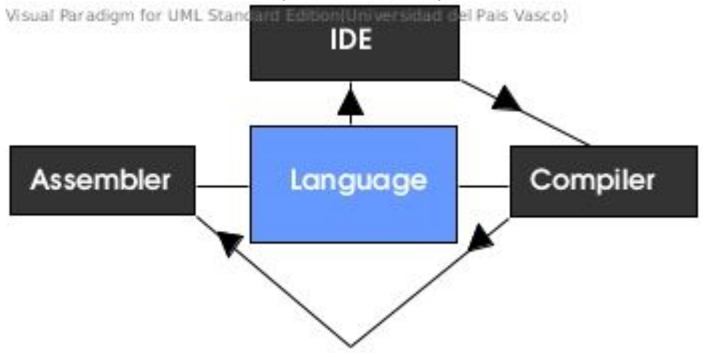
Term	Meaning
Dynamic Programming Language	Dynamic programming language is a term used in computer science to describe a class of high-level programming languages which, at runtime, execute many common programming behaviors that static programming languages perform during compilation. These behaviors could include extension of the program, by adding new code, by extending objects and definitions, or by modifying the type system. These behaviors can be emulated in nearly any language of sufficient complexity, but dynamic languages provide direct tools to make use of them. Many of these features were first implemented as native features in the Lisp programming language.
Static Programming Language	A programming where all expressions have their types determined prior to when the program is executed, typically at compile-time. For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.
Imperative Programming Language	A term used in computer science to describe a programming language that follows imperative programming, which is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.
Object-Oriented	Is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as s.
Domain-Specific Programming Language	Is a computer language specialized to a particular application domain.
Free Software	Is computer software that gives users the freedom to run the software for any purpose as well as to study, modify, and distribute the original software and the adapted versions. The rights to study and modify free software imply unfettered access to its source code.
WORA ideology	"Write once, run anywhere", an ideology that illustrates the cross-platform benefits of a computer programming language.

System Architecture

Project Jebr consists of 4 main components: the language, the IDE, the interpreter and the assembler. These components interact with each other to make Jebr as robust and consistent as possible.

The system is also split into two sub-systems: the Jebr language; which is the core of the project and the additional-system; which offers the additional parts that Project Jebr promises to offer.

The Context-Flow-Model below allows you to see the systems that project Jebr contains. The ones shaded with gray are parts of the “additional-system” subsystem.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
IDE
Assembler
Language
Compiler

Figure[4:1]:Project Jebr High-level Context-Flow-Model

I. ***Language Design:***

Jebr follows static languages style but makes many changes aimed at simplicity, safety, readability, writability, orthogonality, variety and compatibility. The language also desires to keep the language specification simple enough to hold in a programmer's head, in part by omitting features common to similar languages such as not supporting type inheritance, method and operator overloading, pointers and genric programming. The language also adds some basic types not present in any other similar language like vector, space and matrix.

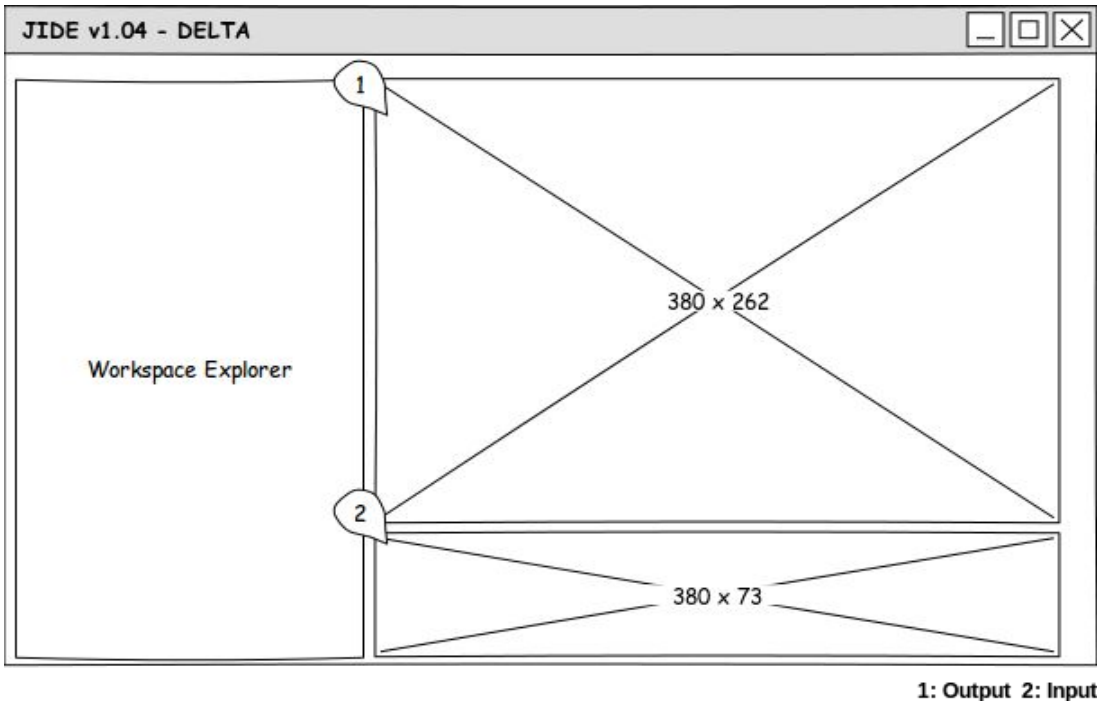
To provide a simple language that anyone can learn quickly, the designers removed equal-syntax-similarity between assigning and declaring, removed pointers, replaced “traditional for loops” with “iterator-based for loops”, removed destructors, removed varying number of parameters and implemented a very-basic garbage collecting mechanism.

For safety, Jebr doesn't allow type inheritance, pointers, method and operator overloading and genric programming.

The language design phase was centered around readability, writability, varity and orthogonality of the language and reducing key strokes for users. Thus, the designers removed equal-syntax-similarity between assigning and declaring, replaced curly braces with begin-end, removed traditional for, replaced == (is equal?) with =?, agreed on a naming convention, adopted WORA ideology and removed varying number of parameters.

II. ***JIDE -Jebr IDE***-

JIDE core systems were forked from MEOW, which is a very minimal development environment that 'Umar developed while taking "Compiler Design" course. JIDE offers an interface for the interpreter and another one for the standard interpreter, where users can write classes and functions. The first interface offers basic functionality and uses basic Java Swing components, while the second interface offers syntax-highlighting, code-folding and line-numbering.



Figure[4:2] Interpreter interface prototype.

JIDE recognizes both .jbl (Jebr Library) and .jdf (Jebr Data File) file extensions, the first is used to save compiled functions and classes, while the second is used to store configuration status and information about the hosting machine.

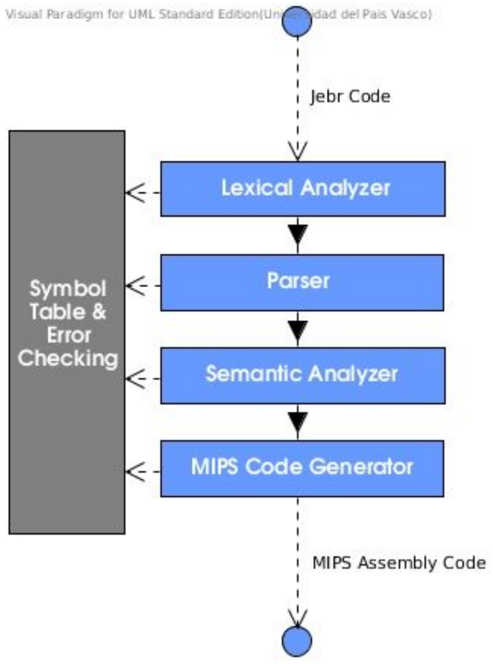
Ease of use, reliability, robustness and portability are the main properties that JIDE tries to offer. The interface is so simple and easy to figure out, yet JIDE will have a small tool that identifies where do users most ask for help and report the data to the designers, so they can fix that specific issues. JIDE is developed using Java so it offers high portability on the three main x86 32-bit operating systems; MS Windows, GNU/Linux and Mac.

JIDE System requirements:

Supported Architecture:
x86 32-bit machines
Supported Operating Systems:
MS Windows 7 or higher, GNU/Linux (uses .deb packages), Mac OSX
Additional Requirements:
gcc, g++ and build-utils should be installed before installing JIDE (GNU/Linux users only)

III. ***JINT (Jebr Interpreter)***

The JINT interpreter is based on the Cyclone compiler, which Fahed developed while taking “Compiler Design” course. The interpreter consists of four main levels; a Lexical Analyzer, which converts a sequence of characters into a sequence of tokens, i.e. meaningful character strings, a parser, which analyses a string of symbols conforming to the rules of the grammar, a semantic analyzer, which adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings, a code generator, which translates Jebr code to MIPS 32-bit instructions.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
Jebr Code
Lexical Analyzer
Parser
Semantic Analyzer
MIPS Code Generator
Symbol Table & Error Checking
MIPS Assembly Code

Figure[4:3] Context-Flow-Model of JINT

The interpreter[2] is based on a LL sequential recursive parser. As a consequence of its sequential nature, the parser had some efficiency troubles, but by using effective algorithms, the parser recorded an excellent time for parsing huge amount of code.

IV. ***MIPS Assembler:***

The MIPS assembly is a very portable language because it runs on it's own virtual 32-bit machine. Thus, the designers chose it to increase the language portability. The JINT translates Jebr code into MIPS 32-bit instructions and here comes the Assembler turn to translate those instructions to a machine code.

