UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Glass Cat – a tool for interactive visualization of the execution of Oz programs in the Pythia platform

Pierre Bouilliez

Thesis submitted for the grade of master degree
in computer science and engineering in networking and security.

Supervisor: Prof. Peter Van Roy

Co-supervisor: Dr Ir. Sébastien Combéfis

Reader: Dr Ir. Virginie Van den Schrieck

Louvain-la-Neuve

Academic year 2013-2014

"If you can dream it, you can do it. Always remember that this whole thing was started with a dream and a mouse."

— *Walt Disney*

Acknowledgments

A master's thesis is not only the biggest project we have to lead during our curriculum. It is also the culmination of our years at University. Therefore, this work is the product of the experience acquired for many years. This is it ... THE experience, what we get from sharing the knowledge that forges our mind and makes us thrive. This is why I want to thanks a lot of people for what we have shared, but I will limit myself to those who have a link with this thesis.

First, I wish to express all my sincere gratitude to **Prof. Peter Van Roy** for his dedication to Oz, always making it writing another story. Moreover, he was always there to answer to my questions despite a very busy schedule.

I would like to give a special thank to **Dr Ir. Sébastien Combéfis** for all the wonderful experiences he shared with me along these years and many more to come. He was undoubtedly my mentor during my years at University, the one who supported me even when it was not easy and made me discover the new era of education via the Internet. Maybe there will be more tasty things in the future...

Then, **Dr Ir. Virginie Van den Schrieck** for the time she will take reading this thesis.

And last but not least, to all my friends who unwind and encourage me throughout this year. Especially **Guillaume Simons** whom I have spent a lot of time this year, working on our respective thesis, sharing our experience.

Pierre Bouilliez
Louvain-la-Neuve, June 2014.

Abstract

Nowadays, education is changing thanks to the Internet. The number of Massive Open Online Courses (MOOCs) increases every year. It opens the doors to develop new tools to help students learning. Moreover, learning programming requires first to understand precisely how the program works. The semantics of a programming language defines the meaning of the language with a mathematical representation. Semantics is used to get a precise and formal understanding of programs.

Ultimately, learning requires practice and students must have a way to check their answers. Currently, many softwares with automatic feedbacks are being developed and made available online. Above that, visualization programs can bring added-value to facilitate the learning of complex concepts. Furthermore, many researches suggest that only using "view-only" programs to learn is not enough, and prone the implication of students with the code.

In addition, a programming language called Oz offers an opportunity to reason about the correctness of a program. As it covers three of the most important programming paradigms, this language is learned by second-year students in engineering at the Université catholique de Louvain. This year, the first version of this course was also available as a MOOC.

This master's thesis proposes Glass Cat, a web-based tool for visualizing the execution of Oz programs. Students can write their own codes and see how semantics is processed at each step.

Glass Cat can interpret and display the semantics of 50 % of the global kernel language of Oz. It corresponds to the content of the first course given with this language. This tool can also be integrated into the INGI Programming Training Server Pythia which offers exercises with smart feedbacks to students.

Contents

Chapter 1

Introduction

The Internet ! What a powerful thing with more than thirty years of experiences, managing lives of billions of people, storing billions of websites. One human lifetime is not enough to visit them all. The internet has become probably the most brilliant brain on earth, connecting people in the world. However there is no precise platform to share this knowledge.

Learners ! Since you were in your mother's womb, you have never stopped learning. We are all learners, but sometimes we do not know where to find answers to our questions. We are going to school to learn new things, to find explanations. Knowledge has to be spread.

Nowadays, these two entities finally met each other and we can find courses on the Internet. Universities are no longer merely geographical places but are accessible from everywhere, to everybody. This master's thesis perfectly fits this belief and proposes a tool to help people understanding how a programming language works.

More specifically, we developed a website to be integrated into the Pythia platform [CLCdSM12], as part of a MOOC (Massive Open Online Course) taught at the Université catholique de Louvain by Prof. Peter Van Roy [VR11]. During this course, students learn a programming language called Oz. This language covers three of the most important programming paradigms and is therefore interesting to learn useful concepts. Moreover, Oz defines tools to reason about both complexity and correctness of a program.

The platform developed in this master's thesis interprets an elementary part of Oz (the kernel language) and shows how execution steps are processed in the memory. We decided to call it *Glass Cat*.

1.1 Objectives

The main objective of this master's thesis is to develop a tool for the interactive visualization of Oz programs to be integrated into the Pythia platform. We can

define a few subgoals to achieve, in order to complete the main objective:

Goal 1: Interpret the subset of the kernel language of Oz covered by the MOOC.

Goal 2: Display the semantics of the code written by the user.

Goal 3: Be interesting for a student.

Goal 4: Integrated into Pythia.

1.2 Structure of this thesis

After presenting the context and the objectives we wish to achieve, the remainder of this master's thesis is organized as follows:

Chapter 2 – This chapter presents Oz, the programming language that Glass Cat interprets. It does not explain the complete language but only a part of the kernel language. We present some operations that can be done and how to code them. Afterwards we show the semantics which is used to define the meaning of the kernel language through the execution of its statements by an abstract machine.

Chapter 3 – In order to fulfil Goals 1 and 4, we have to interpret Oz inside a webpage. Therefore, this chapter introduces concepts related to the parsing of a programming language. We see how the parser can understand the code and manipulate it correctly. We compare some parser generators and choose the one that fits best with our objectives.

Chapter 4 – This chapter is about the design of the User Interface (UI) of Glass Cat in order to satisfy Goals 2 and 3. First, we define the concept of visual programming and how it can be helpful for students, *i.e.* improving their engagement. This definition leads us into a short presentation of some visual programs. Later, we can finally present our UI design choices for Glass Cat.

Chapter 5 – As Glass Cat has to be incorporated into Pythia, this chapter presents briefly this platform. We will see how Glass Cat can bring concrete added-value to this educational website.

Chapter 6 – This chapter presents the implementation of Glass Cat. We look at how we made it come true. We decomposed this chapter in two sections. The first one is about the back-end which encloses everything the user does not see. The second section is about the front-end and contains what the user can see. This chapter gives a global description of how the Glass Cat platform works. A more complete description is given in Appendix A.

Chapter 7 – Finally, we evaluate Glass Cat. We see how many items of the global kernel language of Oz are implemented and if it can correctly compute the semantics for numerous tests. Moreover, we concisely inspect its speed and its memory consumption. We also provide a brief explanation of optimizations that can make it faster.

Chapter 8 – This concluding chapter sums up what we have learned in this master's thesis and gives perspectives for the future.

1.3 Why Glass Cat?

Finding a name for the platform developed in this master's thesis was difficult. We could have chosen a long and explicit name in order to describe what it will talk about at first sight. Nevertheless, such a name is not useful and does not last long. Also, if we delve more deeply in the origin of *Glass Cat*, we can find a nice metaphor.

As this thesis is linked to the Oz programming language, we looked at names of characters from books written by L. Frank Baum [Bau13]. One day, a magician, Dr Pipt wanted to test his *powder of life*. So he threw it on a cat made of glass. Since this day, Glass Cat is alive with a transparent body that reveals her heart and her brain [Wik13].

This name jumped out at me because this master's thesis wants to make the heart of Oz visible to the students while Glass Cat has a transparent body.

Chapter 2

Oz

> "The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language."
>
> — *Donald Knuth*

Oz is a multiparadigm programming language, which was first developed by Gert Smolka and his students in 1999 [VR09]. It was a good starting point to make the network distribution transparent. Nowadays, many new ideas have already come to add innovative functionalities to Oz, and its Mozart compiler currently implements Oz 3.

Given that Oz covers three of the most important programming paradigms (functional, object-oriented, and dataflow concurrent programming), it was agreed in 1999 by Peter Van Roy and Seif Haridi to introduce it in universities. Therefore, since 2003, engineering students at the Université catholique de Louvain study this language during their second-year bachelor [VR11]. Moreover, Oz provides a formal semantics of the kernel language which can be found in [VRH04].

This chapter gives a brief explanation of Oz. In particular, we focus on the kernel language of Oz in order to meet Goal 1. This kernel language defines in a simple mathematical structure what the program does. In this chapter, you will not find high-level codes like the ones you could use in Mozart.

2.1 The kernel language of Oz

This section provides a brief introduction to the kernel language of Oz.

2.1.1 Browser

As a first introduction to Oz, it is mandatory to talk about the browser which is a *special window*. The browser is called with a one-argument function `Browse`. Its unique argument is what the browser has to display. Listing 2.1 shows an example of how to call the browser to display the number `42`.

```
1 {Browse 42}
```

Listing 2.1. Use of the browser in Oz.

2.1.2 The declarative model

The basic concept of the declarative model is "*if a program works today, then it will work tomorrow*", *i.e.* the values of variables do not change and functions do not change their behaviour [VR12].

Table 2.1 shows the declarative kernel language, where ⟨x⟩ is an identifier and ⟨v⟩ is a value (number, record, procedure).

⟨s⟩	::=	`skip`	Empty statement
	|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
	|	`local` ⟨x⟩ `in` ⟨s⟩ `end`	Variable creation
	|	⟨x⟩$_1$ `=` ⟨x⟩$_2$	Variable-variable binding
	|	⟨x⟩ `=` ⟨v⟩$_2$	Value creation
	|	`if` ⟨x⟩ then ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Conditional
	|	`case` ⟨x⟩ `of` ⟨pattern⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Pattern matching
	|	`{`⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$`}`	Procedure application

Table 2.1. The declarative kernel language (from [VRH04]).

Let us define more precisely some of these statements.

Identifiers

Identifiers, also referred to as *variable identifiers*, are textual references used to store values in memory.

First, the identifier has to be declared within the keywords `local` ⟨id⟩ `in` where ⟨id⟩ is the name of the identifier we want to create. Then, we can assign a value to this identifier (a number, an expression...) with ⟨id⟩`=`⟨value⟩ to store it in the memory. This identifier will be available as long as the `end` keyword is not encountered and its value cannot be changed. A typical example of an identifier is:

```
local X in
    X = 3
end
```

Listing 2.2. Declaration and use of an identifier in Oz.

We can notice that Oz is dynamically typed, *i.e.* variables do not have to be declared with a type such as in Java. All variable types are known at compilation time.

The `if` statement

The `if` statement is used to evaluate an *activation condition* and performs some statements if the condition is `true` and other statements if it is `false`. In Oz, it is written:

$$\texttt{if } \langle \text{x} \rangle \texttt{ then } \langle \text{s} \rangle_1 \texttt{ else } \langle \text{s} \rangle_2 \texttt{ end}$$

Listing 2.3 gives an implementation of an `if-then-else` statement.

```
local Y in
  local X in
    X = (4==4)
    if X then
      Y = 3
    else
      Y = 5
    end
  end
end
```

Listing 2.3. The `if-then-else` statement in Oz.

Records

Sometimes, dealing with a single value is not sufficient, so we want to treat data structures. These data structures are called records and start with a label followed by a set of pairs of features and variable identifiers. Examples of records are `person(age:X1 name:X2)`, `person(1:X1 2:X2)`, `'|'(1:H 2:T)`, `'#'(1:H 2:T)`. In the kernel language, there is no tuples nor lists as they can both be implemented with records.

Moreover, you can access the variable identifier in the record simply with the identifier which stores the record followed by a point and the name of the feature you want to access. Listing 2.4 gives an example of how a record can be created and used.

```
local Y in
  local A in
    local B in
      A = 3
      B = 'M'
      Y = person(age:A sex:B)
      {Browse Y.1}
    end
  end
end
```

Listing 2.4. Use of a record in Oz.

Case statement

Another way to check a condition is done with a *pattern matching*:

`case` ⟨x⟩ `of` ⟨pattern⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`

If ⟨x⟩ matches ⟨pattern⟩ then, we will go through the statement ⟨s⟩$_1$ that follows. Otherwise, we go to ⟨s⟩$_2$.

This is a very impressive concept, very often used with records as it can decompose them according to the pattern ⟨literal⟩`(`⟨feature⟩$_1$`:`⟨x⟩$_1$ ... ⟨feature⟩$_n$`:`⟨x⟩$_n$`)`. Moreover, these extra local variables are created and bound respectively to each of the variable identifier of the record. Listing 2.5 shows an example of a pattern matching on a record. Identifier `H` is linked to the identifier which is at feature `1`, *i.e.* `A`, and `T` is bound to the variable `B`.

```
local Y in
  local A in
    local B in
      A = 3
      B = nil
      Y='|'(1:A 2:B)
      case Y of '|'(1:H 2:T) then
        local X2 in
          X2=H
        end
      else
        local X4 in
          X4 = 4
        end
      end
    end
  end
end
```

Listing 2.5. Case statement with a record in Oz.

Procedure

A more complex construction is the one which gives us the opportunity to make generic codes. Procedures are processes defined by the following statement:

`P = proc{$ X Y}` ⟨s⟩ `end`

where `P` is the name given to the procedure. The `$` means that the procedure value is anonymous, *i.e.* created without being bound to an identifier. `X` and `Y` are the parameters of this procedure.

Procedures can take arguments which can be already bound to a value as well as unbound variables that will be linked to an output value (written with `?` as prefix). It implies that a procedure can have more than one output value.

Listing 2.10 shows an example of the definition and use of a procedure. This code defines a new procedure `Sum` to make the sum of two values and returns it via the unbound identifier `Z`. This procedure is called with `{Sum X Y Z}` at line 10. The next line displays the value of this sum which is `7`.

```
1  local Sum in
2    Sum=proc{$ X Y ?Z}
3      Z = X+Y
4    end
5    local X in
6      local Y in
7        local Z in
8          X = 3
9          Y = 4
10         {Sum X Y Z}
11         {Browse Z}
12       end
13     end
14   end
15 end
```

Listing 2.6. Definition and use of a procedure in Oz.

Attentive readers may have observed that we have already used procedures in this chapter. In fact, `Browse` is a procedure as it respects this notation.

2.1.3 The declarative model with explicit state

With the *explicit state*, we can keep a reference to the same identifier and change its value without changing its name. In Oz, what is called in other languages a variable is defined as a *memory cell*. Table 2.2 gives the declarative kernel with explicit state.

⟨s⟩	::=	`skip`	Empty statement
	|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
	|	`local` ⟨x⟩ `in` ⟨s⟩ `end`	Variable creation
	|	⟨x⟩$_1$ `=` ⟨x⟩$_2$	Variable-variable binding
	|	⟨x⟩ `=` ⟨v⟩$_2$	Value creation
	|	`if` ⟨x⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Conditional
	|	`case` ⟨x⟩ `of` ⟨pattern⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Pattern matching
	|	`{`⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$`}`	Procedure application
	|	`{NewCell` ⟨x⟩ ⟨y⟩`}`	**Cell creation**

Table 2.2. The declarative kernel language with explicit state (from [VRH04]).

Cells

First, we can create a new cell with the procedure `NewCell` with the initial value as a parameter. Then, the assignment operation `:=` puts a new value in the cell. Furthermore, the access operation `@` gets the current value stored in the cell.

Listing 2.7 shows an example of how a cell can be used. This short program browses the value of the cell `Y` whose value is initially `0` at line 4, then `3` at the next line. So line 6, shows `3`.

```
1 local Y in
2   local X in
3     X = 0
4     Y = {NewCell X}
5     Y := 3
6     {Browse @Y}
7   end
8 end
```

Listing 2.7. Use of a cell in Oz.

2.1.4 The data-driven concurrent model

In the declarative computation model, there is just one statement that can be executed over a single-assignment store, *i.e.* it is sequential. The *data-driven concurrent model* allows more than one statement to reference the same store. This makes all the statements to be executed roughly "at the same time". Table 2.3 gives the data-driven concurrent kernel language where the new concept is the `thread` statement.

Threads

With the `thread` statement, we can make executions taking place at the same time. A thread can be created around another statement with the keywords `thread` ⟨s⟩ `end`.

⟨s⟩	::=	`skip`	Empty statement
	|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
	|	`local` ⟨x⟩ `in` ⟨s⟩ `end`	Variable creation
	|	⟨x⟩$_1$ `=` ⟨x⟩$_2$	Variable-variable binding
	|	⟨x⟩ `=` ⟨v⟩$_2$	Value creation
	|	`if` ⟨x⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Conditional
	|	`case` ⟨x⟩ `of` ⟨pattern⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Pattern matching
	|	`{`⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$`}`	Procedure application
	|	`thread` ⟨s⟩ `end`	**Thread creation**

Table 2.3. The data-driven concurrent kernel language (from [VRH04]).

Listing 2.8 shows an example of how it can be used. The `thread` at line 5 works in parallel of the main `thread` (the one of the program). So, we can wonder, what is displayed in the browser? We can think that the first call to the `Browse` procedure will be the first to be displayed, but in fact, it can either be `111` or `222`.

```
1 local B in
2   local C in
3     B = 111
4     C = 222
5     thread {Browse B} end
6     {Browse C}
7   end
8 end
```

Listing 2.8. Use of threads in Oz.

2.2 The semantics of Oz

The semantics of the kernel language let the programmer reason about both complexity and correctness of a program. There are four widely used approaches to language semantics described in [VRH04].

In this work, we only focus on the *operational semantics*. This approach defines the meaning of the kernel language through the execution of its statements by an abstract machine. First, we define the basic concepts of this kind of machine. Then, we show all the operations that can be performed on the semantics. Afterwards, an example illustrates the steps of the execution of a program. Finally, we want to ensure that unused values are no more in the memory after a certain time. Therefore, we introduce the garbage collector to perform automatic reclaiming.