V. ***Files & File System Management:***

At the splash screen, the user gets to choose his workspace directory. In that directory each project is stored as a directory containing two subdirectories; “src” and “lib”. The “src” file contains the source code used in that project, and the “lib” folder contains libraries and data files used for that project.

[2] *(for more info about Compilers and Interpreters, take a look at Appendix B).*

User Requirements Definitions

Project Entities and Services:

1. JIDE (Jebr IDE) – Using GUI:

I. **Users:**

- *1.1 Edit Libraries.*
- *1.2 Adding/Creating Libraries.*
- *1.3 Using the interpreter.*
- *1.4 Saving Progress.*
- *1.5 Getting Help/Reporting Bugs.*

II. ***Interpreter:***

- 2.1 *Executing Commands.*

III. ***Filing System:***

- *3.1 Choosing a workspace.*
- *3.2 Renaming a workspace.*
    - *3.3 Adding a project.*
    - *3.4 Renaming a project.*
    - *3.5 Removing a project.*

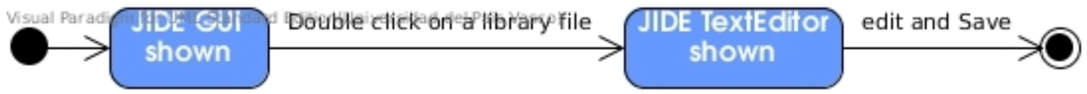
1. Using The JIDE GUI:

1. 1 Users:

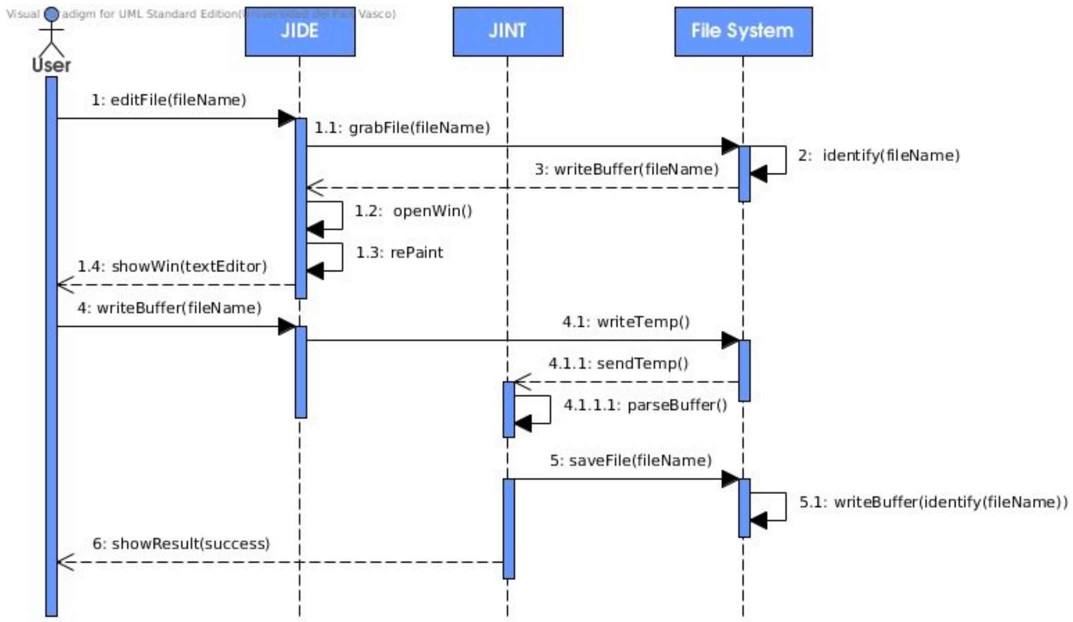
1.1.1 Edit Libraries:

Libraries are those files that contains pre-parsed routines and classes, the user can edit those libraries , if he wishes to abstract data or procedures.

Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should double click the library he wishes to edit from the project explorer next to the interpreter result component. JIDE's text editor will open to allow you to edit that specific library. When the user wishes to save what he modified he can click save, and exit the text editor.*
Errors that could occur: *The modifications contain errors, correcting the errors, then re-saving the file will resolve the issue.*
Final State: *The user will get a message telling him that his modifications are saved.*



Figure[5:1]: Editing libraries Machine State diagram

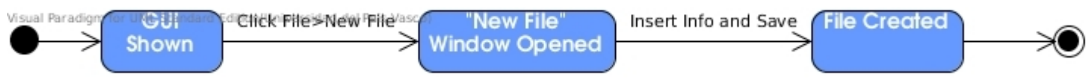


Figure[5:2]: Editing libraries Sequence diagram

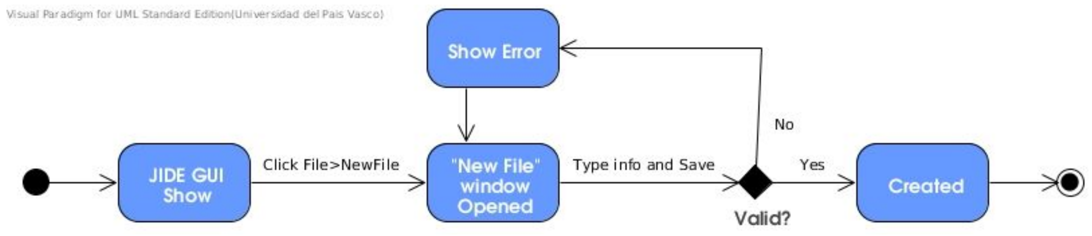
1.1.2 Adding/Creating Files:

Libraries are those files that contains pre-parsed routines and classes, the user can create as many libraries as he want, if he wishes to abstract data or procedures. Source files are those files that contain the program instructions.

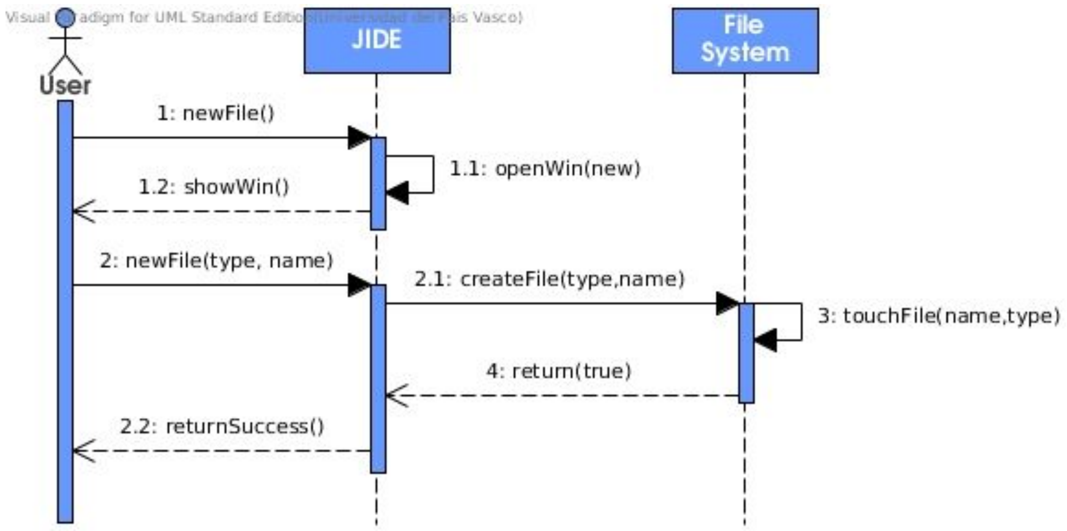
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should choose "New File" from File menu, a window will popup asking for the type and the name of the new file, the user should fill enter the information rquired and then hit Save. A new file will be created.*
Errors that could occur: *A file having the same name already exists, rechoosing a name or the path would solve the issue.*
Final State: *The user will see a new file in the project explorer.*



Figure[5:3]: Adding/Creating Files State diagram.



Figure[5:4]: Adding/Creating Files Activity diagram.

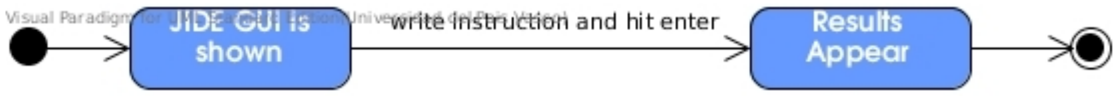


Figure[5:5]: Adding/Creating Files Sequence diagram.

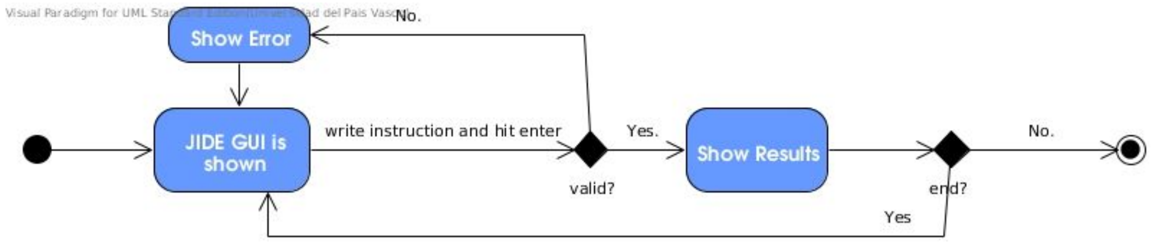
1.1.3 Using the Interpreter:

The interpreter interface is where the user execute line-by-line commands. It splits into two parts; one for the input and one for the output.

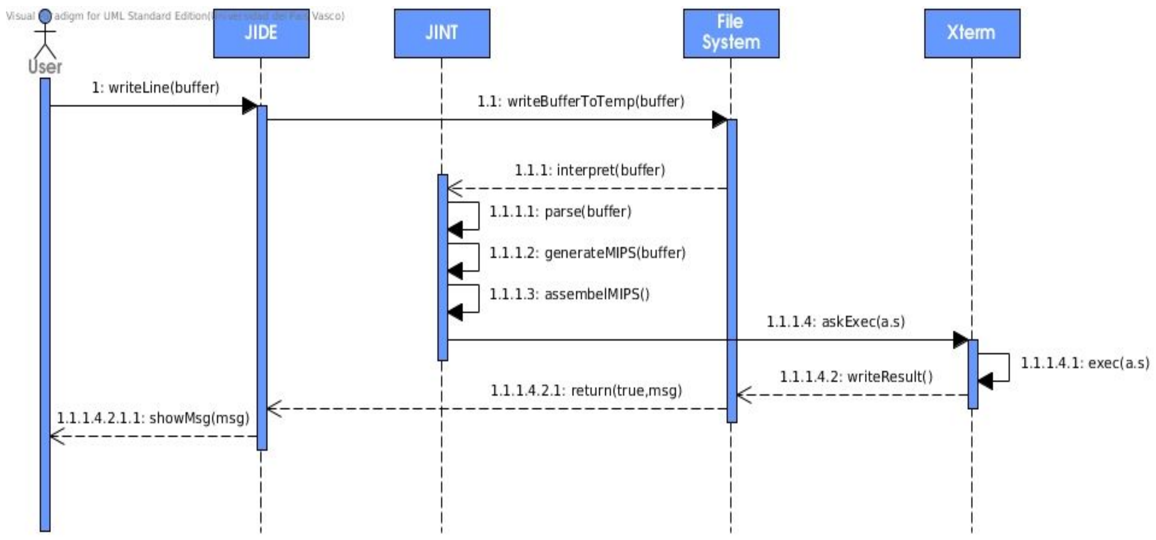
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user start typing Jebr instructions line by line, each one followed by an Enter. JIDE sends each line to JINT, which executes that specific line, and returns the result to the ouptut area in JIDE.*
Errors that could occur: *The instruction contains errors, rewrite the instruction using proper syntax.*
Final State: *The user will get the output of the instructions on the output area.*



Figure[5:6]: Using the Interpreter Machine State diagram.



Figure[5:7]: Using the Interpreter Activity diagram.

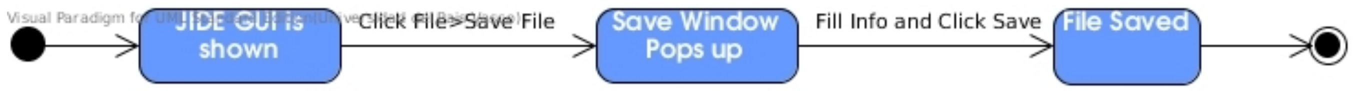


Figure[5:8]: Using the Interpreter Machine State diagram.

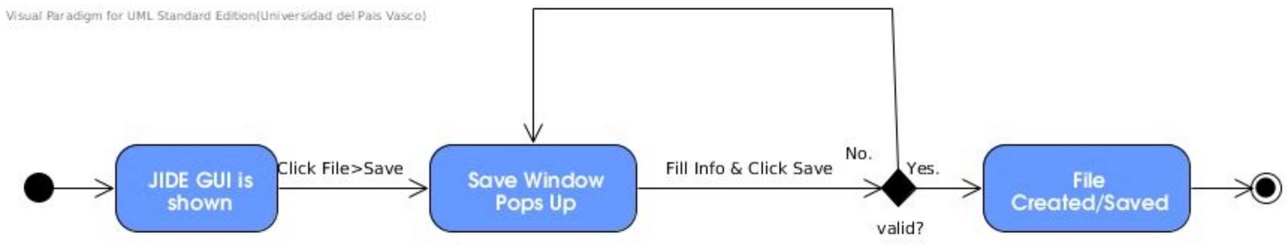
1.1.4 Saving Progress:

When you save a file, you can save it to a folder on your hard disk drive, a network location, disk, DVD, CD, the desktop, flash drive, or save as another file format.

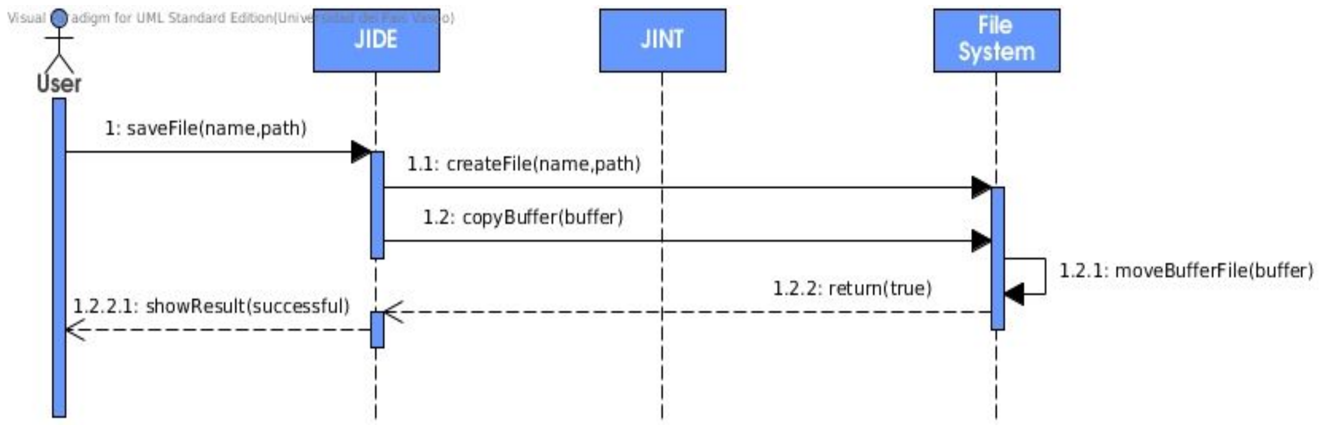
Initial State: *The user is at the JIDE's Interpreter interface after writing some instructions.*
Normal Flow: *The user should choose “Save” from File menu, a window will popup asking for the place where the user would like to keep the source file and then hit Save. A new file will be created containing the previous instructu.*
Errors that could occur: *A file having the same name already exists, rechoosing a name or the path would solve the issue.*
Final State: *The user will see a new file in the project explorer.*



Figure[5:9]: Saving Progress Machine State diagram.



Figure[5:10]: Saving Progress Activity diagram.

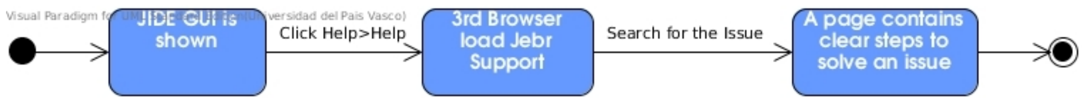

Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
User
JIDE
JINT
File System
1: saveFile(name,path)
1.1: createFile(name,path)
1.2: copyBuffer(buffer)
1.2.1: moveBufferFile(buffer)
1.2.2: return(true)
1.2.2.1: showResult(successful)

Figure[5:11]: Saving Progress Sequence diagram.

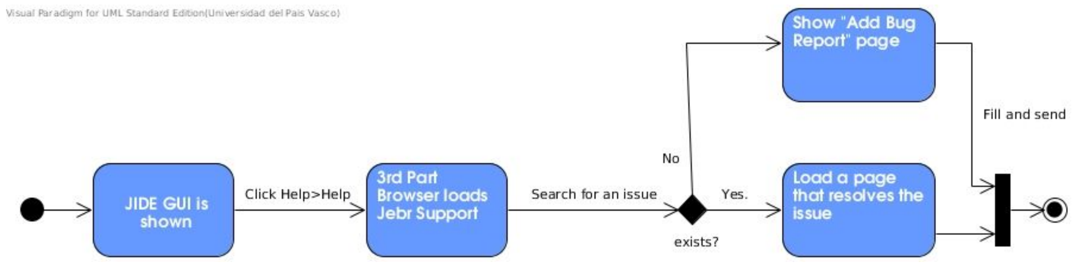
1.1.5 Getting Help/Reporting Bugs:

Search for bugs or feature requests in JIDE support by using Search, If not found, the user should report the bug to help us maintain the project.