2.2.1 The abstract machine

In order to represent execution states, we need the following concepts:

- A *stack of semantic statements* ST: all the semantic statements to be evaluated.
- A *store*: a set of stored variables. There are two kinds of store (1) a single-assignment store σ for the declarative model (2) a mutable store μ for the declarative model with explicit state, *i.e.* cells. The difference between them is that variables can only be bound once in the first store, whereas the second store contains pairs of the form $x : y$ where x and y are variables of the single-assigment store with y representing a cell. These stores have variables that can be bound or unbound.
- An *environment* E: a mapping from variable identifiers to variables in the store (*e.g.* $E = \{\texttt{X} \rightarrow x\}$ means that the identifier `X` refers to the variable x in the store).

Then, we can combine those tools together to get:

- A *semantic statement*: a pair $(\langle \text{s} \rangle, E)$ where $\langle \text{s} \rangle$ is a statement. It relates the statement to what it refers in the store.
- An *execution state*: a pair (ST, σ) so that it gives a representation of which variables values the statement sees.
- A *computation*: a sequence of execution states $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \ldots$

Now that we know some concepts about execution states, we can define some operations that statements enforce. Moreover, we can highlight that every computation step is *atomic*, *i.e.* the computation happens instantaneously, there is no intermediate state.

2.2.2 Operations

Let us first define how semantic statements are performed. First, a program is a statement $\langle \text{s} \rangle$ that has an initial state (empty value is written ϕ):

$$([(\langle \text{s} \rangle, \phi)], \phi)$$

where the stack ST is represented between square brackets and contains the semantic statement to be analyzed. The last entry is the store that has an empty value as no variable can be accessed.

At each step, the first element of the stack ST is popped, and the right operation according to its pattern is performed. The final execution state (if there is one) has nothing more to do, and so its semantic stack is empty.

Now, we can take a look at operations that can be done during the execution of a program. We can split those operations in two partitions. Some of them are *non-suspendable statements*, *i.e.* they do not need to evaluate a value to process, they can never suspend (*e.g.* `skip`, sequential composition, variable declaration, variable-variable binding, value creation). At the opposite, *suspendable statements* are those which can suspend, *i.e.* wait for a value to be bound, until the *activation condition* becomes true for the execution to continue (*e.g.* `if`, procedure call, pattern matching).

The `skip` statement

The semantic statement is:

$$(\texttt{skip}, E)$$

Once this statement is popped, nothing is added in the environment neither in the store, it just jumps to the next statement.

Sequential composition

The semantic statement is:

$$(\langle \text{s} \rangle_1 \ \langle \text{s} \rangle_2, E)$$

When there are two consecutive statements in the semantic stack, we decompose them sequentially (*i.e.* pushing $(\langle \text{s} \rangle_1, E)$ on the stack and then $(\langle \text{s} \rangle_2, E)$).

Variable declaration

The semantic statement is:

$$(\texttt{local}\ \langle \text{x} \rangle\ \texttt{in}\ \langle \text{s} \rangle\ \texttt{end}, E)$$

First, this statement creates a new variable $\langle \text{x} \rangle$ in the store. Second, the environment is extended with a mapping between the identifier `X` which represents the value of $\langle \text{x} \rangle$ and the variable identifier in the store x: $E' = E \cup \{\texttt{X} \rightarrow x\}$. And then the semantic statement $(\langle \text{s} \rangle, E')$ is pushed on the stack.

Variable-variable binding

The semantic statement is:

$$(\langle \text{x} \rangle_1 \ = \ \langle \text{x} \rangle_2, E)$$

It results in a binding of the two environments in the store $E(\langle \text{x} \rangle_1)$ and $E(\langle \text{x} \rangle_2)$.

Value creation

The semantic statement is:

$$(\langle \text{x} \rangle \ = \ \langle v \rangle, E)$$

Where $\langle v \rangle$ is a value that is either a record, a number, or a procedure. This execution creates a new variable x in the store with a reference to the value represented by $\langle v \rangle$.

The `if` statement

The semantic statement is:

$$(\texttt{if}\ \langle \text{x} \rangle\ \texttt{then}\ \langle \text{s} \rangle_1\ \texttt{else}\ \langle \text{s} \rangle_2\ \texttt{end}, E)$$

First, the activation condition $E(\langle \text{x} \rangle)$ must be determined (otherwise the execution is suspended) and its value must be `true` or `false`, otherwise it raises an error. If it is `true`, push $(\langle \text{s} \rangle_1, E)$ on the stack, or if it is `false` push $(\langle \text{s} \rangle_2, E)$.

Procedure call

The semantic statement is:

$$(\{\langle \text{x} \rangle\ \langle \text{y} \rangle_1 ... \langle \text{y} \rangle_n\}, E)$$

When a procedure is called by the semantic statement, if the activation condition is `true` ($E(\langle \text{x} \rangle)$ is determined) but the number of arguments is different from the ones in the declaration of the procedure, execution raises an error.

Then, if $E(\langle \text{x} \rangle)$ has the form

$$(\texttt{proc \{\$}\ \langle \text{z} \rangle_1 ... \langle \text{z} \rangle_n \texttt{\}}\ \langle \text{s} \rangle\ \texttt{end}, CE)$$

push $(\langle \text{s} \rangle, CE + \texttt{\{}\langle \text{z} \rangle_1 \rightarrow E(\langle \text{y} \rangle_1), ..., \langle \text{z} \rangle_n \rightarrow E(\langle \text{y} \rangle_n)\texttt{\}})$ on the stack.

Pattern matching

The semantic statement is:

$$(\texttt{case}\ \langle \text{x} \rangle\ \texttt{of}\ \langle \text{lit} \rangle(\langle \text{feat} \rangle_1 :\ \langle \text{x} \rangle_1 ... \langle \text{feat} \rangle_n :\ \langle \text{x} \rangle_n)\ \texttt{then}\ \langle \text{s} \rangle_1\ \texttt{else}\ \langle \text{s} \rangle_2\ \texttt{end}, E)$$

where $\langle \text{lit} \rangle$ and $\langle \text{feat} \rangle$ stands respectively for the label and the features defined previously. It corresponds to the pattern to be matched (*e.g.* `'|'(1:X 2:Xr)` for lists in the Oz kernel language). If the activation condition is `true` ($E(\langle \text{x} \rangle)$ is

determined), and if the label of $E(\langle \text{x} \rangle)$ is $\langle \text{lit} \rangle$ and its arity is $[\langle \text{feat} \rangle_1 ... \langle \text{feat} \rangle_n]$, push

$$(\langle \text{s} \rangle_1, E + \{\langle \text{x} \rangle_1 \rightarrow E(\langle \text{x} \rangle).\langle \text{feat} \rangle_1, ..., \langle \text{x} \rangle_n \rightarrow E(\langle \text{x} \rangle).\langle \text{feat} \rangle_n\})$$

on the stack.

The `NewCell` operation

The semantic statement is:

$$(\{\texttt{NewCell}\ \langle \text{x} \rangle\ \langle \text{y} \rangle\}, E)$$

First, we have to create a new cell c and bind it to $E(\langle \text{y} \rangle)$ in the store σ. If the binding did well, then add the pair $E(\langle \text{y} \rangle) : E(\langle \text{x} \rangle)$ to the mutable store μ.

The `thread` statement

First, we have to update the concepts of execution state and computation defined in subsection 2.2.1 for multiple semantic stacks:

- The new *execution state* is a pair (MST, σ) where MST stands for multiset of semantic stacks. A multiset is a collection where items can occur more than once, *e.g.* two distinct semantic stacks with identical contents.
- The *computation* is now a sequence of execution states $(MST_0,\ \sigma_0) \rightarrow (MST_1,\ \sigma_1) \rightarrow (MST_2,\ \sigma_2) \rightarrow \ldots$

The execution of a `thread` is represented by Figure 2.1. At each computation step, one semantic stack inside the MST is selected by the scheduler. And if it has the form $[(\texttt{thread}\ \langle \text{s} \rangle\ \texttt{end}, E)] + ST'$, we add a new semantic stack $[(\langle \text{s} \rangle, E)]$ that corresponds to the new thread.

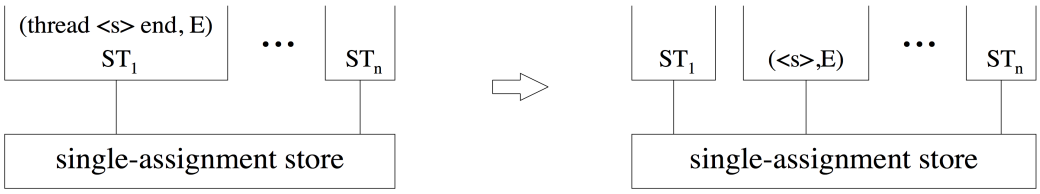


Figure 2.1. Execution of the `thread` statement [VRH04].

More formally, the new computation step is written (with $\uplus$ denoting multiset union):

$$(\{[(\texttt{thread}\ \langle \text{s} \rangle\ \texttt{end}, E)] + ST'\} \uplus MST', \sigma) \rightarrow (\{[(\langle \text{s} \rangle, E)]\} \uplus \{ST'\} \uplus MST', \sigma)$$

2.2.3 Declarative variables vs. dataflow variables

Once bound, a *declarative variable* is a variable in the single-assignment store which is bound to its value throughout the computation and is indistinguishable from its value, *i.e.* if the store is `{x = 1, y = 2}`, doing $x + y$ is the same as doing $1 + 2$.

Dataflow variables can be used before they have been bound. Assume a program that performs first `B = A+1` with `A` not yet bound. The program will wait until the variable gets bound. Therefore, the program returns the same value either if `A = 2` is defined before or after the value creation of `B`. These variables are really powerful in concurrent programming. Assiduous readers might have noticed the link between suspendable statements (*i.e.* active condition) and dataflow variables.

2.2.4 Examples of execution

Now that we have defined concepts of semantics for the kernel language of Oz, let us see how the abstract machine proceeds on some examples, starting with a straightforward one.

A first example

Consider the program in Listing 2.9:

```
1 local A in
2   A = 10
3   local B in
4     B = A+1
5   end
6 end
```

Listing 2.9. Basic code in Oz.

This example can be considered as a statement $\langle s \rangle$ that can be executed by an abstract machine. In the following, $\ell_i - \ell_j$ means the statement from line i to line j.

1. The *initial execution state* is:

$$([(\ell_1 - \ell_6, \phi)], \phi)$$

   The environment and the store are empty.

2. After the variable declaration we have:

$$([(\ell_2 - \ell_5, \{\texttt{A} \rightarrow a\})], \{a\})$$

   The environment contains the binding of the identifier `A` with a, its value in the store.

3. There is a sequential composition of two statements, so we have:

$$([(\texttt{A = 10}, \{\texttt{A} \rightarrow a\})]\ [(\ell_3 - \ell_5, \{\texttt{A} \rightarrow a\})], \{a\})$$

   Statements are decomposed sequentially.
4. Starting with the leftmost statement, we get a binding for `A`:

$$([(\phi, \{\texttt{A} \rightarrow a\})]\ [(\texttt{local B in B = A+1 end}, \{\texttt{A} \rightarrow a\})], \{a = 10\})$$

5. The rightmost statement, is a variable declaration, so we have:

$$([(\phi, \{\texttt{A} \rightarrow a\})]\ [(\texttt{B = A+1}, \{\texttt{A} \rightarrow a, \texttt{B} \rightarrow b\})], \{a = 10, b\})$$

6. There is a new variable binding, with `B = A+1`:

$$([(\phi, \{\texttt{A} \rightarrow a\})]\ [(\phi, \{\texttt{A} \rightarrow a, \texttt{B} \rightarrow b\})], \{a = 10, b = 11\})$$

7. And the final execution state is:

$$([], \{a = 10, b = 11\})$$

Working with procedures

Consider the program in Listing 2.10:

```
local Y in
    Y = proc{$ X}
      local A in
        local B in
          local C in
            A = 4
            C = (X==A)
            if C then
              B = 3
            else
              B = 5
            end
          end
        end
      end
   end
   local X1 in
     X1 = 3
     {Y X1}
   end
end
```

Listing 2.10. Procedures in Oz.

As in the previous example, all the code can be considered as a statement $\langle s \rangle$ that can be executed by an abstract machine.

1. The *initial execution state* is:

$$([(\ell_1 - \ell_{21}, \phi)], \phi)$$

   The environment and the store are empty.
2. After the variable declaration we have:

$$([(\ell_2 - \ell_{20}, \texttt{\{Y} \rightarrow y\texttt{\}})], \{y\})$$

   The environment contains the binding of the identifier `Y` with its value in the store y.
3. There is a sequential composition of two statements so we have:

$$([(\ell_2 - \ell_{16}, \texttt{\{Y} \rightarrow y\texttt{\}})]\ [(\ell_{17} - \ell_{20}, \texttt{\{Y} \rightarrow y\texttt{\}})], \{y\})$$

   Statements are decomposed sequentially.
4. Starting with the left-most statement, we have a binding for `Y`

$$([(\ell_2 - \ell_{16}, \texttt{\{Y} \rightarrow y\texttt{\}})]\ [(\ell_{17} - \ell_{20}, \texttt{\{Y} \rightarrow y\texttt{\}})], \{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi)\})$$

   Note that the contextual environment of `Y` is empty because it has no free identifiers.
5. The rightmost statement starts with a variable declaration, so we have:

$$([(\phi, \texttt{\{Y} \rightarrow y\texttt{\}})]\ [(\texttt{X1 = 3 \{Y X1\}}, \texttt{\{Y} \rightarrow y, \texttt{X1} \rightarrow x1\texttt{\}})],$$

$$\{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi)\})$$

6. Then there is a value creation (we can hide the leftmost which has an empty stack):

$$([(\phi, \texttt{\{Y} \rightarrow y, \texttt{X1} \rightarrow x1\texttt{\}})][(\texttt{\{Y X1\}}, \texttt{\{Y} \rightarrow y, \texttt{X1} \rightarrow x1\texttt{\}})],$$

$$\{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi), x1 = 3\})$$

7. There is now a procedure application:

$$([(\ell_3 - \ell_{15}, \texttt{\{Y} \rightarrow y, \texttt{X} \rightarrow x1\texttt{\}})],$$

$$\{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi), x1 = 3\})$$

8. After the three variables declarations and the two values creations:

$$([(\ell_8 - \ell_{12}, \texttt{\{Y} \rightarrow y, \texttt{X} \rightarrow x1, \texttt{A} \rightarrow a, \texttt{B} \rightarrow b, \texttt{C} \rightarrow c\texttt{\}})],$$

$$\{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi), x1 = 3, a = 4, b, c = \texttt{false}\})$$

9. After executing the condition:

$$([\texttt{B = 5}, \{\texttt{Y} \rightarrow y, \texttt{X} \rightarrow x1, \texttt{A} \rightarrow a, \texttt{B} \rightarrow b, \texttt{C} \rightarrow c\})],$$
$$\{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi), x1 = 3, a = 4, b, c = \texttt{false}\})$$

10. And the final execution state is:

$$([], \{y = (\texttt{proc\{\$ X\}}\ \ell_3 - \ell_{15}\ \texttt{end}, \phi), x1 = 3, a = 4, b = 5, c = \texttt{false}\})$$

2.2.5 Garbage Collector

Sometimes the semantic stack and the store can be very different. This is especially the case when there are recursive calls to a procedure. The semantic stack can remain bound by a constant size (last call recursion) while the store gets bigger at each call.

Moreover, we know that a procedure needs only the information in the semantic stack and in its contextual environment, *i.e.* the part of the store reachable from the semantic stack. Therefore, the garbage collector was introduced in order to reclaim the memory used by unreachable variables. Figure 2.2 shows the lifecycle of a memory block. We see that the garbage collector makes the transition from the inactive state to the free state (that is the reclaim operation).

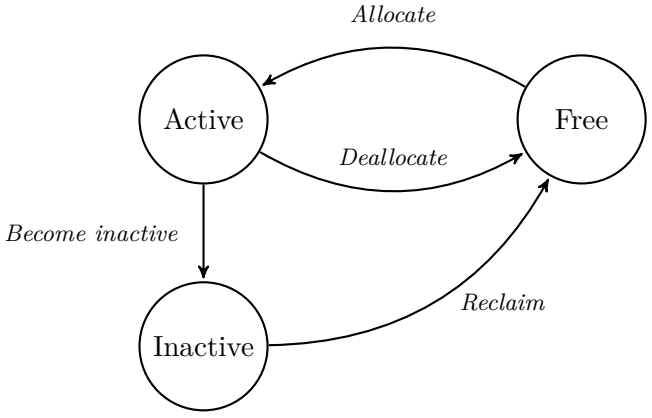


Figure 2.2. Lifecycle of a memory block (from [VRH04]). The reclaim can be done manually or by garbage collection.

When we are dealing with the automatic reclaiming of a garbage collector, two kinds of program error can occur:

- *Dangling reference.* This error happens when a variable is erased from the memory even though it is still reachable.
- *Memory leak.* This error is the opposite of the previous one. It happens when a variable is considered as still reachable but is in fact not.

A typical garbage collector has two phases:

1. Starting from a *root set*, determine the active memory, *i.e.* all data structures that are reachable.
2. Compact the memory by collecting all the active memory blocks in one contiguous block.

Chapter 3

Interpreter

"The face is a picture of the mind with the eyes
as its interpreter."

— *Cicero*

Now that we know how to code in Oz, we need to find a way to understand what is written. For this purpose, we will need an *interpreter*. This tool parses (*i.e.* decomposes) the source code following some rules.

In this chapter, we first discuss theoretical concepts of parsing a language. With this base, we can forge a more accurate opinion on how to generate parsers in JavaScript and how to evaluate them. Finally, a section talks about the parser we choose for Glass Cat and how we can use it to interpret Oz.

3.1 Concepts of parsing a language

When the computer has to understand a programming language, it needs to know what each token (*i.e.* word) means. This operation is called *parsing*. A parser takes the source code as input and builds a data structure to analyze it.

The execution of a parser can be split in two steps (see Figure 3.1). First, the *lexical analysis* splits the input characters stream into meaningful symbols defined by a lexical grammar. This operation generates tokens.

The next stage is the *syntactic analysis*; it checks that tokens form a valid expression. This is done with reference to a language grammar that chooses the appropriate actions to continue [Wik14b].

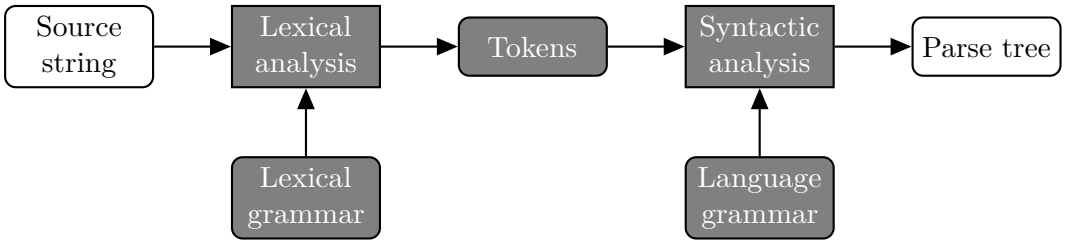


Figure 3.1. Typical flow of data in a parser (steps inside the parser are filled in grey. Rectangles stand for programs whose inputs and outputs are represented with rounded boxes).

3.1.1 Define grammars

The first stage is to defined grammars. There are two kinds of grammar to define: a lexical grammar for the lexer and a language grammar for the syntactic analysis. Let us see how to define them.

Lexical grammar

This grammar works with the lexer which takes a stream of characters and returns them as a list of tokens. Therefore, this grammar has to define to token which corresponds to the input character. For example, we can match the `+` operator to the token `PLUS`:

```
"+"      {return 'PLUS';}
```

Language grammar

This grammar defines a set of rules (*i.e.* a path to follow) to analyze the input stream according to the language.

In computer science, the most common type of grammar is the context-free grammar [Nel14, Mig14]. A rule is composed of two parts:

1. a name;
2. an expansion of the name.

When describing languages, we cannot miss the extended Backus-Naur Form (EBNF) which is a formal notation for describing grammars. The expansion can be a *terminal* symbol which is simply a token or a *non-terminal* symbols which is a sequence of tokens. Every rule in EBNF has the following structure:

```
<name> ::= expansion
```

Where `<name>` is a non-terminal symbol whereas `expansion` is a sequence of terminal and non-terminal symbols in a specific order. The expansion part can have different values which are separated by a vertical bar (`|`).

For example, we can define an expression to represent a sum:

```
<exp>  ::=  INTEGER '+' <exp>
        |   INTEGER
```

In the previous example, there are two terminal symbols (`INTEGER` and `'+'`).

Now that we can write a grammar, let us see how the first step of the parser works [Nor08].

3.1.2 Lexical analysis (tokenizing)

The lexical analysis (or lexer) breaks the input stream of characters into tokens. For example, if we have the following code as input stream:

```
local X in X = 10 end
```

Then it must produce the sequence of tokens:

```
local, X, in, X, =, 10, end
```

For example, a famous tokeniser is GNU Flex (Fast Lexical Analyser) [Pro08].

3.1.3 Syntactic analysis

Once tokens have been generated, the syntactic analysis checks if tokens form an allowable expression. This step works with reference to the language grammar defined earlier.

It produces a *parse tree* which is a representation of the structure of the source code (see Figure 3.2).

At this stage we can have a very common error due to the ambiguity of a grammar. A grammar is *ambiguous* if it describes at least one sentence for which there are more than one parse tree. This kind of error is particularly frequent in arithmetic expressions, let us see how the parse tree can be built for the sentence `id+id*id`:

With the following grammar:

```
E  ::=  E+E | E*E | (E) | id
```

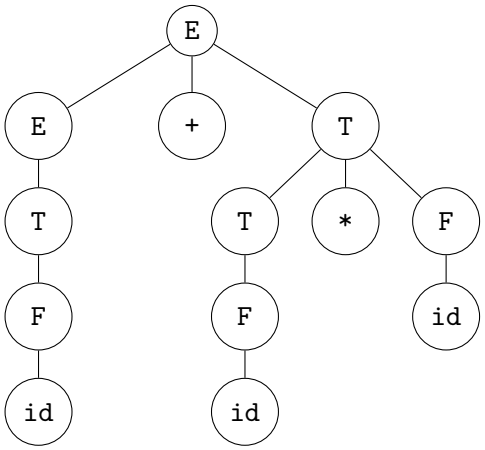


Figure 3.2. Example of parse tree for the input `id+id*id` [CIAD12].

The sentence can be parsed with a shift/reduce operation [1] (underlined token is the next one to be expanded):

- $\underline{\text{E}} \Rightarrow \underline{\text{E}}$ + E $\Rightarrow$ id + $\underline{\text{E}} \Rightarrow$ id + $\underline{\text{E}}$ * E $\Rightarrow$ id + id * $\underline{\text{E}} \Rightarrow$ id + id * id
- $\underline{\text{E}} \Rightarrow \underline{\text{E}}$ * E $\Rightarrow \underline{\text{E}}$ + E * E $\Rightarrow$ id + $\underline{\text{E}}$ * E $\Rightarrow$ id + id * $\underline{\text{E}} \Rightarrow$ id + id * id

We see that the sentence `id+id*id` with this grammar leads to two different expansions, *i.e.* two parse trees, which are depicted in Figure 3.3.

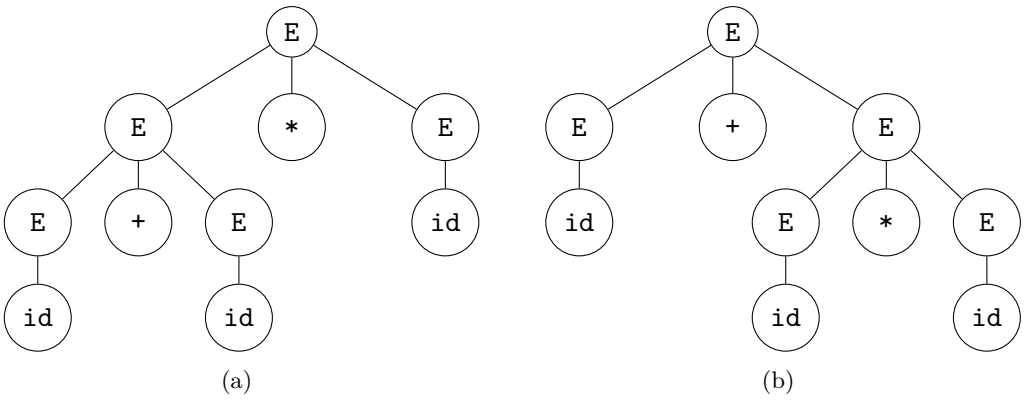


Figure 3.3. Examples of parse trees for the input `id+id*id` leading to an ambiguity [CIAD12].

[1] In this example, we use the leftmost derivation but it could have been also demonstrated with a rightmost derivation. A complete definition and examples can be found in [CIAD12].

3.2 Parser generators

A lot of tools can generate the source code of a parser, they are called *parser generators*.

In this section, we present some of them and make a comparison in order to select the most adequate one. All of them combine both lexical and syntactical analysis.

3.2.1 Bison

GNU Bison is a parser generator written in the 1990s by Robert Corbett and Richard Stallman [Lev09]. It converts a context-free grammar into a deterministic parser. Moreover, it can generate a parser in C, C++ and in Java but the latter is still experimental [FSF12a]. It can recognize languages described by LALR(1), LR(1), IELR(1), GLR grammars [Wik14a].

It works for a wide range of languages, from a simple calculator to complex programming languages.

3.2.2 Jison

Jison was written by Zach Carter in 2009 to help study for a compilers course [Car09]. It is essentially a clone of Bison but in JavaScript. It can recognize languages described by LALR(1), LR(0), SLR(1) grammars.

3.2.3 PEG.js

PEG.js was written in 2010 by David Majda and also generates a parser in JavaScript [Maj10]. The grammar used is slightly different from the one of Jison (and thus Bison) and the syntax is sensibly the same too. PEG.js uses an alternative to context-free grammars, *packrat* and supports any language defined by an LL(k) or LR(k) grammar [For14].

3.2.4 Comparison

As Glass Cat is a web platform, it needs to run in a web browser. Moreover, as we want the user not to wait too long for the result, Glass Cat has to be computed as fast as possible. As we supposed codes to be reasonably small, it does not require a specific power on servers. It can be parsed directly on the client side. Therefore, we have to compare the two parser generators which produce JavaScript.

We can compare Jison and PEG.js according to [Byn10] (see Tab. 3.1). Based on the number of features it has, the code size it produces and finally the speed.

	Features	**Code size**	**Speed [operations/s]**
Jison	More features (operator association, precedence)	± 12300 characters	972
PEG.js	Easier but less features	± 10000 characters	12.25

Table 3.1. Comparison of Jison and PEG.js parsing the same input. The speed test was run from [Byn10] and parses a 512 characters long text.

The test run on [Byn10] shows that Jison is 79 % quicker than PEG.js. Moreover, Jison has more features than PEG.js so it may be more suitable for a more complex language. We conclude that Jison is a better parser generator than PEG.js for Glass Cat.

3.3 Work with Jison

As explained earlier, Jison combines both lexical and syntactical analysis. Its lexical analyzer is modeled according to Flex and its parser is based on Bison. Someone who knows Bison, which is well documented, can easily switch to Jison.

Furthermore, Jison reports errors in the lexer and in the parser. Unfortunately, errors in the lexer are not very verbose and it can be smart to add some dummy regular expressions in order to process them in the parser and show a better error message [Pos12].

The lexical grammar and the language grammar must be in the same file and they are going to be compiled together with the command `jison my_grammar.jison`.

3.3.1 Lexical analysis

Let us start with the definition of the tokens for the lexical analysis:

```
1  /* lexical grammar */
2  %lex
3
4  %%
5  \s+                   /* skip whitespace */
6  [0-9]+("."[0-9]+)?\b  {return 'NUMBER';}
7  "*"                   {return '*';}
8  "-"                   {return '-';}
9  "+"                   {return '+';}
10 "^"                   {return '^';}
11 "$"                   {return 'DOLLAR';}
12 "?"                   {return 'QUESTION_MARK';}
13 "="                   {return 'OPERATOR_ASSIGNMENT';}
14 <<EOF>>               {return 'EOF';}
15
16 /lex
```

Listing 3.1. Lexical grammar with Jison.

Listing 3.1 shows a lexical grammar where lines 2 and 16 are the borders of the lexical grammar (`lex` stands for lexer). And its principle is relatively easy to understand:

- Write the token to parse at the beginning of a line between quotes.
- Specify which name corresponds to this token. It can be the same symbol as the token (like for `*` at line 7) or a new string (like for `$` at line 11).

3.3.2 Precedence

After the `/lex` of the lexical analysis, we can define precedence for some tokens (see Listing 3.2), *i.e.* the order in which the shift/reduce operation will be done. Ambiguity of Figure 3.3 can be resolved with this concept as the parser knows which parse tree is good.

```
1  /* operator associations and precedence */
2
3  %left '+' '-'
4  %left '*' '/'
5  %left '^'
6  %left UMINUS
```

Listing 3.2. Definition of operator associations and precedence.

3.3.3 Define a new rule

In order to define a new rule in the language grammar, we can follow the pattern defined in Listing 3.3.

```
1 name
2     : left TOKEN right
3        { /*do something*/ ; }
4     | left TOKEN_2 right
5        { /*do something 2*/ ; }
6     ;
```

Listing 3.3. Definition of a new rule.

This example begins with the name of the rule and two sentences which can perform an action once its children are parsed.

A more complex example is given in Listing 3.4 and defines a real grammar according to the tokens given in the previous subsection in Listing 3.1.

Line 1 says from which rule the parser must start to parse, *i.e.* which rule is the root of the parse tree. Furthermore, it requires some comments to understand what it really does [FSF12b]:

- `$$` returns the value of the left-hand side of the rule.
- `$`n returns the value of the n-th variable on the right-hand side of the rule.
- `yytext` is the actual token read so the line `Number(yytext)` gives a numerical representation of the token which has just been read.
- as the generated code is JavaScript, we can replace the `/*do something*/` part with a JavaScript code.

Moreover, as this stage builds a tree to know the structure of the sentence (like in Figure 3.2), it will perform a *depth-first* traversal of the tree, *i.e.* recursively visiting the subtrees of each node from left to right and then the node itself. The programmer must be aware that the first rule encountered has its `/*do something*/` performed at the end.

```
%start expressions

%% /* language grammar */

expressions
    : exp EOF
        {print($1);}
    ;

exp
    : exp '+' exp
        {$$ = $1+$3;}
    | exp '-' exp
        {$$ = $1-$3;}
    | exp '*' exp
        {$$ = $1*$3;}
    | exp '/' exp
        {$$ = $1/$3;}
    | exp '^' exp
        {$$ = Math.pow($1, $3);}
    | '-' exp %prec UMINUS
        {$$ = -$2;}
    | '(' exp ')'
        {$$ = $2;}
    | NUMBER
        {$$ = Number(yytext);}
    ;
```

Listing 3.4. Definition of rules.

Mid-rules

Sometimes we want to `/*do something*/` before the end of the complete sentence. This is called a *mid-rule* and is easily handled by Bison where it is written [FSF12a]:

```
name
    : left TOKEN  { /*do something 1*/ ; }  right
       { /*do something 2*/ ; }
    ;
```

Listing 3.5. Mid-rule in Bison. `do something 1` is performed once `TOKEN` is parsed.

But in Jison it is not yet defined and we must decompose the rule in order to do it [Car13]:

```
name
    : name2 right
       { /*do something 2*/ ; }
    ;

name2
    : left TOKEN
       { /*do something 1*/ ; }
    ;
```

Listing 3.6. Mid-rule in Jison. `do something 1` is performed once `TOKEN` is parsed.

Chapter 4

Visual Programming

"Draw me a sheep!"

— *A. de Saint Exupéry, The Little Prince*

All over the years, humans have always tried to represent the things they could not see in order to understand them. Moreover, as the adage says: a picture is worth a thousand words. To help students understanding what they cannot see, we are building a learning tool for them to represent the memory. This tool has to show what is happening during the execution of an Oz program. Hundhausen et al. said *visualization technology is not an instrument for transferring knowledge into students, but can serve as a catalyst for learning* [Hun02].

This chapter first defines the concept of software visualization. Then, we see which specifications this kind of tool must satisfy and more precisely those in the educational field. Afterwards, we look at some programs that show the execution of a code. For each of them, we spot the most innovative concepts they propose. As a conclusion, the design choices of GlassCat are unveiled.

4.1 Definition

The global denomination for software tools that can visualize programs is *software visualization*. In [Die07], Stephan Diehl defines software visualization as a tool that includes a wide range of techniques in order to develop and evaluate codes. This tool has to represent graphically the structure, the execution and the evolution of a program.

Therefore, we can be more specific and identify two broad subfields within software visualization [S+12]. On the one side, *algorithm visualization* has a high level of abstraction and is used to visualize general algorithms (*e.g.* quicksort,

binary tree operations). On the other side, *program visualization* has a lower level of abstraction and visualizes concrete, implemented programs.

In this master's thesis, we focus on program visualization tools. Algorithm visualization is too complex to be interesting for learning about a basic program execution like those a beginner wants to do with Oz.

4.2 A reference model for visualization

Card et al. propose to see visualization as a mapping from data to a visual form that the human perceives. Therefore, they made a reference model for visualization [MMC02]. Figure 4.1, adapted from [CMS99] shows the transformations applied to the flow of data.

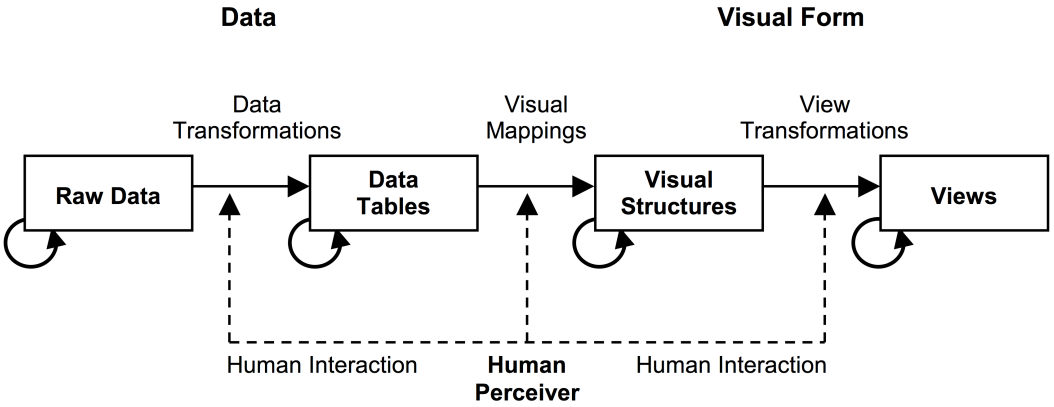


Figure 4.1. The reference model for visualization [MMC02] proposed by Card et al. Visualization converts data into a visual form. Of course, the output must make sense for a human. [CMS99]

The first transformation (Data Transformations) converts raw data into more usable data tables because the format of the raw data can be hard to work with. Data tables are an abstraction of the raw data, such as parse trees or dependence graphs, and can include some metadata to give more information about the characteristics.