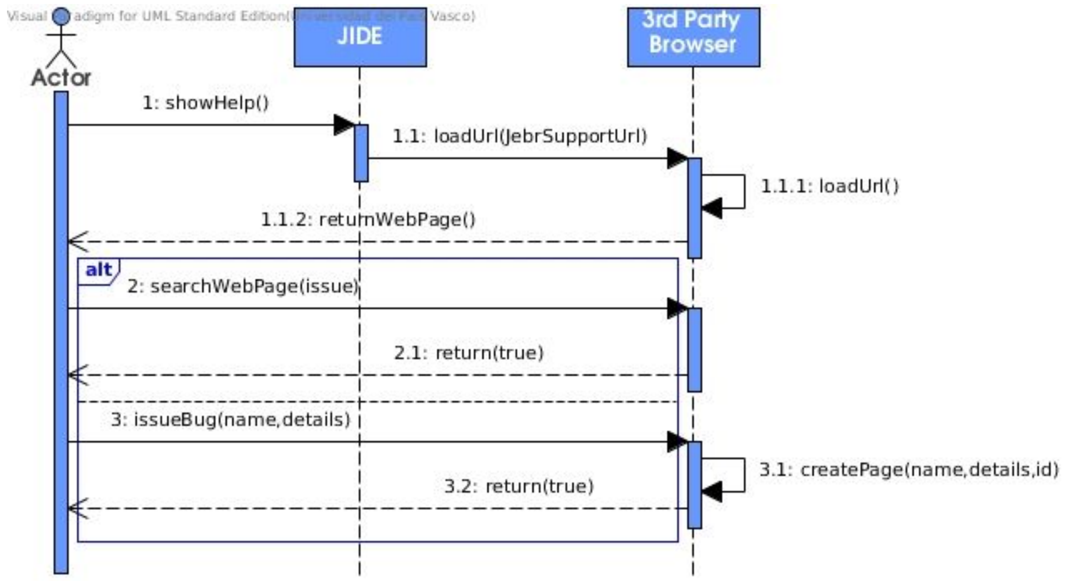
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *A user should choose “Help” from the Help menu, a 3rd party browser will load Jebr support page, wher user could search for whatever he wants.*
Errors that could occur: *The user might not find what he/she wants, then the user should report the bug to us.*
Final State: *The user will have a page containing clear steps that solves his/her issues.*


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
JIDE GUI is shown
Click Help>Help
3rd Browser load Jebr Support
Search for the Issue
A page contains clear steps to solve an issue

Figure[5:12]: Getting Help/Reporting Bugs Machine State diagram.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
JIDE GUI is shown
Click Help>Help
3rd Part Browser loads Jebr Support
Search for an issue
No
Yes.
exists?
Show "Add Bug Report" page
Load a page that resolves the issue
Fill and send

Figure[5:13]: Getting Help/Reporting Bugs Activity diagram.


Visual Paradigm for UML Standard Edition(Universidad del País Vasco)
Actor
JIDE
3rd Party Browser
1: showHelp()
1.1: loadUrl(JebrSupportUrl)
1.1.1: loadUrl()
1.1.2: returnWebPage()
alt
2: searchWebPage(issue)
2.1: return(true)
3: issueBug(name,details)
3.1: createPage(name,details,id)
3.2: return(true)

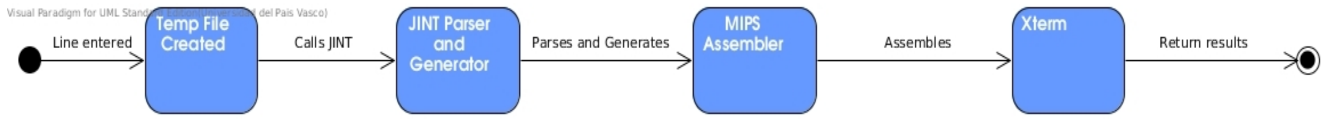
Figure[5:14]: Getting Help/Reporting Bugs Sequence diagram.

1.2 Interpreter:

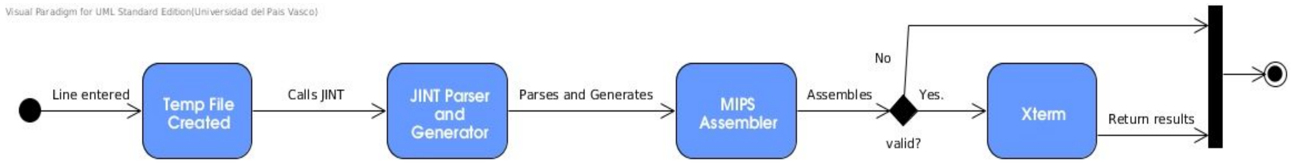
1.2.1. Executing Commands:

A command is a single instruction that instructs the machine to do a single task. executing this command goes through multiple phases.

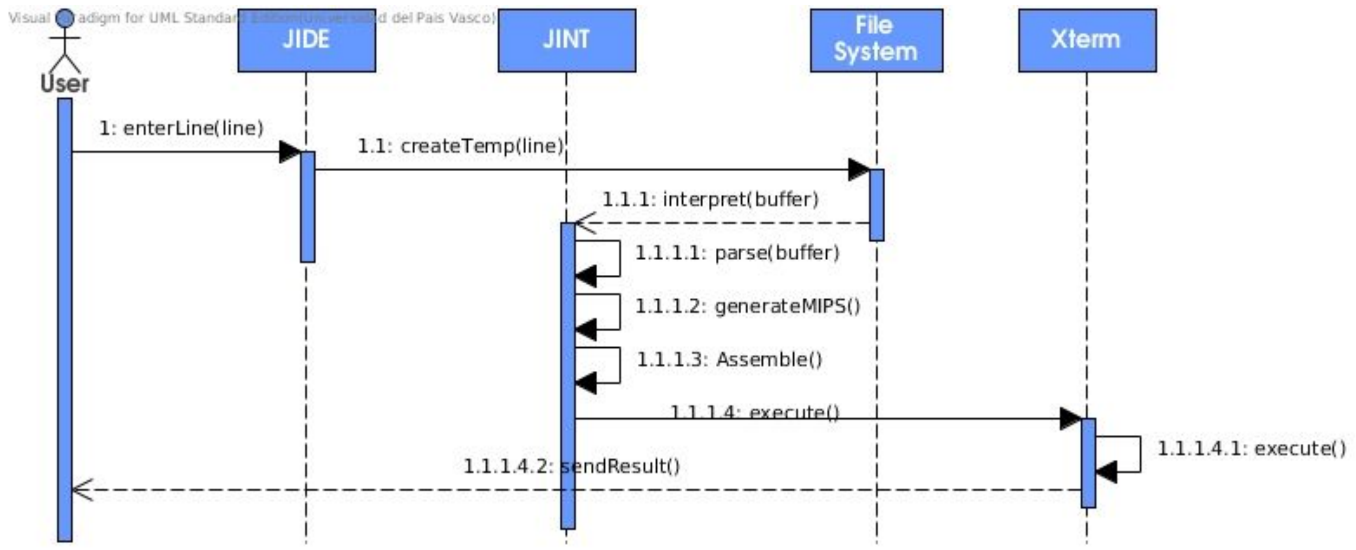
Initial State: *The user entered an instruction in the input field and hit enter.*
Normal Flow: *JIDE copies the instruction to a temp file, then calls JINT to be executed on that temp file. JINT parses and translate the instructions to MIPS assembly and then calls the assembler to convert it to the target machine language. Xterm finally, runs that executable file and send results to JIDE, which show it on the output screen.*
Errors that could occur: *The instruction contains errors, rewrite the instruction using proper syntax.*
Final State:_*The user will see the result of that instruction on the output screen.*



Figure[5:15]: Executing Command Machine State diagram.



Figure[5:16]: Executing Commands Activity diagram.



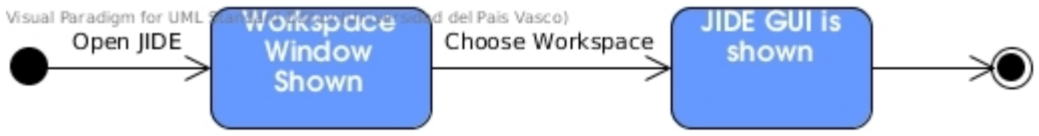
Figure[5:17]: Executing Commands Sequence diagram.

1.3 File System:

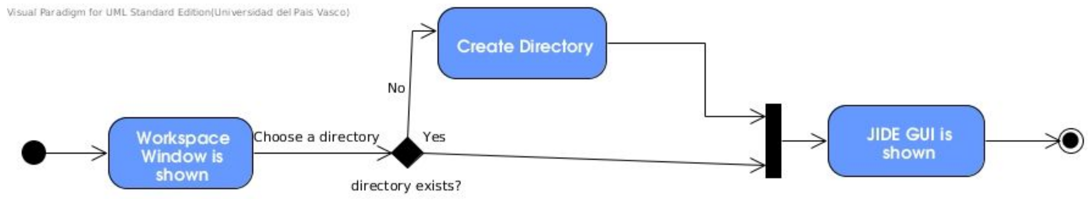
1.3.1 Choosing a Workspace:

A workspace is (often) a file or directory that allows a user to gather various source code files and resources and work with them as a cohesive unit. Often these files and resources represent the complete state of an IDE at a given time, a snapshot. Workspaces are very helpful in cases of complex projects when maintenance can be challenging. How to choose your JIDE workspace?