The second transformation (Visual Mappings) is the core of the reference model as it has to take the mathematical relationships of the data tables and map them to a visual structure based on graphical properties.

Finally, the last one (View Transformations) creates items and displays datas in a user-friendly way.

We will establish the link between Glass Cat and this figure in the chapter about implementation.

4.3 Classification

Before starting to develop a new tool, it is mandatory to focus on some questions. Otherwise, the software will be too general and will not focus on a particular problem that will lead to a lack of functionalities. In their task-oriented classification [MMC02], Maletic et al. give five dimensions to guide the development of visualization softwares.

Here are their five dimensions:

1. Task – *why* do we need this tool? (*e.g.*, reverse engineering, defect location)
2. Audience – *who* will use it? (*e.g.*, expert developer, team manager)
3. Target – *what* do we need to represent? (*e.g.*, source code, execution data)
4. Representation – *how* can we represent it? (*e.g.*, 2D graphs, 3D objects)
5. Medium – *where* the visualization will be represented? (*e.g.*, onscreen graphics, virtual reality)

For this master's thesis, our specifications considering the classification of Maletic are:

1. Task – help the learning and teaching of the Oz kernel language.
2. Audience – Oz beginners who know the Oz kernel language.
3. Target – part of the Oz kernel language.
4. Representation – step by step execution.
5. Medium – onscreen graphics.

4.4 Visual representation in education

Students are sometimes afraid of delving into a new language or the first programming language they learn. Most of the time they are lost and have no idea of what kind of tool to use. Reading a tutorial is not sufficient to learn something (only 10 % is learned), but they will remember 30 % of what they see [SJR+13]. So we can wonder how visual representation can be helpful for education?

4.4.1 What to see?

In their classification, Kelleher and Pausch [KP05] identify three ways to help students to understand program execution.

1. Tracking: a system for tracking program execution that will show what happens in memory when a program is running.

2. Physical explanation: replace a general-purpose programming language by another one whose commands have a physical explanation in a virtual world.
3. Metaphors and graphics: describe every action in a programming language through metaphors and graphics.

According to these criteria, we see that this master's thesis brings a real help for education as it lies in the tracking criterion.

4.4.2 Improve the student's engagement

If students program by their own, they will remember 90 % of what they are doing [SJR+13]. Therefore, we can present the *engagement taxonomy* introduced by Naps et al. in [NRA+02] which describes increasing ways of engagement with a visual interaction.

Seven years later, Myller et al. improved this taxonomy with four additional levels. They were more focused on collaborative learning and they concluded that as the level of visualization gets higher, the engagement, the communication and collaboration between the users get higher too.

The *extended engagement taxonomy* as described by Myller et al. is summarized in Table 4.1. We have to note that the levels after viewing do not form a strict hierarchy but we can hypothesize that a higher level has a greater impact on learning than a lower one. Moreover, levels higher or equal to viewing implies to view the visualization. So for Naps et al., the combination of many levels leads to a better learning.

We want Glass Cat to gather four items from Table 4.1: (3) the user can control the visualization; (4) the learner can enter input; (7) the student can change a value and see the difference; (9) the learner can present and explain the visualization to others.

4.5 A brief presentation of some visual programs

For many years, a lot of visualization softwares have been developed with some good ideas. Let us have a look at some of them and which interesting concepts it brings.

4.5.1 JIVE

JIVE (see Figure 4.2) is an interactive execution environment for Eclipse [Uni11]. It can be used to debug (with visualizations of object structure and method interactions), to facilitate software maintenance (by providing insight into the dynamic behavior of programs) and it can also be used to teach and learn Java.

#	Level	Description
1	No viewing	There is no visualization tool at all.
2	Viewing	The learner simply views a visualization without interacting with it.
3	Controlled viewing	The learner can control the visualization, *e.g.*, by changing the animation speed or choosing which objects to examine.
4	Entering input	The learner enters input to the target software before or during execution.
5	Responding	The learner answers questions about the target software.
6	Changing	The learner changes a visualization while viewing, *e.g.*, via direct manipulation of the visualization components.
7	Modifying	The learner modifies a visualization before viewing, *e.g.*, by changing the target software or an input set.
8	Constructing	The learner constructs a visualization interactively from components such as text and geometric shapes.
9	Presenting	The learner explains a visualization to others.
10	Reviewing	The learner views the visualization in order to provide feedback to others about the visualization or the target software.

Table 4.1. The *extended engagement taxonomy* (based on Myller, Naps et al. [MMSBA04, NRA+02]).

A very nice thing with JIVE is that it uses a sequence diagram to display method calls per object/class and thread.

4.5.2 Memview

The Memview debugger (see Figure 4.3) can automate the creation of memory diagrams [GMT+05]. Memview focuses on three key concepts: (1) the memory address (2) the reference created in memory (3) the difference between the stack, the heap and static space.

Memview brings a new idea to show the stack, the static space and the heap of Java programs.

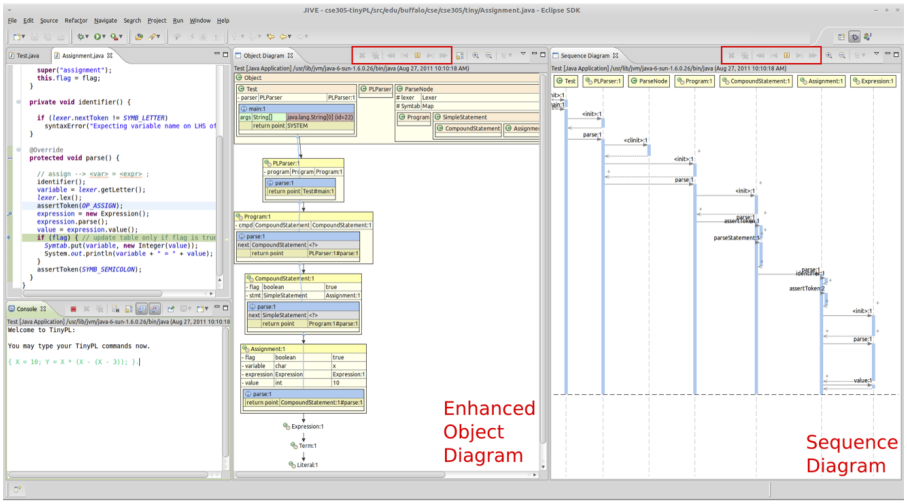


Figure 4.2. JIVE represents an event sequence as a sequence diagram and each execution state as an object diagram [Uni11].

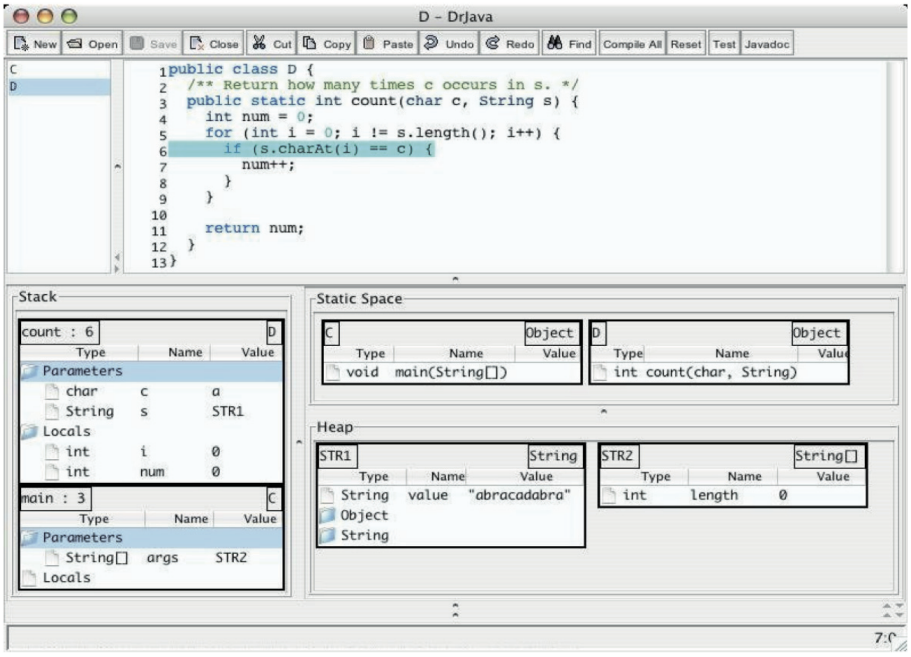


Figure 4.3. Memview shows the stack, the heap and static space [GMT+05].

4.5.3 PlanAni

PlanAni (see Figure 4.4) executes Pascal programs and uses images as metaphors [SK]. For example, a fixed-value is represented by a carving stone, changing variables inside loop are represented by footprints with the new value written at each step.

With PlanAni, it is easier for beginners to visualize the steps of a program but it is more adequate for younger people than for university students.

4.5.4 CSmart

CSmart (see Figure 4.5) takes another approach than what we have already seen [GBWS11]. Before students start coding in C, the teacher has to type the code he expects and annotate it with comments. Then, CSmart will give annotations to the students at each step [GBWS11].

CSmart proposes a good way to learn coding but it is mandatory for the teacher to define its own exercises.

4.5.5 ViLLE

ViLLE (see Figure 4.6) is an online learning platform [RLKS08]. It is used to show the dynamic behaviour of program execution. Moreover, ViLLE supports Java, C++, pseudo-code and it can also handle new languages.

A new concept here is that students have to answer to questions while executing the code, which makes them engaged more deeply in the learning process.

4.5.6 Python online tutor

Online Python Tutor (see Figure 4.7) is a web-based program visualization tool for Python [Guo13]. With this tool, a student can type his own code and see how the computer executes it, line by line. Moreover, it shows the stack frames, variables and heap object contents.

Online Python Tutor has a nice design and furthermore, a student can generate an URL of the current visualization at an exact execution point and send it to someone.

4.5.7 Jeliot 3

Jeliot is one of the longest-lasting and most-studied program visualization tools for CS1 [MMSBA04, S+12, Hel09]. It was introduced as Eliot in 1997. Then, Jeliot I, Jeliot 2000 and Jeliot 3 came. The two first versions shared the same goal: to ease the production of algorithms animations. Jeliot I can be used on the Internet

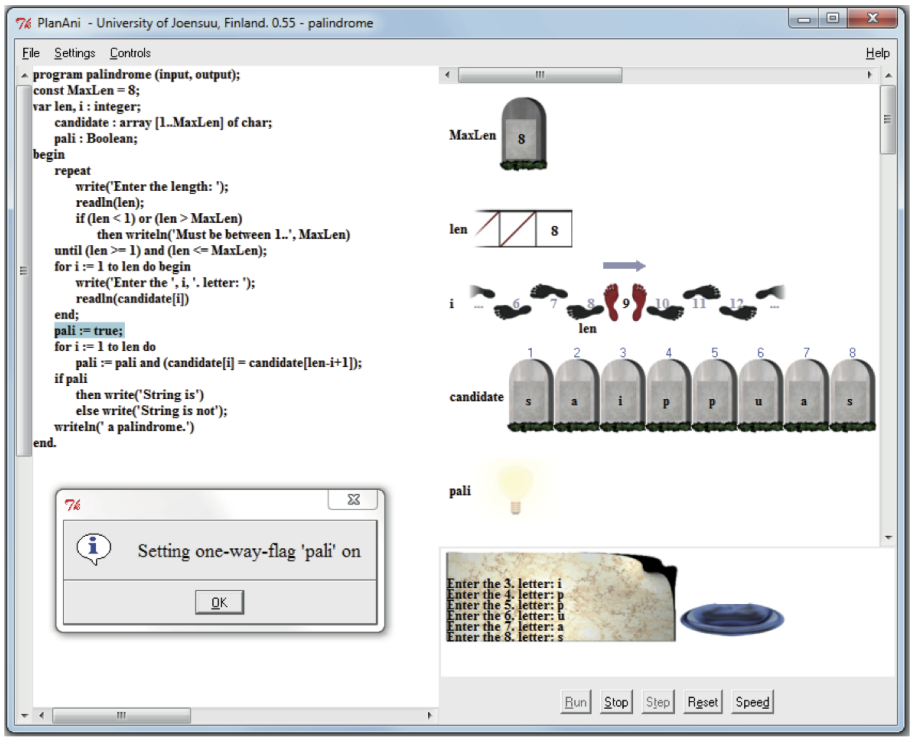


Figure 4.4. PlanAni executes Pascal programs. Here it is the visualization through an array [SK].

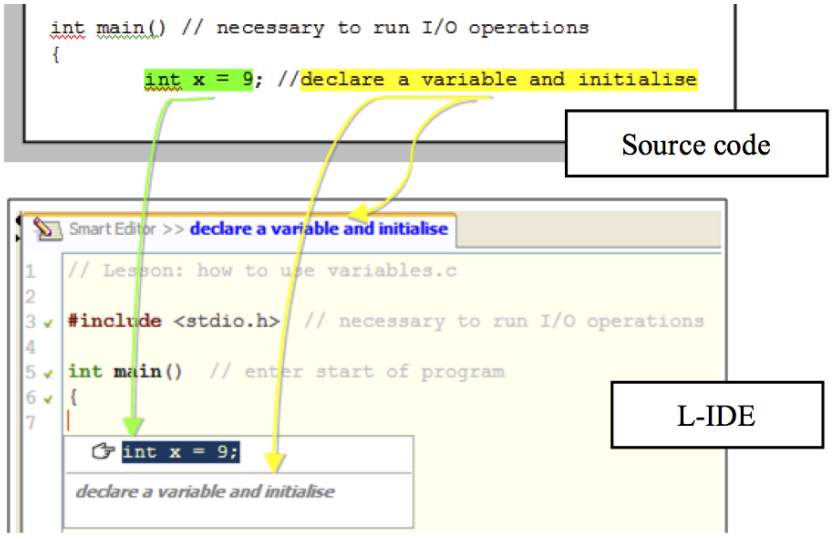


Figure 4.5. CSmart extracts comments from the teacher's source code and gives them as a help in the student's code: the Learner's Integrated Development Environment (L-IDE) [GBWS11].

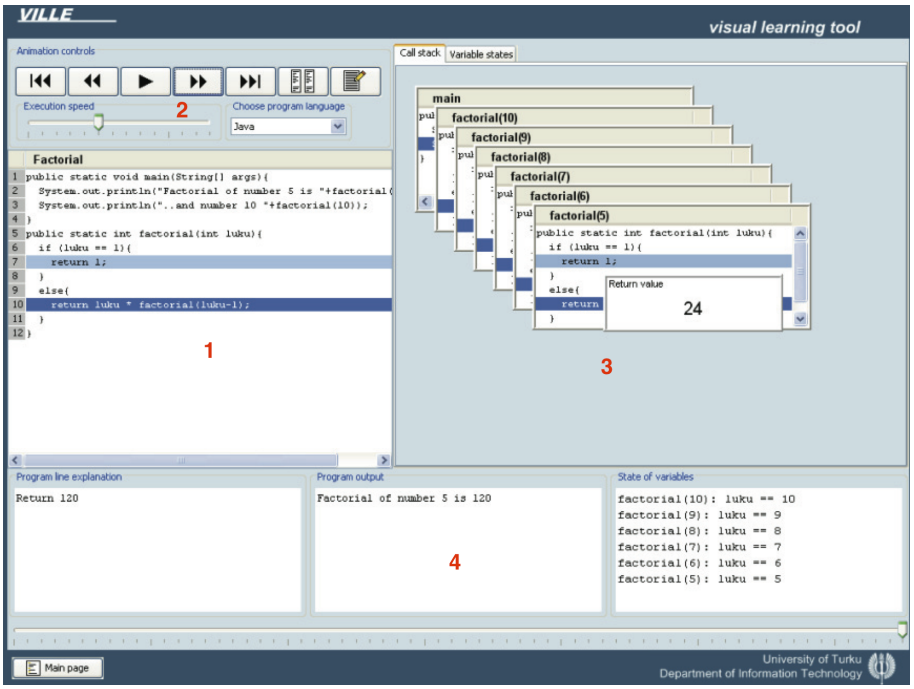


Figure 4.6. The visualization view of ViLLE consists of three areas. (1) contains the program code (2) is the program controls (3) displays the call stack (4) gives the program outputs and variable states [RLKS08].

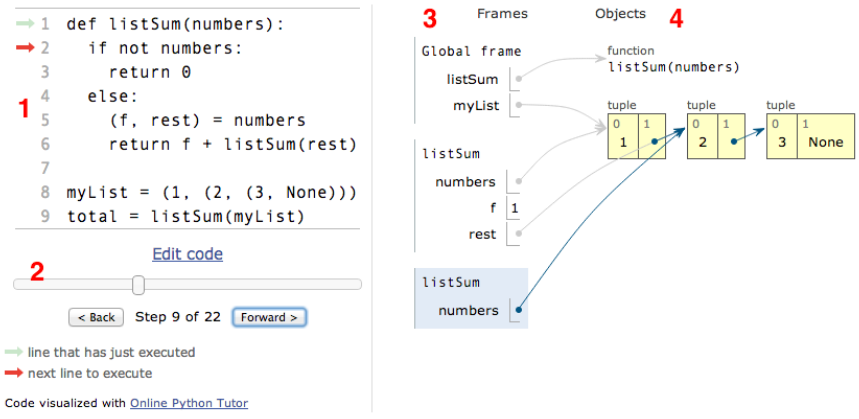


Figure 4.7. Online Python Tutor displays (1) the line of code currently executed (2) a way to control the state of the execution (3) stack frames and variables (4) heap object contents and pointers [Guo13].

while Jeliot 2000 was designed for novice learners. In order to make it as modular as possible, they separated the interpreter from the visualization, so they use an intermediate language between them. This language consists of simple ASCII text lines that carry all the information needed to visualize the interpretation of a program.