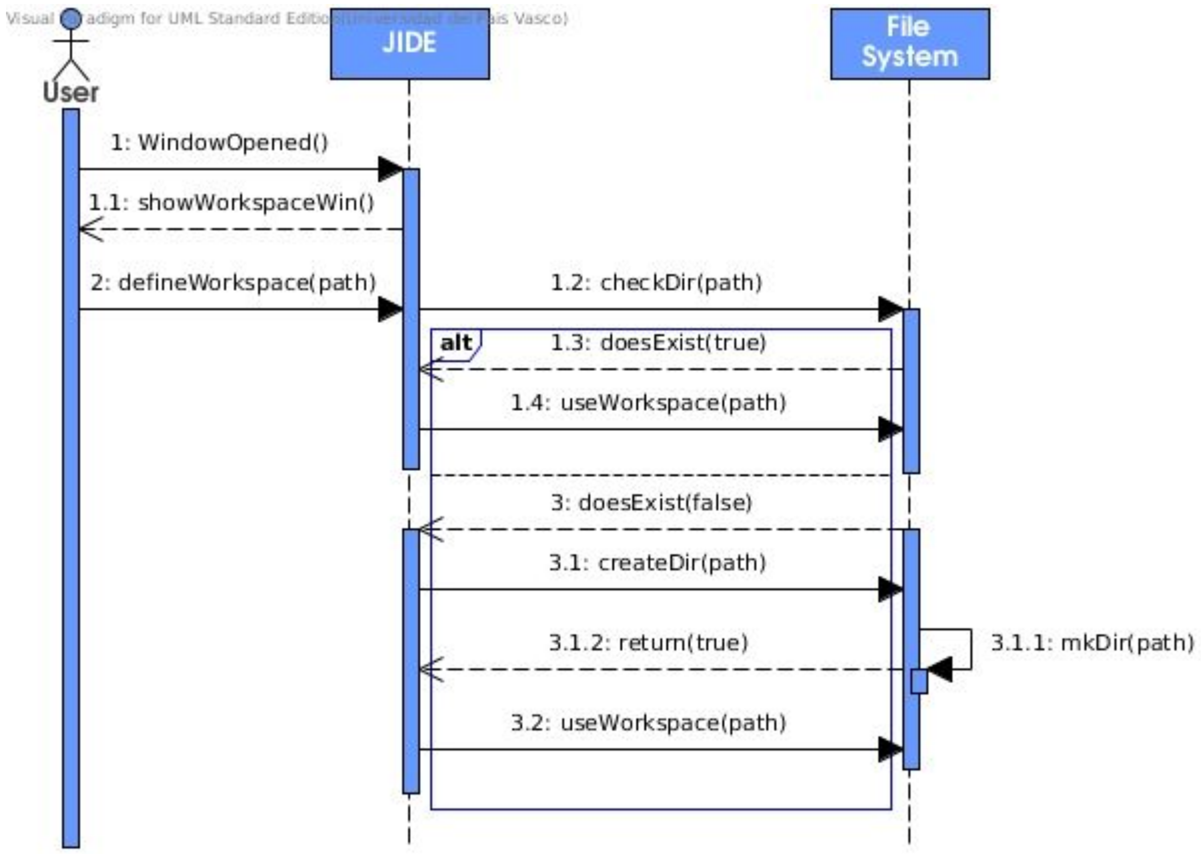
Initial State: *The user is at his desktop, ready to open JIDE.*
Normal Flow: *After Opening JIDE, the IDE will prompt the "Choose a Workspace" window where user can specify the path of where he/she wishes to save his projects and source files. After specifying the path and hitting Ok, JIDE will check if the workspace already exists to import contained projects or to create it and initialize it, if not.*
Final State:*The user will see his projects in the project explorer area.*


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
Open JIDE
Workspace Window Shown
Choose Workspace
JIDE GUI is shown

Figure[5:18]: Choosing a Workspace Machine State diagram.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
Workspace Window is shown
Choose a directory
directory exists?
No
Yes
Create Directory
JIDE GUI is shown

Figure[5:19]: Choosing a Workspace Activity diagram.

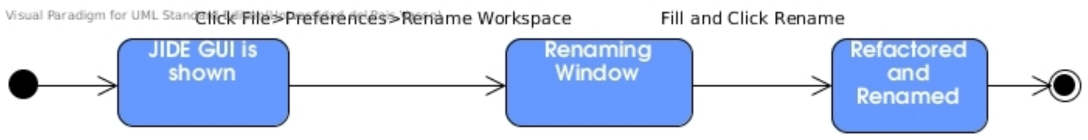


Figure[5:20]: Choosing a Workspace Sequence diagram.

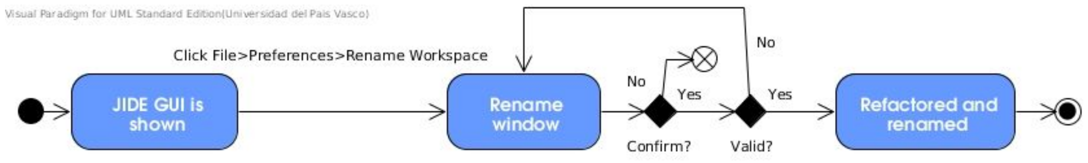
1.3.2 Renaming a Workspace:

If a user wishes to rename a workspace, a "refactor" is needed for all projects inside that specific workspace, in order to modify their configuration files.

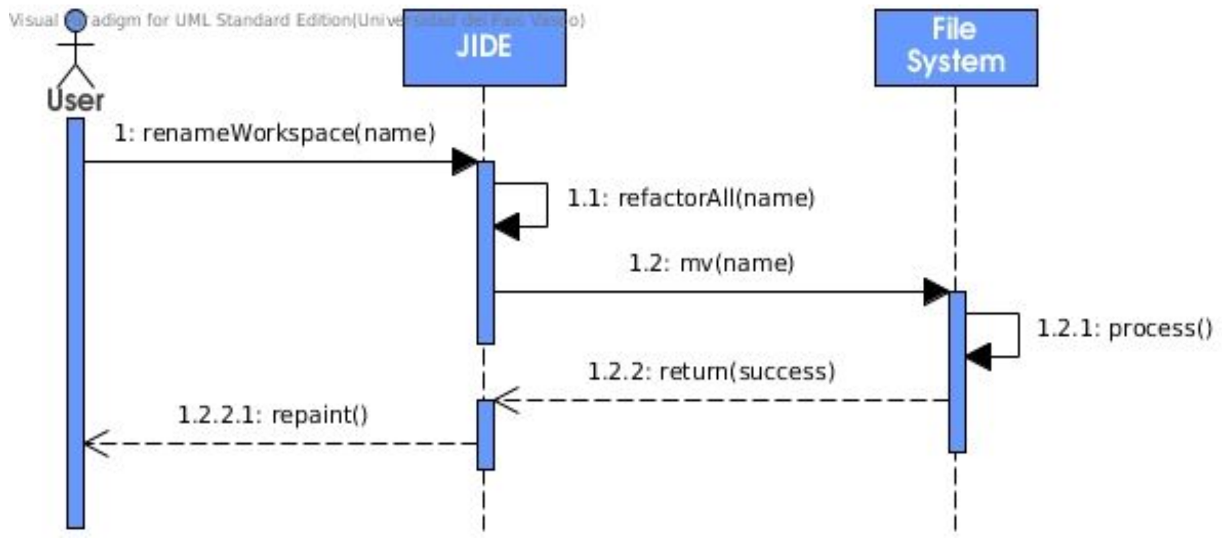
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should click on the "File" menu, choose "preferences", then choose "Rename Workspace". A window will popup asking for the new name and path of the workspace, the user should provide the required info and click "Rename". A confirmation box will appear, which the user should accept. JIDE will rename the workspace.*
Errors that could occur: *The new directory already exists, the user should choose another name or path.*
Final State: *The projects will be located in the new directory.*



Figure[5:21]: Renaming a Workspace Machine State diagram.



Figure[5:22]: Renaming a Workspace Activity diagram.

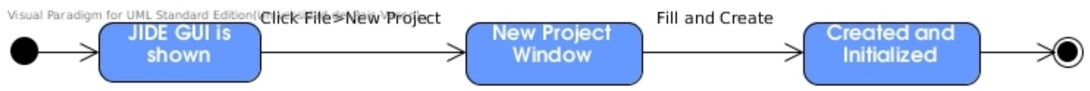


Figure[5:23]: Renaming a Workspace Sequence diagram.

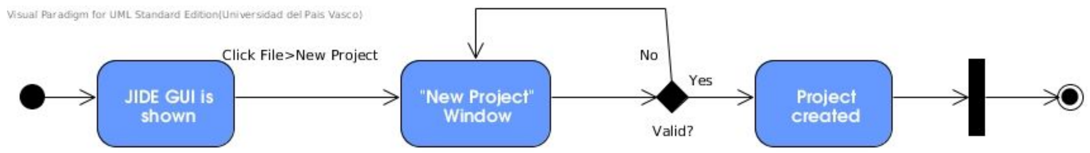
1.3.3 Adding a Project:

A project is a directory that contains all source files and libraries associated with that project. Those files are used to correctly parse and execute that specific project.

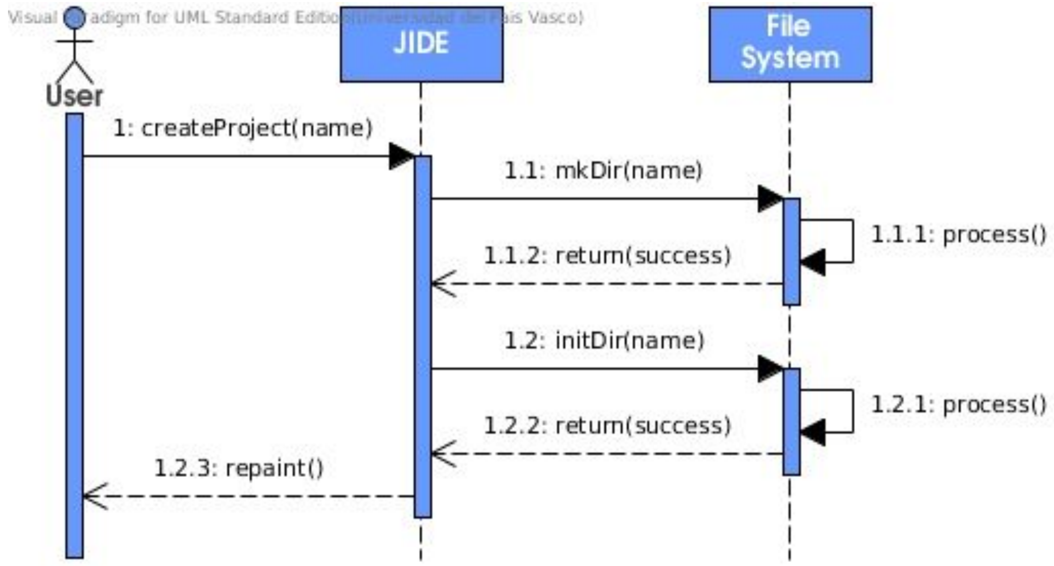
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should click on the "File" menu and choose "New Project". A window will popup asking for the new project's name. the user should provide the required info and click "Create".JIDE will create and initialize that project.*
Errors that could occur: *Aproject with the same name already exist already exists. Choosing another project name will do the trick.*
Final State: *A new project will be added in the "project-explorer".*



Figure[5:24]: Adding a Project Machine State diagram.