Jeliot has a control panel with *VCR-like* buttons to control the visualization frames. When an error is raised, it highlights the code where the error happened.

4.5.8 Summary

In this section, we have seen some visual programs. We started with JIVE which has the advantage to display the method calls so that students can follow a timeline of the execution of their programs. Then a nice thing with Memview is that it shows the stack, the static space and the heap of Java programs. With this tool, students can have a look at what is happenning in the memory. Afterwards, we saw PlanAni which uses a metaphoric way to process. But this concept is only interesting when working with young students. CSmart is a good idea but not very attractive as it only consists of writing the comments from the teacher's code in the student's code. ViLLE has the same advantage as Memview but supports many programming languages. Moreover, Python online tutor can generate a sharable URL of the current visualization at an exact execution point. Finally, Jeliot 3 can highlight the erroneous line in the code.

4.6 Design Choices

We defined Glass Cat according to each theoretically concepts of visual programming presented in Section 4.4. Furthermore, we saw a lot of inspiring ideas in the previous section. It is now time to unveil the design of Glass Cat.

4.6.1 Goals

The most valuable goal we want to achieve in the design part, is to produce the best user-friendly website. To do that, we can define three sub-goals:

Goal 1: Minimal number of buttons as the user is lost when there are a lot of choices.

Goal 2: Easy way for the user to see what he wants.

Goal 3: Scalability in order to deal with long results.

We explain in the following subsections how we achieve those goals. The first subsection treats the two firsts goals and the scalability subsection talks about the third goal.

First, we have to decide what will be shown to the user . . . semantics of course! But there are a lot of things to show so we have to decide what is the most relevant part. It turns out that the state (*i.e.* the code that must still be parsed) and the store (*i.e.* variables in the memory) are the most useful part of semantics to display. The environment is not displayed as it only shows the link between variables and their value in the memory, which can be deducted according to variables name.

4.6.2 Interaction

The main objective of the design is to make it easy to use without the need of reading a manual. Moreover, it must be beautiful otherwise people do not want to use it and they are bored working with it.

After many ideas and reflections, Glass Cat was born. The website contains five areas (see Figure 4.9):

1. A menu: access code examples, information about the author and Glass Cat.
2. A textarea: auto-indent the code and colorize keywords, strings, comments.
3. A program control: used to manage steps of the semantics (previous, next and execute to start parsing, GO TO step is a plus as the user can directly go faster to a specific step).
4. State: displays the state from right to left.
5. Stores (single-assignment and cell stores): display the store (up is the single-assignment one, down is the cell store).

Another interaction that must be handled is when an error occurs while parsing. The user must be warned in an understandable way. Errors appear in a popup in front of the window, while the background is dimmed. Some errors are explained in a human-language with the exact explanation of what is going wrong (see Figure 4.10 (a)). Others are basic errors returned by the parser (see Figure 4.10 (b)) and those are displayed without any treatment.

4.6.3 Scalability

Since the student's code can be very long, Glass Cat must be able to handle huge store and state. Therefore, boxes are scrollable in both ways. Moreover, the store is a stack so every new element is added at the top.

In the implementation part, we talk a little more about the scalability in order to avoid infinite loops.

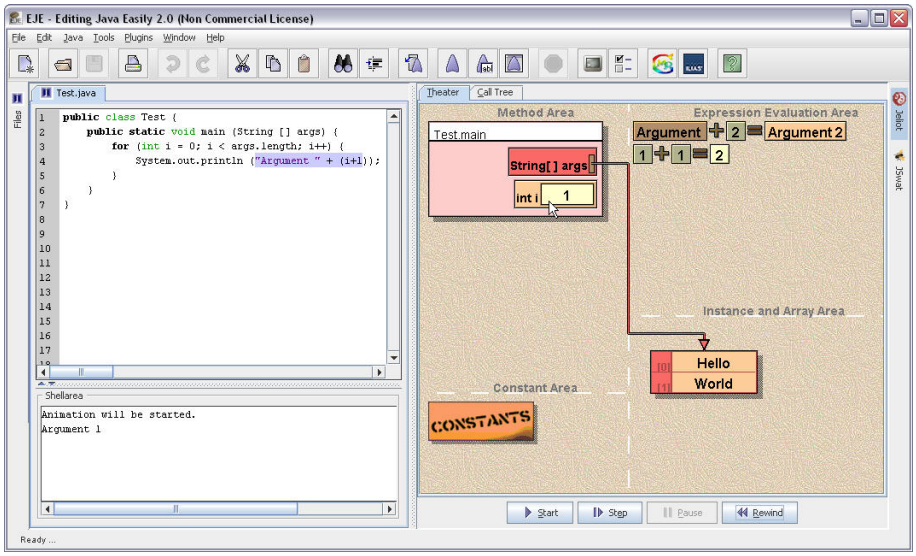


Figure 4.8. Jeliot 3 shows the execution of Java programs step by step.

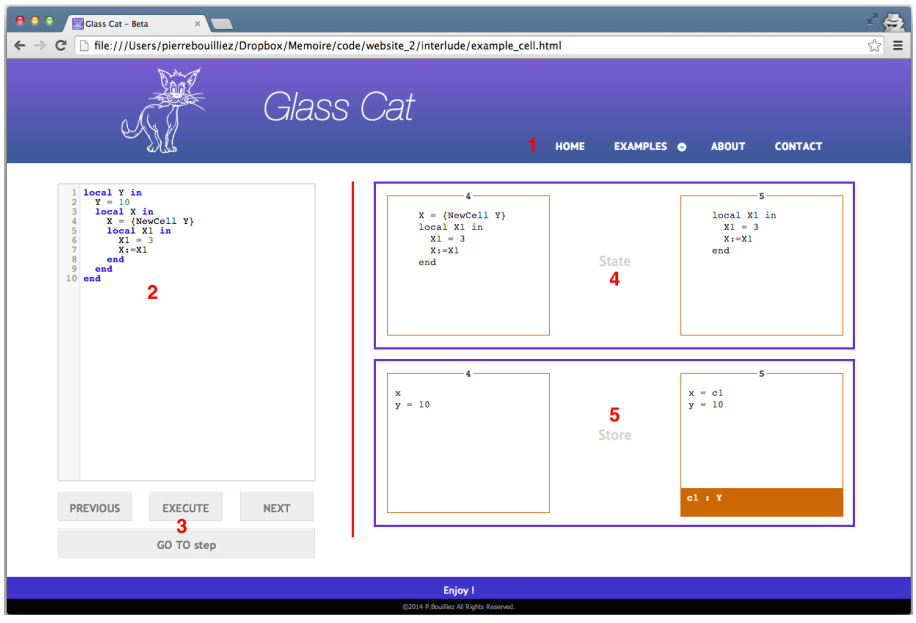
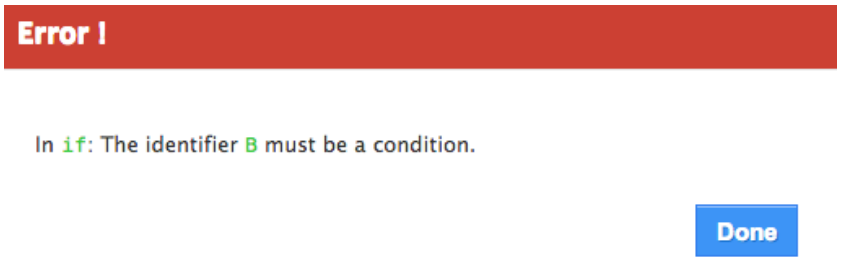
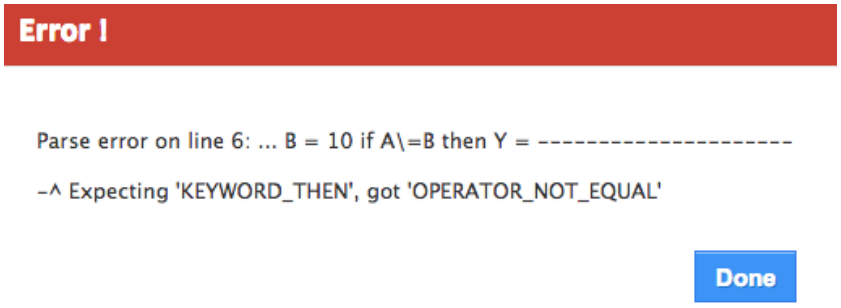


Figure 4.9. Design of Glass Cat.



(a) Smart



(b) Dumb

Figure 4.10. Example of errors shown by the error manager in Glass Cat.

Chapter 5

Pythia

> "Do not train a child to learn by force or harshness; but direct them to it by what amuses their minds, so that you may be better able to discover with accuracy the peculiar bent of the genius of each."
>
> — *Plato*

Nowadays, the Internet is accessible to a large number of people. Some of them are interested in programming but they may be stuck because they do not know how to code. Maybe they want to learn a new language but have no idea about how to learn it. These people can use Pythia, a web-platform to learn how to program. Students can register on the website to get exercises with smart feedbacks. Glass Cat has the same goal than Pythia, offering to the students a way to learn programming. Moreover, as unity makes strength, the work of this master's thesis can be integrated inside Pythia.

This chapter briefly explains how Pythia works and how Glass Cat could be integrated into it.

5.1 The Pyhia platform

On the Pythia website, students choose a module. Then they have some exercises to do. Figure 5.1 shows the first one for Oz. There are multiple areas to focus on:

1. The first box gives the context, *i.e.* the aim of the exercise and functions that can be useful to solve it.
2. A texteara to code in.

3. Once the student clicked on the button to submit his/her work, this box displays the output of the code with a smart feedback.

4. A green frame means that the result is what it is expected. On the contrary, a red frame is not a good one.

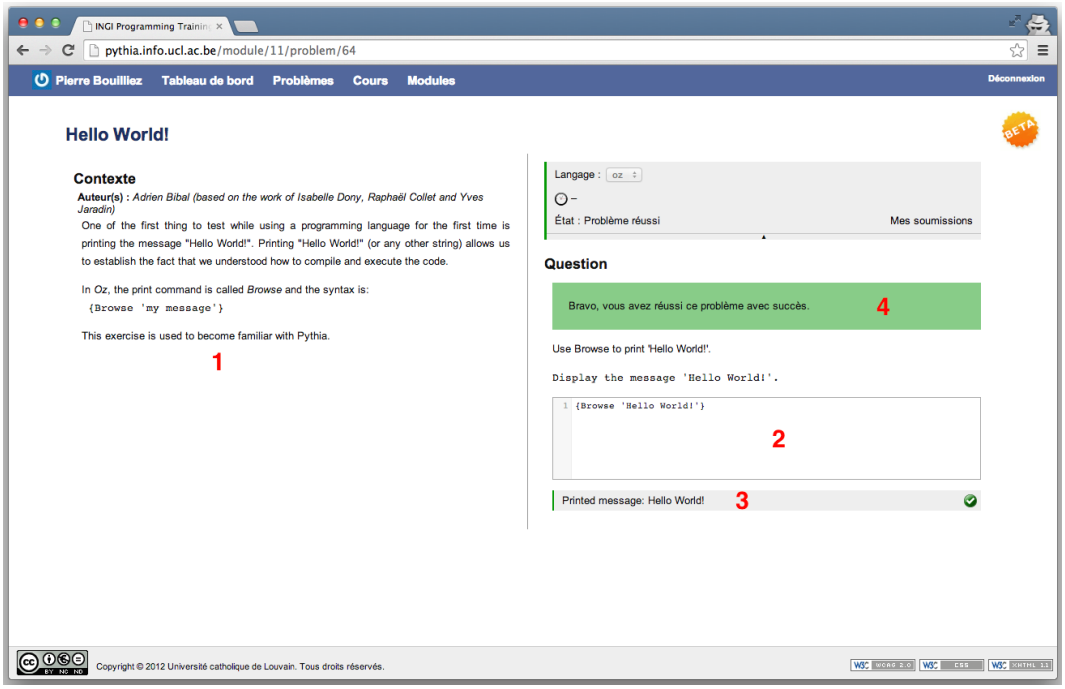


Figure 5.1. The first exercise for Oz on Pythia.

5.2 Pythia and the edX platform

Since January 2014, the Oz course taught by Prof. Peter Van Roy is accessible on edX, a platform for MOOCs. As a programming course, assessing the students must be performed by checking their ability to produce code [CBR14]. Therefore, the Pythia platform was integrated into edX to grade programs while providing relevant feedbacks.

This course sees a part of LFSAB1402 taught at the Université catholique de Louvain and also contains the semantics. On the MOOC, students start by watching videos where the professor introduces the theory for the week. Then, they have *classical* and *coding* exercises to do. Classical exercises expect a word or a sentence as answer while coding exercises are based on Pythia.

5.3 Integration of Glass Cat

Figure 5.2 shows a text area on the edX platform where the student can code and get the feedback from Pythia. We can integrate Glass Cat at this stage. Learners can code in this textarea and check the semantics of their code directly on Glass Cat simply by pressing the button "Run on Glass Cat". Once this button is pressed, a new tab with Glass Cat is opened and the input is already filled with the code from the textarea (see Figure 5.3).

Figure 5.2. Example of textarea on Pythia/edX. When the student has written a code, it can checks its semantics on Glass Cat.

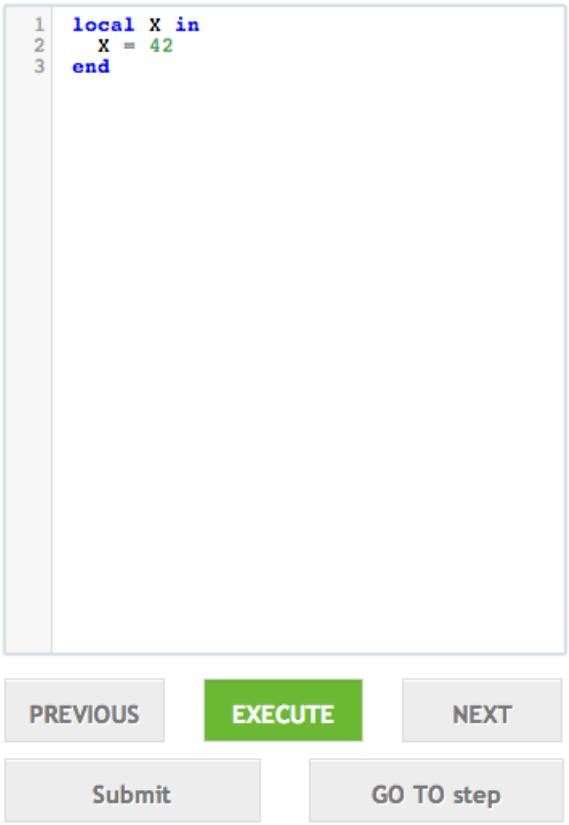


Figure 5.3. Glass Cat is filled with the student's code and she/he can submit it back to Pythia/edX by pressing the "Submit" button.

Chapter 6

Implementation

"Description begins in the writer's imagination,
but should finish in the reader's."

— *Stephen King, On Writing: A Memoir of the Craft.*

In the previous chapters, we decided how Glass Cat should look like. We also found a parser generator to make it alive. Now, it is time to explain the structure of the program, to see the steps to go from a grammar to a website that displays the semantics of a code in Oz.

This chapter is divided according to the structure of Glass Cat, *i.e.* the interpreter and the visualisation part. We explain how they interact together and give a global description of the code.

6.1 Structure of the program

Along this master's thesis, we have defined Glass Cat according to two distinct parts (see Figure 6.1) (1) a back-end which is the heart of the program and analyzes the code the user provided and (2) a front-end that implements the visual part.

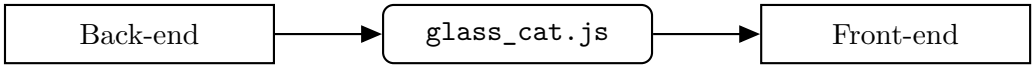


Figure 6.1. The structure of Glass Cat can be divided into a back-end which generates a JavaScript file used by the front-end to display semantics. Files are represented in rounded boxes and compilations or calls to external programs are symbolized by rectangles.

6.2 Back-end

The back-end generates a JavaScript file that parses a code written in the kernel language of Oz. The structure of this part is sketched in Figure 6.2. It takes two input files (`oz.jison` and `ast_nodes.js`) and outputs one single file `glass_cat.js`. The upper path generates the parser `oz.js` with Jison, from a grammar. Afterwards, this file is concatenated with `ast_nodes.js` for performance reasons [Bak12].

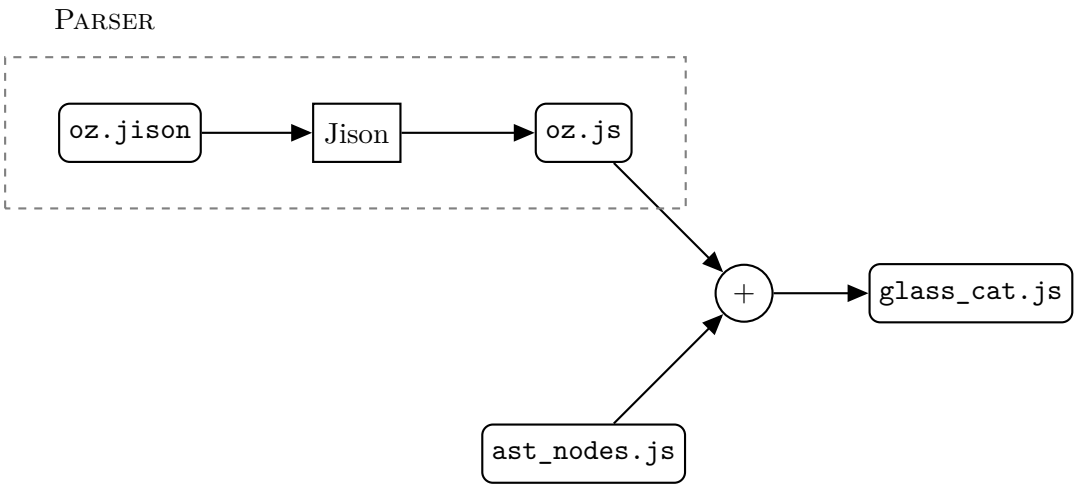


Figure 6.2. Structure of the back-end of Glass Cat. Files are represented in rounded boxes and compilations or calls to external programs are symbolized inside rectangles. The + operator represents a concatenation.

In this part, we start with an explanation of the parser which converts the code into a syntax tree. Then we look at the objects that are created by this parser and how the interactions can be done between the nodes of the tree.

6.2.1 Parser

Inside the file `oz.jison`, we must define a grammar following an EBNF notation as explained in Chapter 3. An example of a rule is shown at Listing 6.1 but a complete description can be found in Appendix A. In this subsection, we explain some points we must be aware of when we use the parser `oz.js`.

Jison receives the grammar and generates a scanner and a parser inside the same output file `oz.js`. But, before the end of the compilation, it can raise an error if there is an *ambiguity* in the grammar. If a sentence can lead to more than one parse tree, it gives the paths in the trees which generated a failure.

When `oz.js` is used, the *scanner* tokenizes the code according to the lexical grammar. During this step, we must pay attention to some points:

- It can remove *comments* from the input according to a regex.
- A *bad match*. When the scanner reads a new character, it checks sequentially if it is in the grammar we provided. E.g consider a grammar that contains first a token for `:` and then another one for `:=`. If the input stream has a `:=`, the scanner reads the first character of this string and finds a match for `:` in its grammar. Instead, if we define a token for `:=` before the one for `:`. Then when the scanner sees `:` in the input stream, it thinks that it is the beginning of `:=` and checks if the next character is `=`. Longest match must be at the beginning of the grammar and the more inaccurate ones at the end of the grammar.
- *Errors* are raised when there is no match. Unfortunately, there is no error when the input is assigned to a wrong token.

Next step, the *parser* has to find a sentence that matches the input according to the tokens returned by the scanner. If it does not succeed, it throws a syntax error with the line where the error happened. As we are working on a educational tool, a better description of an error can help students to understand why their code is wrong. Therefore, we can add some dummy regular expressions in the grammar to let wrong inputs to match. Then, the next step has to verify if this input is correct or not and throw an error with an exact definition of the failure.

6.2.2 Parse tree nodes

Once the parser has treated the input, a parse tree is generated. This tree contains objects that are created by the parser every time a sentence is matched. These objects are defined in the file `ast_nodes.js`.

This part is the heart of Glass Cat because it manages every operation done on sentences. Moreover, it implements the concepts of Oz defined in Chapter 2.

Let us have a look back at the language grammar used by the parser. For example, a part of the variable declaration from the grammar [1] is shown in Listing 6.1.

We can observe the tokens for the `local` ⟨x⟩ `in` at line 2. When the parser sees this kind of sentence, it executes the children of this node, then it creates a new `VariableDeclarator` object with the name of the identifier ⟨x⟩ as argument (the other argument is explained with procedures).

[1] The complete definition of the rule contains many sub-rules but this short example gives a first approach of how it works.

```
variable_decl
   : KEYWORD_LOCAL variable_creation_id KEYWORD_IN@
     { $$ = new VariableDeclarator($2, cur_decl);}
   ;
```

Listing 6.1. Variable declaration in the grammar with the creation of a new object.

As explained in Chapter 3, `$$` means the returned value. This implies that the parent node can access this object.

Inside the class `VariableDeclarator`, a few things must be done:

- Define `this` variables with the type of the node and its value. These variables are used to know the kind of node we are dealing with. This is very useful to build the semantics.
- Check if there is already another identifier with the same name in the store. Consequently, we count the number of declarations i for each identifier $\langle x \rangle$ in a global variable. If this number is equal to 0, then this is the first time that an identifier has this name. Otherwise, the identifier must be renamed $\langle x \rangle\_(i+1)$.
- Create a new variable $\langle x \rangle$ in the store. We represent the store as a dictionary in a global variable. When a new entry is added, we put the identifier as a key of this dictionary with an empty value.

We see that this implementation is really similar to what we defined in Section 2.2 with the semantics statements of Oz. The novelty is that we check the name of the identifier and rename it.

We can wonder why the variable declaration is not defined as a whole sentence. In fact, as we have seen earlier, the parse tree is traversed with a depth-first search algorithm. Therefore, the tree of Figure 6.3 executes first the code from the node `variable creation id`, then `variable decl` and so on. If there were no mid-rules, the statement $\langle s \rangle$ where the identifier is declared would have been evaluated first... even before the creation of the identifier.

Moreover, with mid-rules, we can define the end of the scope of an identifier at the `variable creation` node as its right child is the terminal token `END`. We perform this operation by deleting the identifier from the store [2]. Doing this way, we know if an identifier is declared or not at the current line.

6.2.3 Semantics

Now that we can define a new rule, we must store semantics of each statement in order to display them. We have to save the store and the state to be evaluated after the current statement.

[2] The store here does not correspond to the semantic store that is displayed.

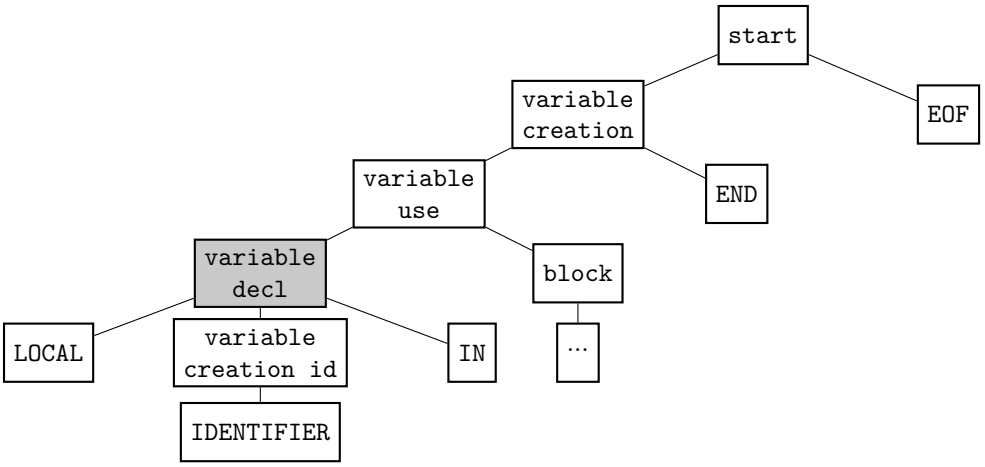


Figure 6.3. Abstract syntax tree (AST) from the starting point of the grammar down to the creation of a variable. The tree is traversed in the depth first search order [Lev09]. The grey node corresponds to the place where the object representing the variable declaration is created.

We implemented a function `add_semantics` for each object. This one takes three arguments (1) the state that follows (2) the value to add in the store (3) the type of this statement. The object stores its semantics in a variable. Moreover, while this object goes up the tree, its semantics is merged with the one of the parent object and so on up to the root.

As a new call to the function `add_semantics` defines a new step to browse, it is mandatory to know where we must add this new semantics. Therefore, let us have a look at Figure 6.3 to find a node which knows the informations we need. If we continue with the variable declaration, the state is the statement inside `in` ⟨s⟩ `end` which corresponds to the node `block` in the tree. And the new entry in the store is the identifier ⟨x⟩. The most adequate node is `variable use` as it sees the next statement [3] and it can get the object returned by the `variable decl` node (and so access the name of the identifier).

But there is a bottleneck, as the `block` is evaluated before `variable use`, their semantics is evaluated in the same order. Whereas, the semantics of the variable creation must be displayed first. Therefore, we must re-order them before returning the object.

[3] Jison can give the location references of a token inside an input with `@`n where n is the location of the n-th variable on the right-hand side of the rule.

Cells

For the cells, we define another store like the mutable store defined in Section 2.2. When there is a `Y = {NewCell X}` we must create a new cell `c`i in this store where i is an incremental number according to the number of cells. Therefore, we have a global variable which keeps track of every cell created. And when the value of the cell changes, the mutable store knows if the cell has to point to another identifier.

6.2.4 Procedures

When there are no procedures, we can execute the parse tree instantaneously. As the parser reads the input, it creates all the nodes and they can build the output directly.

On the other hand, when we are working with procedures, things are different. We cannot process the parse tree for the content of the procedure anymore, as the values of the parameters are not yet known. As the parser reads sequentially the input, it cannot skip the procedure and parse it when there is a call. Therefore, we must keep track of every operation that is happening inside the procedure in order to replay it.

While parsing, every new object created in the parse tree adds a string to a global variable with all the informations necessary to replay it (statements and a reference to their locations). As the parser is the only entity that can see the structure of a program and thus add the semantics, we must add them to the global string too. In fact, statement locations are added to a global dictionary while parsing. When objects are created, we give them a key (`cur_decl`) in the dictionary. When the objects are replayed, the same value is passed in order to add the right semantics.

Once the procedure is completed, the value of the string is added to the store as the value of the identifier of the procedure.

Moreover, procedures are also called *closures* because they capture, *i.e.* close the environment when the procedure is defined. At this exact time, we add the value of the contextual environment to the store.

Once the procedure is called, the contextual environment is restored, identifiers are bound to the parameters values and the procedure trace is replayed. This algorithm also works for recursive calls.

Recursive calls might be infinite and can make the page crash. Consequently, we must ensure that an infinite loop cannot occur. Therefore, we check that the number of recursion does not reach a critical threshold.

6.3 Front-end

The front-end of Glass Cat takes four items[4] (see Figure 6.4). This part requires JavaScript/jQuery, HTML and CSS. Starting from `glass_cat.js` and rotating clock-wise, we have (1) the output of the back-end (2) `anim.js` which is the JavaScript for the animations on the website (for the states/stores and to start the parser) (3) HTML/CSS codes to display the content of the website (4) Code Mirror is a JavaScript editor that colorizes and indents codes dynamically [Hav14].

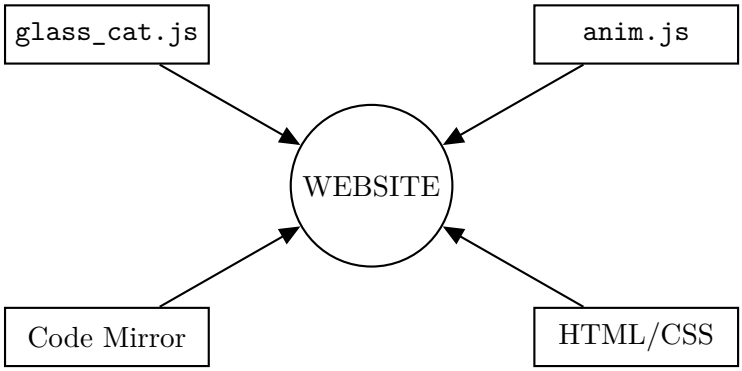


Figure 6.4. Structure of the front-end of Glass Cat. Files are represented in rectangle boxes and compilation or call to external programs are symbolized by circles.

As a reminder, Figure 6.5 shows the four buttons to control Glass Cat.

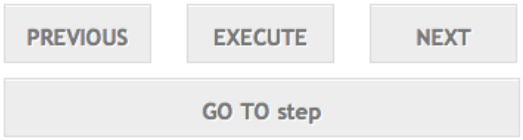


Figure 6.5. The four buttons to control Glass Cat.

Moreover, a more expressive graph is designed in Figure 6.6 to show interactions between all these files.

Once the user pressed the EXECUTE button, `anim.js` asks `glass_cat.js` to parse the input codes. At the end of this step, states are already computed waiting to be displayed.

When the NEXT button is pressed, the stack is computed in `glass_cat.js`, *i.e.* the store of each semantic statement is merged from the current point. At this stage, there are a few items that need a specific treatment, *e.g.* the initialization of a variable must erase the declaration of this variable and if it is a cell, the two stores must change.

[4]Here we call it items as some of them are composed of many files (like Code Mirror and HTML/CSS). Therefore, they are not written with a typewriter font.

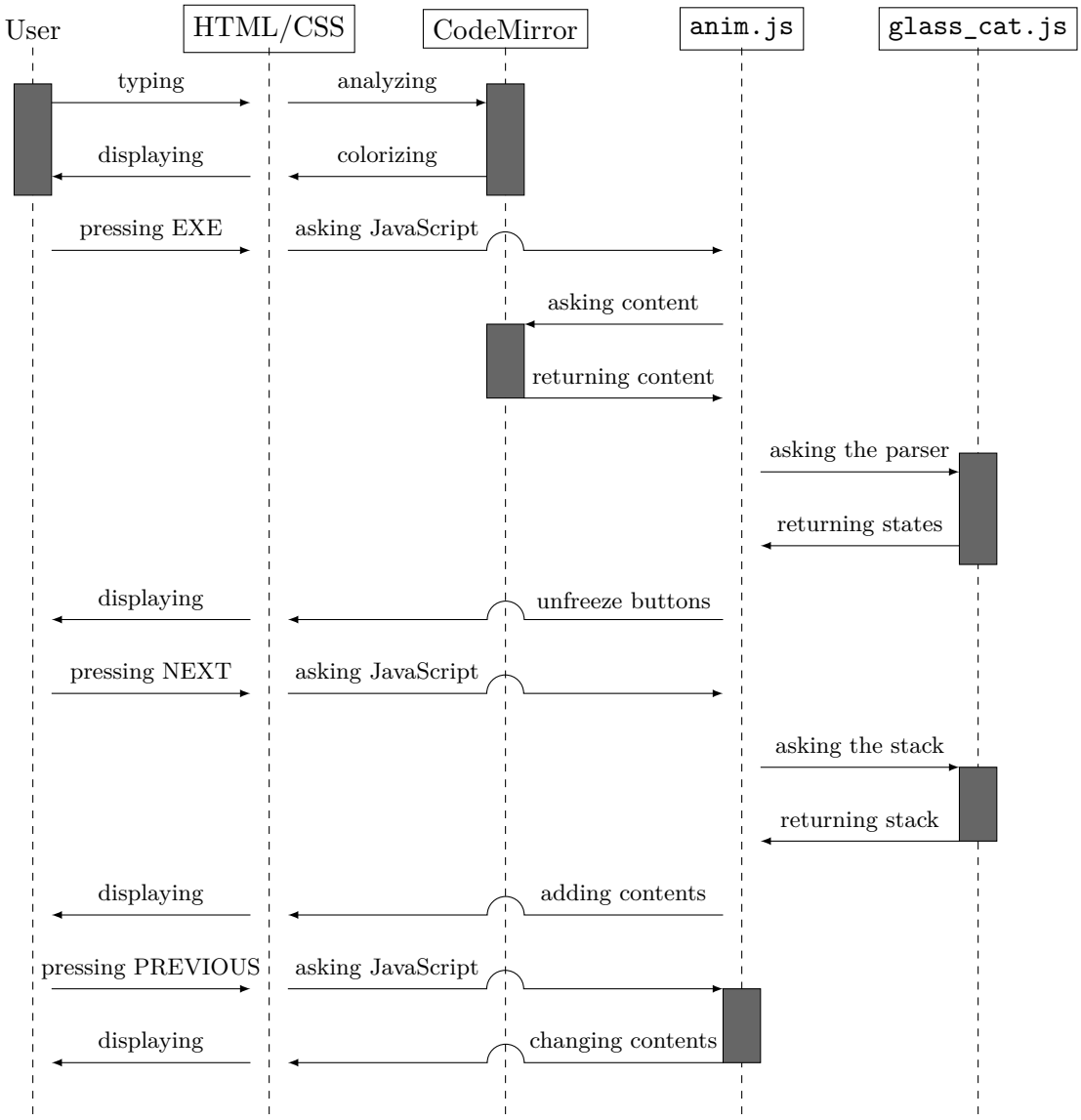


Figure 6.6. Sequence diagram displays a scenario from top to bottom, where a user type some code and then the EXE button, the NEXT one and then PREVIOUS. Black boxes represents computations steps. This figure shows, in a different way, how Glass Cat can be transposed to the reference model for visualization (Figure 4.1).

On the other hand, pressing the PREVIOUS button does not require a call to `glass_cat.js`, everything is done in `anim.js`. It calls a jQuery function to animate the box and hide the old one.

Chapter 7

Evaluation

> "Our greatest glory is not in never falling, but in rising every time we fall."
>
> — *Confucius*

We implemented this thesis following the agile methodology. First, we spent a lot of time looking for the best design, the most relevant part of the semantics to show. Once we knew where we wanted to go, a minimal version of Glass Cat was developed, *i.e.* limited to the creation of a variable and check if the semantics is right. The next steps were to extend this basic version with a new concept of Oz. Then we must check if the added concept works and if the old ones still work.