Figure[5:25]: Adding a Project Activity diagram.

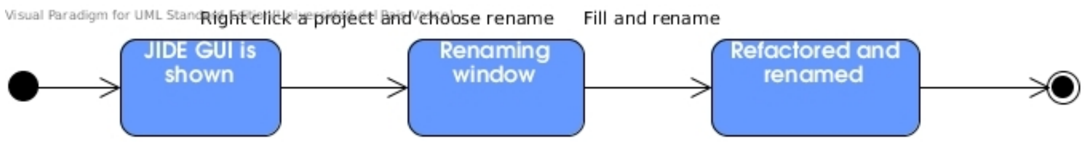


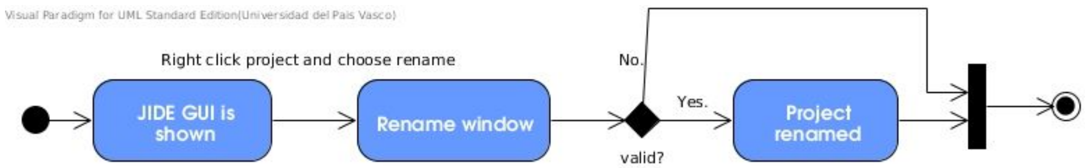
Figure[5:26]: Adding a Project Sequence diagram.

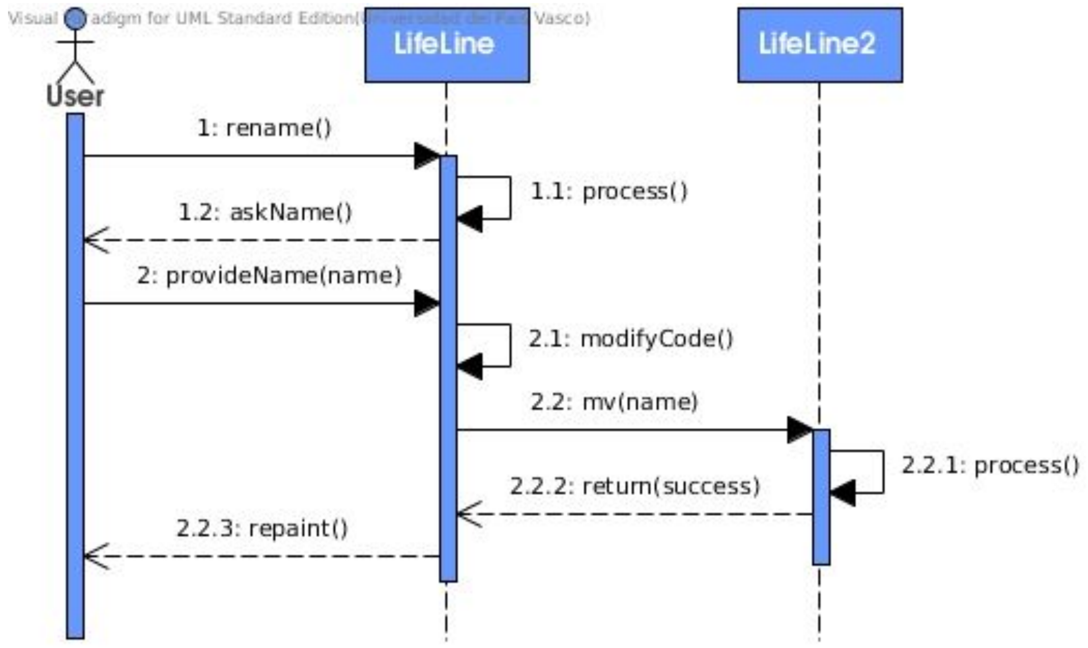
1.3.4 Renaming a Project:

If a user wishes to rename a project or specify another path, a refactor is needed for all files contained in that project's directory, in order to modify all dependencies.

Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should right click on the project and then click rename. A window will popup asking for the new name. the user should provide the required info and click “Rename”.JIDE will rename that project.*
Errors that could occur: *A project with the same name already exist already exists. Choosing another project name will do the trick.*
Final State: *A new name will be associated with that specific project.*


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
Right click a project and choose rename
Fill and rename
JIDE GUI is shown
Renaming window
Refactored and renamed
Figure[5:27]:Renaming a Project Machine State diagram.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
Right click project and choose rename
No.
Yes.
valid?
JIDE GUI is shown
Rename window
Project renamed
Figure[5:28]: Renaming a Project Activity diagram.

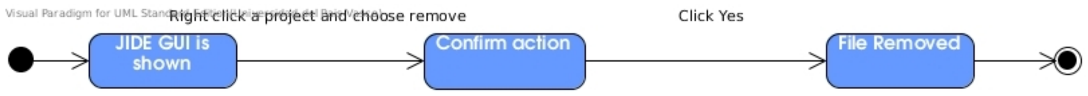


Figure[5:29]: Renaming a Project Sequence diagram.

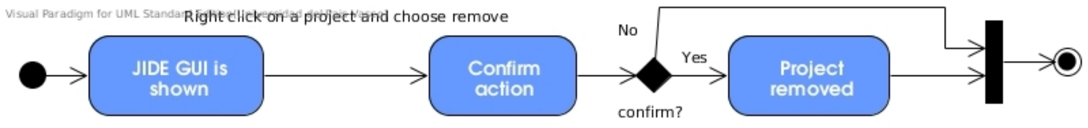
1.3.5 Removing a Project:

If a user wishes to remove all source files and libraries associated with a single, he can remove the project as a whole unit. This action needs caution, because it's not reversible.

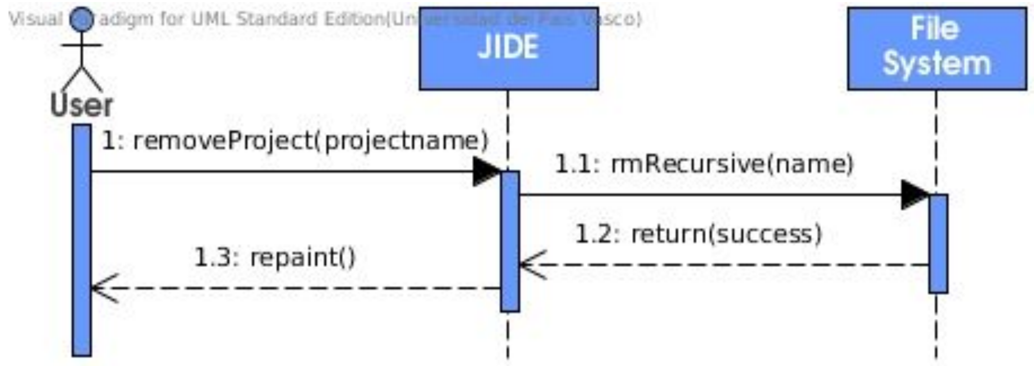
Initial State: *The user is at the JIDE's Interpreter interface.*
Normal Flow: *The user should right click on the project and then click remove. A window will popup asking the user to confirm the action, which the user should accept. .JIDE will remove that project with its associated files and libraries.*
Final State: *The project won't be visible anymore in the “project-explorer”.*


Visual Paradigm for UML Standard Edition
Right click a project and choose remove
Click Yes
JIDE GUI is shown
Confirm action
File Removed

Figure[5:30]: Removing a Project Machine State diagram.


Visual Paradigm for UML Standard Edition
Right click on a project and choose remove
No
Yes
confirm?
JIDE GUI is shown
Confirm action
Project removed

Figure[5:31]:Removing a Project Activity diagram.


Visual Paradigm for UML Standard Edition(Universidad del Pais Vasco)
User
JIDE
File System
1: removeProject(projectname)
1.1: rmRecursive(name)
1.2: return(success)
1.3: repaint()

Figure[5:32]: Removing a Project Sequence diagram.

Language Description and Analysis

1. *The Jebr Language Programming Ideology:*

Jebr follows static languages style but makes many changes aimed at simplicity, safety, readability, writability, orthogonality, variety and compatibility. The language also desires to keep the language specification simple enough to hold in a programmer's head, in part by omitting features common to similar languages such as not supporting type inheritance, method and operator overloading, pointers and genric programming. The language also adds some basic types not present in any other similar language like vector, space and matrix.

```
var_name := matrix(3,2) [[3,2],[7,8],[8,8]];
var_name := vector (4) [1,2,3,4];
var_name := space(funcName,funcName2,R1);
```

To provide a simple language that anyone can learn quickly, the designers removed equal-syntax-similarity between assigning and declaring, removed pointers, replaced "traditional for loops" with "iterator-based for loops", removed overridable destructors, removed varying number of parameters and implemented a very-basic garbage collecting mechanism.

```
var_name := integer 5; //declaring var_name with initial value of 5
var_name = 5; // assign 5 as a value to var_name

for var_name in 1..10
beginfor
   //CODE goes here
endfor

instance_name.doFinalize(); //automatically destroy the instance
```

For safety, Jebr doesn't allow type inheritance, pointers, method and operator overloading and genric programming.

The language design phase was centered around readability, writability, varity and orthogonality of the language and reducing key strokes for users. Thus, the designers removed equal-syntax-similarity between assigning and declaring, replaced curly braces with begin-end,   removed traditional for, replaced == (is equal?) with =?, agreed on a naming convention, adopted WORA ideology and removed varying number of parameters.

```
if (x =? 5)
beginif
    //CODE goes here
endif
```

2. *The Syntax of the Jebr Language:*

I- Decelerations and Simple Statements:

Declaring a variable:

General Form:

```
var_name := datatype [(size)] [initvalue];
```

Examples:

```
x := integer;
x := integer 5;
x := matrix (1,2);
x := matrix (1,2) [[1,2]];
```

Assigning values to variables:

General Form:

```
var_name = value // could be an identifier
```

Examples:

```
x = 5;
x = y;
x = x+4;
```

Calling a Routine:

General Form:

```
funcName(); //parameters go inside braces
```

Example:

```
sum(1,2);
```

Parameters' Declarations:

General Form:

```
var_name := datatype [(size)] [initvalue
(passingtype)];
```

Examples:

```
x := integer;
x := integer 5;
x := integer 5 (ino);
x := matrix (1,2);
x := matrix (1,2) [[1,2]];
x := matrix (1,2) [[1,2]] (out);
```

Calling an Instance's Method:

General Form:

```
instance_name.methodName(); //parameters go between braces.
```

Example:

```
x.sum(1,2);
```

II- Control Statements:

If:

```
if (condition)
beginif
  //CODE goes here
endif
```

If-Else:

```
if (condition)
beginif
    //CODE goes here
  else
    //CODE goes here
Endif
```

If-Elif-Else:

```
if (condition)
beginif
    //CODE goes here
  elif(condition)
    //CODE goes here
  else
    //CODE goes here
endif
```

For:

```
for x in 1..5
beginfor
   //code goes here
endfor
```

While:

```
while(x>5)
beginwhile
   //code goes here
Endwhile
```

do-while:

```
do
   //code goes here
while(x>5);
```

III- Routines & Classes:

Routines:

```
routine funcName
x := integer 5 (in); //parameters
begin
   //empty ROUTINE stub
end;
```

Classes:

```
class ClassName
x:=integer 5;
beginclass
  constructor ClassName
    //parameters
  begincons
         //empty CONSTRUCTOR stub
  endcons
        //list of routines
endclass;
```

Note: *each Class has a built-in non over-ridable function called doFinalizer that acts as a distructor.*

IV- Data Types, Operators and Conditions:

Primitive Data Types:

integer, float, double, string, character, array, matrix, vector, space.

Primitive Operators:

*^(transpose), !(inverse), @(determinant), #(basis), $(rref), -$(ref), &(adjoint), ?(solve-system), *(scalar-multiple & regular multiplication), .(dot- product), %(modulo), +(add), -(subtract), /(divide), +=, -=, *=, /=*

Passing Type:

in(by value), out(by result), ino(value_result)

Conditions:

!=(not-equal), =?(equals?), >, >=, =<,<

3. The Grammar of the Language (The BNF Notation):

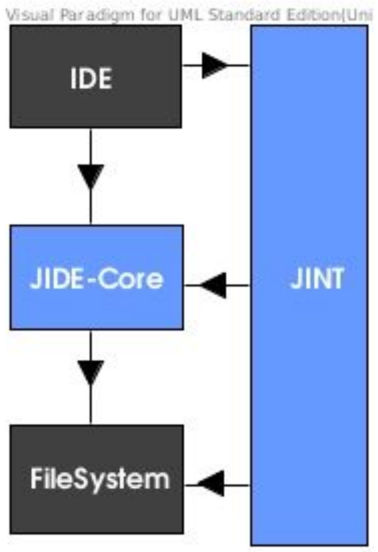
```
<merge_list> ::= <merge><merge_list>|e
<merge> ::= @merge '<identifier>.jlb'|e
<if> ::= if(<cond>)beginif <code> <else_elif> endif
<else_elif> ::=<elif_list> else <code>|e
<elif_list> ::= <elif><elif_list_prime>|e
<elif_list_prime> ::= <elif_list>|e
<elif> ::= elif (<cond>) <code>
<while> ::= while (<cond>) beginwhile <code> endwhile
for <identifier> in <number> .. <number> beginfor <code> endfor
do <code> while (<cond>);
<function> ::= routine <identifier> <parameter_list> begin <code> end ;
<class> ::= class <identifier> <decleration_list> beginclass <constructor> <function_list> endclass ;
<constructor> ::= constructor <identifier> <parameter_list> begincons <code> endcons;
<decleration> ::= <identifier> := <instance_decleration_datatype> <value_zero>;
<parameter_list> ::= <parameter><parameter_list_prime>|e
<parameter_list_prime> ::= <parameter_list>|e
<parameter> ::= <identifier> := <instance_decleration_datatype><value_zero><passing_with_braces>;
<passing_with_braces> ::= ( <passingtype> ) | e
<ops> ::= <complexoperators><identifier>;|<ops_append><ops_prime>|
<identifier><singleoperator><value>;|<identifier>=<ops>
<ops_prime> :: +<ops_append><ops_prime>|-<ops_append><ops_prime>|e
<ops_append>:= <ops_term><ops_append_prime>
<ops_append_prime> ::= *<ops_term><ops_append_prime>|
/<ops_term><ops_append_prime>|.<ops_term><ops_append_prime>|e
<ops_term> ::= id | number | (<ops>)
<passing_type> ::= ino|out|in
<number> ::= <digit><number>|<number>.<number>|<number>^<number>|e
<number> ::= <digit><number>|e
<number_cat> ::= <number><number_beta>
<number_beta> ::= .<number>|^<number>|e
<decleration_list> ::= <decleration_list><decleration>|<decleration>|e
<code> ::= <line><code_prime>|e
<code_prime> ::= <code>|e
<line> ::= <if> | <while> | <for> | <do> | <....> |e
<value> ::= <number>|<identifier>
<value_zero> ::=<value>|e
<instance_decleration_datatype> ::= <datatype>|<identifier>
<datatype> ::= integer|matrix|float|space|vector|double|character|.......
<cond> ::= <simple>|<compound>
<simple> ::= <number>|<identifier>
<compound> ::=
<ops_append><ops_prime>|<identifier>=<ops_append><ops_prime>|<identifier><singleoperator><
value>|<com_append><com_prime>
<com_prime> ::= < <com_terminal><com_term_prime>|>
<com_terminal><com_term_prime>|>= <com_terminal><com_term_prime>| <=
<com_terminal><com_term_prime>| != <com_terminal><com_term_prime>| !
<com_terminal><com_term_prime>|=? <com_terminal><com_term_prime>|e
<com_append>::= <com_term><com_append_prime>
<com_append_prime> ::= ||<com_term><com_append_prime>|e
<com_term> ::= <com_terminal><com_term_append>
<com_term_append> ::= &&<com_append><com_prime>|e
<com_terminal> ::= <identifier> | <number> | (<compound>)
<identifier> ::= <char><symbol>
```

```
<symbol> ::= <char><symbol>|<digit><symbol>|e
<identifier_list> ::= ,<identifier><identifier_list>|e
<func_calling> ::= <identifier>(<identifier><identifier_list>);
<char> ::= _|a|.......|z|A|....|z
<digit> ::= 0|....|9
```

Jebr Integrated Development Environement (JIDE)

JIDE core systems were forked from MEOW, which is a very minimal development environment that 'Umar developed while taking “Compiler Design” course. Meow is a free-software minimized educational IDE licensed under the GNU General Public License -version 3 - GPLv3. The project offers a graphical UI implemented in Java programming language based on the "Khwarizmi Baby C Compiler" - (kbcc) which is a single-pass toy compiler implemented in "x86/GAS Syntax Assembly". The compiler aims to transfer c code into a target abstract stack machine code, with a report showing the details of each compiler phase. MEOW is compatible with GNU/Linux x86 machines. Based on it, JIDE offers an interface for the interpreter and another one for JINT parser, where users can write classes and functions. The first interface offers basic functionality and uses basic Java Swing components, while the second interface offers syntax-highlighting and line-numbering.

JIDE as a system consists of two parts; the IDE and the JIDE core. The IDE provides a very easy-to-use environment for users and programmers allowing them to write code, customize their experience and many other options. The core is where those actions get executed. The core of the system integrates with JINT and the filing system in order to get the whole system to work properly. As shown in figure[7:1] JIDE is a two layered software,where data source differs for each layer.



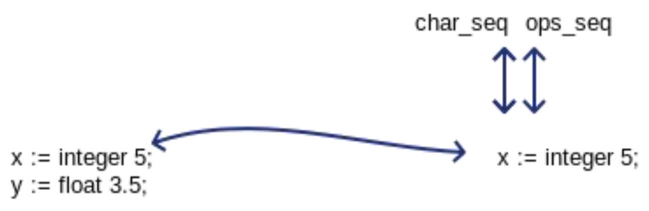
Figure[7:1]: Context-Flow-Model of JIDE.