In this chapter, we present which subset of Oz is implemented and what kinds of operations the users can do. Then, we evaluate the correctness of Glass Cat and how we proceed to test it. Finally, we have a brief talk about how we can make it even faster.

7.1 Subset of Oz

Re-implementing every concepts supported by Mozart in JavaScript might have been a gargantuan work for a master's thesis. Therefore it was decided to make Glass Cat work only with the kernel language of Oz and especially the one seen in the second-year bachelor in Civil Engineering at the Université catholique de Louvain.

Table 7.1 shows the general kernel language of Oz with a check mark on the right of each rule that works in Glass Cat. Half of the kernel language has been implemented. However, every rule seen in the course LFSAB1402 except the thread creation can be parsed.

⟨s⟩	::=	`skip`	Empty statement	✓
	|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence	✓
	|	`local` ⟨x⟩ `in` ⟨s⟩ `end`	Variable creation	✓
	|	⟨x⟩$_1$ `=` ⟨x⟩$_2$	Variable-variable binding	✓
	|	⟨x⟩ `=` ⟨v⟩$_2$	Value creation	✓
	|	`if` ⟨x⟩ then ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Conditional	✓
	|	`case` ⟨x⟩ `of` ⟨pattern⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`	Pattern matching	✓
	|	`{`⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$`}`	Procedure application	✓
	|	`thread` ⟨s⟩ `end`	Thread creation	
	|	`{ByNeed` ⟨x⟩ ⟨y⟩`}`	Trigger creation	
	|	`{NewName` ⟨x⟩`}`	Name creation	
	|	⟨y⟩`=!!`⟨x⟩	Read-only view	
	|	`try` ⟨s⟩$_1$ `catch` ⟨x⟩ `then` ⟨s⟩$_2$ `end`	Exception context	
	|	`raise` ⟨x⟩ `end`	Raise exception	
	|	`{FailedValue` ⟨x⟩ ⟨y⟩`}`	Failed value	
	|	`{NewCell` ⟨x⟩ ⟨y⟩`}`	Cell creation	✓
	|	`{Exchange` ⟨x⟩ ⟨y⟩ ⟨z⟩`}`	Cell exchange	
	|	`{IsDet` ⟨x⟩ ⟨y⟩`}`	Boundness test	

Table 7.1. The general kernel language of Oz (from [VRH04]) with a check mark next to the statements that Glass Cat can parse.

7.2 Correctness

In order to verify that Glass Cat works well, we performed a lot of tests. Those tests are based on the concept of functional correctness. For a specific input, we first check visually that the output is correct. If it is good, we take the value of the store and add it to the database for the tests. Once the test is run, it compares the good value with the returned value.

The tester is presented on a web page where a user can add tests with their expected result. Then all tests can be run individually or together. Figure 7.1 shows test 60 which contains a procedure, a recursive procedure call and a condition.

We tried to make Glass Cat as modular as possible, *i.e.* when adding a new rule, the rest of the system does not have to be changed. With this kind of test we can ensure that it is still working on old tests even with a new version of the interpreter.

A total of 62 tests have been implemented, but as there can be an infinity of paths to tests, it can still be improved. Nevertheless, it covers each statement of the kernel language defined in the previous section at least once.

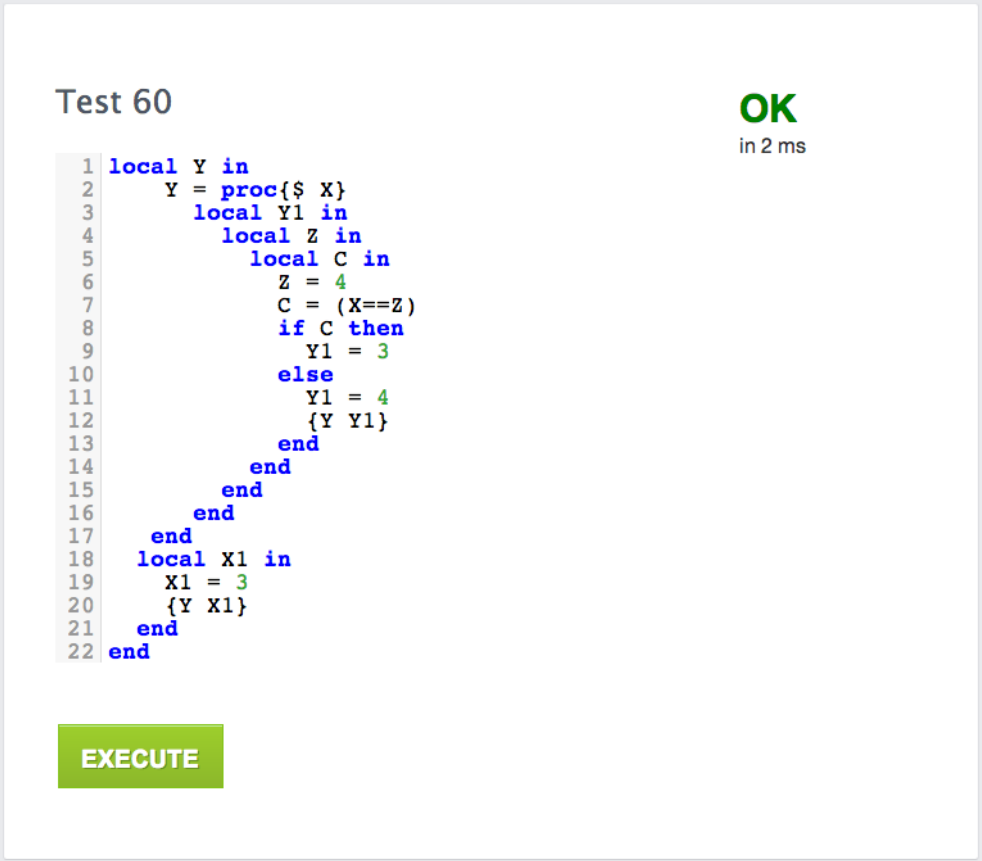


Figure 7.1. A test inside the web page for the tests. Test 60 contains a procedure, a recursive procedure call and a condition. Once the execute button is pressed, the tester compares the right value of the store with the one produced with the actual version of Glass Cat. If it matches, OK is displayed otherwise KO. Here the test did well in 2 ms.

7.3 Execution time

The first objective of a master's thesis is to deliver a working tool before optimizing it. Meanwhile, if the fans of the user's computer start to blow while the parser works, this is not a program that will be used.

We ran test 60 (Figure 7.1) with Google Chrome version 34 and it took 2 ms to parse it and it used 4.8 MB of JavaScript heap. Moreover, we tested Glass Cat on Chrome 34, Firefox 29 and Safari 7.

Figure 7.3 shows a very strong positive correlation (correlation coefficient according to the Pearson's correlation equal to 0.77) between the number of characters of the input and the time taken to parse it.

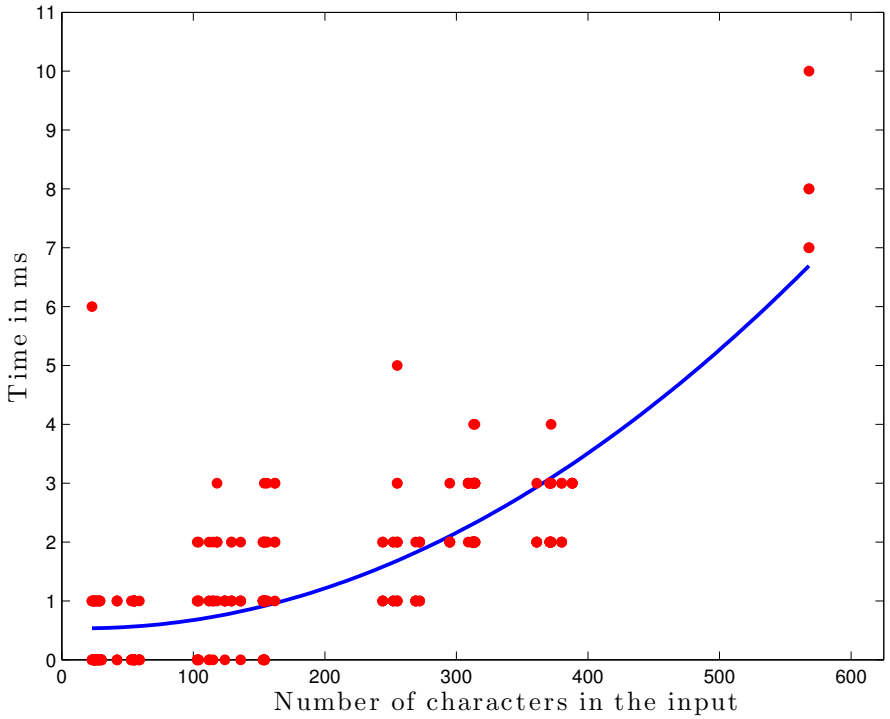


Figure 7.2. Time to parse the code as a function of the number of characters in the input. This graph is generated from the website to test Glass Cat without recursive tests. Codes do not process the same task. All the tests were run 5 times and are displayed as red dots. The blue line is a second-order interpolation.

There are some outliers in this graph (1) values of the first column because they are the first to be generated [1], we can see that the time decreases at each test for this first column, (2) between 200 and 300 characters there is a higher value at test 30 but there is nothing particular at this time, a condition and a `if`-statement (maybe the garbage collector cleaned the previous objects) (3) the highest values are naturally the ones that correspond to test 42 with the longest code.

Figure 7.3 shows the time taken for the computation of the factorial, starting with the factorial of 1 up to the factorial of 5. There is a huge bottleneck with recursive calls as the time fits a fourth-order interpolation relative to the number of procedure calls. It comes from the evaluation of the procedure calls which evaluates two times one procedure call (first to create the semantics then to add it to the node), so the last call is evaluated in $\Theta(N!)$ with N the number of recursive calls.

[1] See next paragraph to understand why.

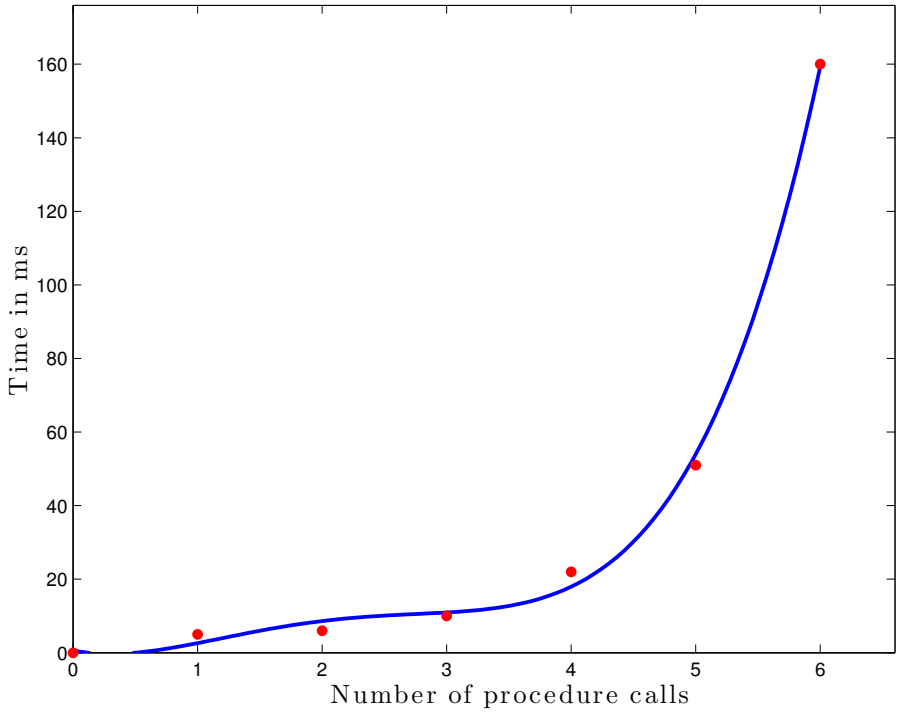


Figure 7.3. Time to parse the code as a function of the number of calls and recursive calls in the procedure `Fact` going from factorial of 1 up to factorial of 5. All the tests are displayed as red dots. The blue line is a fourth-order interpolation.

7.4 Optimizations

But as the time taken to evaluate a code increases almost linearly with the number of characters and quadratically with recursive calls, we might want to optimise the code and make it run faster. In [Bak12], we can find some good practices in JavaScript like avoiding string concatenation and initializing instance variables and class methods with the `prototype` keyword. This keyword avoids unnecessary initialization each time the constructor is called and it can create only a single function for an object, how many instances there are.

Undoubtedly, the first thing to do in order to optimize performances of a JavaScript code is to remove every print in the console because it costs a lot. Moreover, some tools can be used to make your JavaScript codes better like Google Closure [Dev14] which:

- Removes white space : reducing the size in order to load faster.
- Changes variable names to shorter ones : reducing the size.
- Checks code : warning for illegal JavaScript operations.

When the back-end generates `glass_cat.js`, it also produces an optimized version with Google Closure. Table 7.2 shows the differences between the two outputs. We see that the optimized version has better performances than the non-optimized version. By reducing the number of lines and characters, the size is also reduced. On the small code, it takes roughly the same time but on the long code, the optimized version is 15 % faster than the non-optimized version.

Unfortunately, the optimized version does not resolve the bottleneck of recursive calls except that it goes a little bit faster, the complexity does not change.

	Number of lines	**Number of characters**	**Size (KB)**	**Execution time (small code)**	**Execution time (long code)**
Normal	3041	101627	102	± 5 ms	± 160 ms
Optimized	149	68150	68	± 4 ms	± 140 ms

Table 7.2. Comparison of the normal version of Glass Cat and the optimized version. The small code corresponds to test 60 (338 lines with one recursive call) and the long code is the computation of the factorial of 5 (470 lines with 5 recursive calls).

Chapter 8

Conclusions

"Everything has to come to an end, sometime."

— *L. Frank Baum, The Marvelous Land of Oz*

While the Internet has been growing, education was confined to paper-and-pencil. But now, education is changing and students can take university courses from their home. Therefore, developing tools to learn via the Internet is really an actual topic.

However, how hard can it be to make a tool for education? First, it must be attractive to involve students in the exercise. Nonetheless, this is nothing without a background, a program that computes a function according to the input given by the user. Present master's thesis tried to answer that question in a structured way for a tool that shows the semantics of Oz.

The initial step presented some concepts of the programming language. Especially, we looked at the kernel language up to the abstract machine to express the formal semantics.

Then, we focused on the back-end, looking for a tool that can understand and interpret the code the user just wrote. It was required to explain briefly the theory of parsers. This lead us throughout the definition of grammars, lexical and syntactic analysis. Therefore, we used a parser generator to create an interpreter for Glass Cat with a grammar based on the kernel language of Oz. This is definitely the most complex part of this thesis as it must be modelled on an existing language while respecting its properties.

Later, we talked about the design. Based on papers and researches, we tried to understand how to make the best visual programming tool in order to kindle in student's mind a desire to learn. We have been led to compare some visual programming tools principally ones that are used with an educational aim. Thanks to them, we built the design of Glass Cat. Taking care of being user-friendly and scalable.

Since Glass Cat is an educational tool, the idea was to integrate it to an existing platform. There is one at the INGI department, Pythia. Consequently, a succinct explanation of this platform adds a more general context to this thesis.

After all, in the implementation part we explained the global structure of the code and how Glass Cat came to life with the interpreter acting like a brain whose face is the visualization part. We travelled through the code from the definition of the grammar and its transformation into a parser creating objects and managing errors. Then, we took a look at the human-machine interaction and the path of the request inside the program.

Finally, we evaluated Glass Cat based on the objectives defined in the introduction. First, the global kernel language of Oz was reminded with check marks on statements that were implemented. Second, we tested Glass based on functional correctness with numerous input codes. Third, we checked its speed and the memory used, making it as light as possible.

8.1 Perspectives and limitations

Finally, we are aware that Glass Cat is not perfect, it could have been more structured with a distinct split between the back-end and the front-end. Furthermore, nodes could have been more generic, *i.e.* exactly the same processing for each node and not some exceptions.

We can also point out that the recursion is not optimal and the number of calls to replay a procedure increases in $\Theta(N!)$ with N the number of real calls to the procedure in the student's code.

Nevertheless, Glass Cat has already asked a lot of work but it can be extended with some other tools. I tried to make a good documentation to make it survive the final presentation of this master's thesis. Above all, feedbacks can give other ideas and improve it for understanding.

Extend the grammar The most important thing is to extend the grammar of Glass Cat with most features from Oz, starting with threads.

Highlight the current state Instead of displaying a box for a state, we could highlight the current state and the line popped from the stack. Making more space for the store.

Errors Errors are raised by the parser, but Glass Cat could be extended in order to act like a debugger, showing stores and states up to the error or a break point.

Display the scope of variables We can add many things to the website, *e.g.* a area where we can check scopes of each variables.

A more descriptive semantics As we keep track of every type of statements, we could imagine an area that displays the name of every semantics operation that is performed.

Sharable URL If a student is stuck at a certain step, Glass Cat could generate a short URL. This one re-opens Glass Cat with the same input code at the right semantics step.

8.2 Open questions

During this master's thesis a lot of choices were made. Therefore, we can wonder how Glass Cat might have looked like if we took another path.