Ease of use, reliability, robustness and portability are the main properties that JIDE tries to offer. The interface is so simple and easy to figure out, yet JIDE will have a small tool that identifies where do users most ask for help and report the data to the designers, so they can fix that specific issue. JIDE is developed using Java so it offers high portability on the three main x86 32-bit operating systems; MS Windows and GNU/Linux.

Jebr Interpreter (JINT)

The JINT interpreter is based on the Cyclone compiler, which Fahed developed while taking “Compiler Design” course. The interpreter consists of four main levels; a Lexical Analyzer, which converts a sequence of characters into a sequence of tokens, i.e. meaningful character strings, a parser, which analyses a string of symbols conforming to the rules of the grammar, a semantic analyzer, which adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings, a code generator, which translates Jebr code to MIPS 32-bit instructions.

JINT's lexical analyzer contains two main phases; the preprocessing phase and the standard phase. The first eliminates whitespace and process merging libraries, while the other tokenize the file and recognize the type of each token. The lexical analyzer reads the file line by line and then assign two pointers on each line, one associated to identify operators and the other is associated to identify every thing else, as shown in figure[8:1], and then the lexical returns the longest match possible.



Figure[8:1]: Lexical analyzer tokenizing process.

The interpreter is based on a LL sequential recursive parser. As a consequence of its sequential nature, the parser had some efficiency troubles, but by using effective algorithms, the parser recorded an excellent time for parsing huge amount of code. The parser has a fundamental method called “code” and each statement and control structure is represented as an independent method, which checks if this specific statement is considered correct, if not the parser identifies the errors knowing the place of each error in the code segment. Thus, the interpreter automatically can guess how to correct the code. The fundamental method contains instructions to direct JINT to choose the correct method to evaluate the current statement, which the lexical analyzer just tokenized.

Project License

The project is fully licensed under the GNU General Public License version-3 (GPLv3)[3]. The GNU General Public License (GPL) is a computer software copyleft license. This license lets the user of the software use a program in many of the same ways as if it were public domain. They can use it, change it, and copy it. They can also sell or give away copies of the program with or without any changes they made to it. The license lets them do this as long as they agree to follow the terms of the license. The GPL was created by Richard Stallman. The current version is version 3, created in 2007, although some software still uses version 2, created in 1991.

There are two main terms to the license. Both apply to giving the program away or selling it.

1. A copy of the source code or written instructions about how to get a copy must be included with the software. If the software is able to be downloaded from the internet, the source code must also be available for downloading.
2. The license of the software can not be changed or removed. It must always use the GPL.

If the user does not agree to follow the GPL, they can still use the software under copyright laws. They can use it and make copies or changes to it for themselves, but they can not give it away or sell it. They also can not change the license.

The logo of the Jebr language is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License[4].

[3] *(for more info about the GNU General Public License v3 “GPLv3”, take a look at Appendix C).*
[4] *(for more info about Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, take a look at Appendix D).*

Appendices.

Appendix A: Programming Paradigms:

A programming paradigm is a fundamental style of computer programming, a way of building the structure and elements of computer programs. Capabilities and styles of various programming languages are defined by their supported programming paradigms; some programming languages are designed to follow only one paradigm, while others support multiple paradigms.

Programming paradigms that are often distinguished include imperative, declarative, functional, object-oriented, logic and symbolic programming.

Different approaches to programming have developed over time, being identified as such either at the time or retrospectively. An early approach consciously identified as such is structured programming, advocated since the mid 1960s. The concept of a "programming paradigm" as such dates at least to 1978, in the Turing Award lecture of Robert W. Floyd, entitled The Paradigms of Programming, which cites the notion of paradigm as used by Thomas Kuhn in his The Structure of Scientific Revolutions (1962).

The lowest level programming paradigms are machine code, which directly represents the instructions (the contents of program memory) as a sequence of numbers, and assembly language where the machine instructions are represented by mnemonics and memory addresses can be given symbolic labels. These are sometimes called first- and second-generation languages.

Machine Language Paradigm:

In the 1960s, assembly languages were developed to support library COPY and quite sophisticated conditional macro generation and pre-processing capabilities, CALL to (subroutines), external variables and common sections (globals), enabling significant code re-use and isolation from hardware specifics via use of logical operators such as READ/WRITE/GET/PUT. Assembly was, and still is, used for time critical systems and frequently in embedded systems as it gives the most direct control of what the machine actually does.

Procedural Languages Paradigm:

The next advance was the development of procedural languages. These third-generation languages (the first described as high-level languages) use vocabulary related to the problem being solved. For example,

- COBOL (COmmon Business Oriented Language) – uses terms like file, move and copy.
- FORTRAN (FORmula TRANslation) – using mathematical language terminology, it was developed mainly for scientific and engineering problems.
- ALGOL (ALGOrithmic Language) – focused on being an appropriate language to define algorithms, while using mathematical language terminology and targeting scientific and engineering problems just like FORTRAN.
- PL/I (Programming Language One) – a hybrid commercial/scientific general purpose language supporting pointers.
- BASIC (Beginners All purpose Symbolic Instruction Code) – it was developed to enable more people to write programs.
- C – a general-purpose programming language, initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.

All these languages follow the procedural paradigm. That is, they describe, step by step, exactly the procedure that should, according to the particular programmer at least, be followed to solve a specific problem. The efficacy and efficiency of any such solution are both therefore entirely subjective and highly dependent on that programmer's experience, inventiveness and ability.

Object-oriented Languages Paradigm:

Later, object-oriented languages (like Simula, Smalltalk, C++, C#, Eiffel and Java) were created. In these languages, data, and methods of manipulating the data, are kept as a single unit called an object. The only way that a user can access the data is via the object's "methods" (subroutines). Because of this, the internal workings of an object may be changed without affecting any code that uses the object. There is still some controversy by notable programmers such as Alexander Stepanov, Richard Stallman and others, concerning the efficacy of the OOP paradigm versus the procedural paradigm. The necessity of every object to have associative methods leads some skeptics to associate OOP with software bloat. Polymorphism was developed as one attempt to resolve this dilemma.

Since object-oriented programming is considered a paradigm, not a language, it is possible to create even an object-oriented assembler language. High Level Assembly (HLA) is an example of this that fully supports advanced data types and object-oriented assembly language programming – despite its early origins. Thus, differing programming paradigms can be thought of as more like "motivational memes" of their advocates – rather than necessarily representing progress from one level to the next. Precise comparisons of the efficacy of competing paradigms are frequently made more difficult because of new and differing terminology applied to similar (but not identical) entities and processes together with numerous implementation distinctions across languages.

Appendix B: Introduction to Compilers:

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be build by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in.
A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.
Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The phases of a compiler :

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-defined interfaces. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate module. Some of these modules are written by hand, while others may be generated from specifications. Often, some of the modules can be shared between several compilers.

A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

Lexical analysis This is the initial part of reading and analysing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

Syntax analysis This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

Type checking This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that doesn't make sense given the type of the variable, such as trying to use a boolean value as a function pointer.

Intermediate code generation The program is translated to a simple machine-independent intermediate language.

Register allocation The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

Machine code generation The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

Assembly and linking The assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined.

The first three phases are collectively called the frontend of the compiler and the last three phases are collectively called the backend. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimisations and transformations on the intermediate code.

Each phase, through checking and transformation, establishes stronger invariants on the things it passes on to the next, so that writing each subsequent phase is easier than if these have to take all the preceding into account. For example, the type checker can assume absence of syntax errors and the code generation can assume absence of type errors.

Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself, so we will not further discuss these phases in this book.

Interpreters:

An interpreter is another way of implementing a programming language. Interpretation shares many aspects with compiling. Lexing, parsing and type-checking are in an interpreter done just as in a compiler. But instead of generating code from the syntax tree, the syntax tree is processed directly to evaluate expressions and execute statements, and so on. An interpreter may need to process the same piece of the syntax tree (for example, the body of a loop) many times and, hence, interpretation is typically slower than executing a compiled program. But writing an interpreter is often simpler than writing a compiler and the interpreter is easier to move to a different machine (see chapter 10), so for applications where speed is not of essence, interpreters are often used.

Compilation and interpretation may be combined to implement a programming language: The compiler may produce intermediate-level code which is then interpreted rather than compiled to machine code. In some systems, there may even be parts of a program that are compiled to machine code, some parts that are compiled to intermediate code, which is interpreted at runtime while other parts may be kept as a syntax tree and interpreted directly. Each choice is a compromise between speed and space: Compiled code tends to be bigger than intermediate code, which tend to be bigger than syntax, but each step of translation improves running speed.

Using an interpreter is also useful during program development, where it is more important to be able to test a program modification quickly rather than run the program efficiently. And since interpreters do less work on the program before execution starts, they are able to start running the program more quickly. Furthermore, since an interpreter works on a representation that is closer to the source code than is compiled code, error messages can be more precise and informative.

Appendix C: GNU General Public License:

GNU GENERAL PUBLIC LICENSE

```
Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
```

The Corresponding Source for a work in source code form is that
same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met. This License explicitly affirms your unlimited
permission to run the unmodified Program. The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work. This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force. You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright. Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under
the conditions stated below. Sublicensing is not allowed; section 10
makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,

model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any , procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because

```
    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
```

but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Appendix D: Creative Commons License:

Index