Glass Cat in a virtual machine Instead of re-implementing Oz in JavaScript, we could have loaded a virtual machine on the server side with a customised version of Mozart which can build semantics and give it back to the front-end. Is such a choice better than the current version of Glass Cat?

Change the design In Chapter 4 we saw many tools that displays the memory. Can Glass Cat be improved if we do not display semantics as defined steps but only with a memory represented with boxes and references that evolves dynamically?

Appendices

Appendix A

Developers

This appendix describes the code of Glass Cat. If you want to extend it with a new rule or a new functionality, this chapter is for you.

Glass Cat is available at http://pbouilliez.github.io/glass-cat/, then you choose the developer version. You have access to the code and to the tests.

A.1 Installation of Jison

To generate a parser, install Jison for Node using `npm` [Car09].

I did it on a computer which runs on Linux Mint 15 Olivia (GNU/Linux 3.8.0-19-generic x86_64) with the following steps:

1. `sudo apt-get install python-software-properties`
2. `sudo apt-get update`
3. `sudo apt-get install nodejs`
4. `curl https://npmjs.org/install.sh | sudo sh`
5. `npm -v` (to check if it everything is ok)
6. `npm install jison -g`
7. `git clone git://github.com/zaach/jison.git`

A.2 Grammar

The complete grammar is defined in Listing A.1.

```
1
2  /* description: Parses end executes mathematical expressions. */
3
4  /* lexical grammar */
5  %lex
6
7  %s                    comment
8
9  %%
10
11 "%".*                 /* skip comments */
12 \s+                   /* skip whitespace */
13
14 "~"                   {return '~';}
15
16 "{"                   {return 'EMBRACE';}
17 "}"                   {return 'UNBRACE';}
18
19 "true"                {return 'TRUE_LITERAL';}
20 "false"               {return 'FALSE_LITERAL';}
21 "NewCell"            {return 'NEWCELL_LITERAL';}
22
23 "=<"                  {return 'OPERATOR_LESS_THAN_EQUAL';}
24 "<"                   {return 'OPERATOR_LESS_THAN';}
25 "=="                  {return 'OPERATOR_EQUAL';}
26 ">="                  {return 'OPERATOR_GREATER_THAN_EQUAL';}
27 ">"                   {return 'OPERATOR_GREATER_THAN';}
28 "\="                  {return 'OPERATOR_NOT_EQUAL';}
29
30 "*"                   {return '*';}
31 "div"                 {return "div";}
32 "mod"                 {return "mod";}
33 "-"                   {return '-';}
34 "+"                   {return '+';}
35 "/"                   {return '/';}
36 "("                   {return '(';}
37 ")"                   {return ')';}
38 "$"                   {return 'DOLLAR';}
39 "?"                   {return 'QUESTION_MARK';}
40 "="                   {return 'OPERATOR_ASSIGNMENT';}
41 ":="                  {return 'OPERATOR_CELL_ASSIGNMENT';}
42 ":"                   {return 'COLON';}
43 "@"                   {return 'OPERATOR_CELL_GET';}
44 "."                   {return 'OPERATOR_RECORD_GET';}
45
46 "skip"                {return 'KEYWORD_SKIP';}
47 "nil"                 {return 'KEYWORD_NIL';}
48 "local"               {return 'KEYWORD_LOCAL';}
49 "in"                  {return 'KEYWORD_IN';}
50 "end"                 {return 'KEYWORD_END';}
51 "if"                  {return 'KEYWORD_IF';}
52 "then"                {return 'KEYWORD_THEN';}
53 "else"                {return 'KEYWORD_ELSE';}
```

```
54  "case"                  {return 'KEYWORD_CASE';}
55  "of"                    {return 'KEYWORD_OF';}
56  "proc"                  {return 'KEYWORD_PROC';}
57  "[]"                    {return 'OPERATOR_BRACE';}
58
59
60  [0-9]+["."]{1}[0-9]+    {return 'FLOAT';}
61  [0-9]+([0-9]+)?\b       {return 'INTEGER';}
62  [A-Z][a-zA-Z0-9_]*      {return 'IDENTIFIER';}
63  "\"\""                  {return 'STRING_LITERAL';}
64  "\""([^"]|{BSL})*"\""   {return 'STRING_LITERAL';}
65  ['])*[a-z0-9\W][a-zA-Z0-9]*[']*   {return 'LABEL';}
66
67
68  <<EOF>>                 {return 'EOF';}
69
70  /lex
71
72  /* operator associations and precedence */
73
74  %left '+' '-'
75  %left '*' "div" '/'
76  %right "mod"
77
78  %start instruction
79
80  %% /* language grammar */
81
82  instruction
83     : EOF
84         { throw new Error("Did you forget to write something?"); }
85     | variable_creation EOF
86          { var cu = new CompilationUnit(); $1.add_semantics(@1,
           "\{\}", 100); $1.generate_semantics(); cu.close(); return cu
           ; }
87     | KEYWORD_SKIP EOF
88         { var cu = new CompilationUnit(); var v = new
           VariableDeclarator('E'); v.add_semantics('', '', 99); v.
           add_semantics(@1, "\{\}", 100); v.generate_semantics(); cu.
           close(); return cu;}
89     ;
90
91  block
92     :
93       { $$=new Block(); }
94     | variable_creation block
95       { $$=new Block();
96           if(!$2.isEmpty()) {
97              $$.add_semantics(@1, '', 4);
98              $2.add_semantics(@2, '', 5);
99           }
100          $$.eval_semantics($1.get_semantics(), $2.get_semantics());
101      }
```

```
102     | KEYWORD_SKIP block
103       { $$=new Block();
104           do_skip();
105           if(!$2.isEmpty()) {
106              $$.add_semantics(@1, '', 4);
107              $2.add_semantics(@2, '', 5);
108           }
109           $$.eval_semantics([], $2.get_semantics());
110       }
111     | if_then_statement block
112       { $$=new Block();
113           if(!$2.isEmpty()) {
114              $$.add_semantics(@1, '', 4);
115              $2.add_semantics(@2, '', 5);
116           }
117           $$.eval_semantics($1.get_semantics(), $2.get_semantics());
118       }
119     | if_then_else_statement block
120       { $$=new Block();
121           if(!$2.isEmpty()) {
122              $$.add_semantics(@1, '', 4);
123              $2.add_semantics(@2, '', 5);
124           }
125           $$.eval_semantics($1.get_semantics(), $2.get_semantics());
126       }
127     | smth block
128       { $$=new Block();
129           if($2.isEmpty()) {
130              $1.add_semantics('', $1.get_value(), 2);
131           }
132           else {
133              $1.add_semantics(@2, $1.get_value(), 2);
134           }
135           $$.eval_semantics($1.get_semantics(), $2.get_semantics());
136       }
137     | case_of_statement block
138       { $$=new Block();
139           if(!$2.isEmpty()) {
140              $$.add_semantics(@1, '', 4);
141              $2.add_semantics(@2, '', 5);
142           }
143           $$.eval_semantics($1.get_semantics(), $2.get_semantics());
144       }
145     | procedure_call block
146       { $$=new Block();
147           if(!$2.isEmpty()) {
148              $$.add_semantics(@1, '', 4);
149              $2.add_semantics(@2, '', 5);
150           }
151           $$.eval_semantics($1.get_semantics(), $2.get_semantics());
152       }
153     ;
154
```

```
155 smth
156    : variable_initializer
157      { $$ = $1; }
158    | cell_initializer
159      { $$ = $1; }
160    | cell_getter
161    ;
162
163 /*** VARIABLE CREATION ***/
164
165 variable_creation
166    : variable_use KEYWORD_END
167      { $$=$1; $1.endScope(); }
168    ;
169
170 variable_use
171    : variable_decl block
172       { $$ = $1; $1.add_semantics(@2, $1.getIdentifier(), 1);  $1.
          eval_semantics($1.get_semantics(), $2.get_semantics());  }
173    ;
174
175 variable_decl
176    : KEYWORD_LOCAL variable_creation_id KEYWORD_IN
177      { $$ = new VariableDeclarator($2, cur_decl); }
178    ;
179
180 variable_creation_id
181    : IDENTIFIER
182        { $$ = $1; }
183    ;
184
185 /*** VARIABLE INITIALIZATION ***/
186
187 variable_initializer
188    : variable_creation_id OPERATOR_ASSIGNMENT variable_assign
189        { $$ = new VariableInitializer($1, $3, cur_decl); }
190    ;
191
192 variable_assign
193    : expression
194    ;
195
196 /*** CELL INITIALIZATION ***/
197
198 cell_initializer
199    : IDENTIFIER OPERATOR_CELL_ASSIGNMENT cell_assign
200        { $$ = new CellInitializer($1, $3, cur_decl); }
201    ;
202
203 cell_assign
204    : expression
205    ;
206
```

```
207 cell_getter
208    : OPERATOR_CELL_GET IDENTIFIER
209       { $$ = get_cell_value($2); }
210    ;
211
212 /*** IF THEN ***/
213
214 if_then_statement
215    :  if_use KEYWORD_END
216      { $$ = $1; $1.endIf(); }
217    ;
218
219 if_use
220    : if_cond block
221      { $$ = $1; $1.set_block(@2, 1);  if($1.getCondition()) { $1.
       add_semantics(@2, '', 4); $1.eval_semantics($1.get_semantics(),
        $2.get_semantics());} }
222    ;
223
224 if_cond
225    : KEYWORD_IF IDENTIFIER KEYWORD_THEN
226      { $$ = new IfThen(new Identifier($2), cur_decl); }
227    ;
228
229 /*** IF THEN ELSE ***/
230
231 if_then_else_statement
232    :  if_use_else KEYWORD_END
233      { $1.endIf(); }
234    ;
235
236 if_use_else
237    : if_use KEYWORD_ELSE block
238      { $$ = $1; $1.set_block(@3, 2); if(! $1.getCondition()) {
       cur_condition[cur_id-1]=true; $1.add_semantics(@3, '', 4); $1.
       eval_semantics($1.get_semantics(), $3.get_semantics());} }
239    ;
240
241
242 /*** CASE OF ***/
243
244 case_of_statement
245    : case_use KEYWORD_ELSE block KEYWORD_END
246     { $$ = $1;  $1.set_block(@3, 2); if(!$1.is_it_good()){$1.
       add_semantics(@3, '', 5);  $1.eval_semantics($1.get_semantics(),
        $3.get_semantics());} $1.endIf(); }
247    ;
248
249 case_use
250    : case_cond block
251      { $$ = $1; $1.set_block(@2, 1); if($1.is_it_good()){$1.
       add_semantics(@2, $1.getIdentifier(), 5);  $1.eval_semantics($1
       .get_semantics(), $2.get_semantics());} $$.endScope(); }
```

```
252     ;
253
254 case_cond
255     : case_eval KEYWORD_THEN
256     ;
257
258 case_eval
259     : KEYWORD_CASE IDENTIFIER KEYWORD_OF pattern
260       { $$ = new Case($2, $4, cur_decl); }
261     ;
262
263 pattern
264     : IDENTIFIER
265       { $$ = new Pattern_Identifier($1); }
266     | LABEL '(' records_list ')'
267        { $$ = new Record($1, $3); }
268     ;
269
270
271 /*** PROCEDURE ***/
272
273 formal_parameter_list
274     : formal_parameter
275       { $$ = [$1]; cur_ce = get_ce(); cur_CE=jQuery.extend(true, {},
        identifiers_list); cur_proc = ""; }
276     | formal_parameter_list formal_parameter
277       { $$ = $1; $$.push($2); cur_ce = get_ce(); cur_CE=jQuery.extend
        (true, {}, identifiers_list); cur_proc = ""; }
278     ;
279
280 formal_parameter
281     : IDENTIFIER
282       { $$ = new Identifier($1); }
283     | QUESTION_MARK IDENTIFIER
284       { $$ = new ReturnIdentifier($2); }
285     ;
286
287 formal_parameter_list_call
288     : formal_parameter_call
289       { $$ = [$1]; }
290     | formal_parameter_list_call formal_parameter_call
291       { $$ = $1; $$.push($2); }
292     ;
293
294 formal_parameter_call
295     : IDENTIFIER
296       { $$ = new Identifier($1); }
297     ;
298
299 procedure_call
300     : EMBRACE IDENTIFIER formal_parameter_list_call UNBRACE
301       { $$ = new ProcedureCall($2, $3, @1, @4, cur_decl); }
302     ;
```

```
303
304 /*** EXPRESSION ***/
305
306 expression_case
307    : IDENTIFIER
308       { $$ = get_identifier($1); }
309    ;
310
311 records_list
312    : records_entry
313      { $$ = [$1]; }
314    | records_list records_entry
315      { $$ = $1; $$.push($2); }
316    ;
317
318 records_entry
319    : LABEL COLON IDENTIFIER
320      { $$ = new Record_entry($1, new Identifier($3)); }
321    | INTEGER COLON IDENTIFIER
322      { $$ = new Record_entry($1, new Identifier($3)); }
323    ;
324
325 expression
326    : expression1
327       { $$ = new Expression($1); }
328    | '(' expression ')'
329         {$$ = new Parenthesis($2);}
330    | KEYWORD_PROC EMBRACE DOLLAR formal_parameter_list UNBRACE block
       KEYWORD_END
331       { $$ = new Procedure($4, @1, @7, @6); $$.eval_semantics([],$6
          .get_semantics()); }
332    | IDENTIFIER
333       { $$ = new Identifier($1); }
334    | EMBRACE NEWCELL_LITERAL IDENTIFIER UNBRACE
335       { $$ = new Cell($3); }
336    | TRUE_LITERAL
337       { $$ = new Parenthesis(new Expression(new Condition(new
          Expression(4),new Expression(4),'==')); }
338    | FALSE_LITERAL
339       { $$ = new Parenthesis(new Expression(new Condition(new
          Expression(4),new Expression(4),'!=')); }
340    | LABEL '(' records_list ')'
341       { $$ = new Record($1, $3); }
342    | KEYWORD_NIL
343       { $$ = new Keyword_nil($1);}
344    | STRING_LITERAL
345       { $$ = new String_literal($1); }
346    ;
347
348 expression1
349    : expression '+' expression
350         { $$ = new MathExp($1,$3,$2); }
351     | expression '-' expression
```

```
352           {$$ = new MathExp($1,$3,$2); }
353       | expression '*' expression
354           {$$ = new MathExp($1,$3,$2); }
355       | expression "div" expression
356           {$$ = new MathExp($1,$3,$2); }
357       | expression "mod" expression
358           {$$ = new MathExp($1,$3,$2); }
359      | expression '/' expression
360           { $$ = new MathExp($1,$3,$2); }
361      | INTEGER OPERATOR_LESS_THAN_EQUAL exp
362           { $$ = new Condition(new Expression(Number($1)), $3, '<='); }
363      | IDENTIFIER OPERATOR_LESS_THAN_EQUAL exp
364           { $$ = new Condition(new Identifier($1), $3, '<='); }
365      | FLOAT OPERATOR_LESS_THAN_EQUAL exp
366           { $$ = new Condition(new Expression(Number($1)), $3, '<='); }
367      | INTEGER OPERATOR_LESS_THAN exp
368           { $$ = new Condition(new Expression(Number($1)), $3, '<'); }
369      | IDENTIFIER OPERATOR_LESS_THAN exp
370           { $$ = new Condition(new Identifier($1), $3, '<'); }
371      | FLOAT OPERATOR_LESS_THAN exp
372           { $$ = new Condition(new Expression(Number($1)), $3, '<'); }
373      | INTEGER OPERATOR_EQUAL exp
374           { $$ = new Condition(new Expression(Number($1)), $3, '=='); }
375      | IDENTIFIER OPERATOR_EQUAL exp
376           { $$ = new Condition(new Identifier($1), $3, '=='); }
377      | FLOAT OPERATOR_EQUAL exp
378           { $$ = new Condition(new Expression(Number($1)), $3, '=='); }
379      | INTEGER OPERATOR_GREATER_THAN_EQUAL exp
380           { $$ = new Condition(new Expression(Number($1)), $3, '>='); }
381      | IDENTIFIER OPERATOR_GREATER_THAN_EQUAL exp
382           { $$ = new Condition(new Identifier($1), $3, '>='); }
383      | FLOAT OPERATOR_GREATER_THAN_EQUAL exp
384           { $$ = new Condition(new Expression(Number($1)), $3, '>='); }
385      | INTEGER OPERATOR_GREATER_THAN exp
386           { $$ = new Condition(new Expression(Number($1)), $3, '>'); }
387      | IDENTIFIER OPERATOR_GREATER_THAN exp
388           { $$ = new Condition(new Identifier($1), $3, '>'); }
389      | FLOAT OPERATOR_GREATER_THAN exp
390           { $$ = new Condition(new Expression(Number($1)), $3, '>'); }
391      | INTEGER OPERATOR_NOT_EQUAL exp
392           { $$ = new Condition(new Expression(Number($1)), $3, '!='); }
393      | IDENTIFIER OPERATOR_NOT_EQUAL exp
394           { $$ = new Condition(new Identifier($1), $3, '!='); }
395      | FLOAT OPERATOR_NOT_EQUAL exp
396           { $$ = new Condition(new Expression(Number($1)), $3, '!='); }
397       | '~' expression %prec '*'
398           {$$ = new Integer(-Number(yytext));}
399      | INTEGER
400           { $$ = new Integer(Number(yytext)); }
401      | FLOAT
402           { $$ = new Float(Number(yytext)); }
403       | PI
404           {$$ = Math.PI;}
```

```
405     | IDENTIFIER OPERATOR_RECORD_GET LABEL
406        { $$ = new Get_entry_record($1, $3); }
407     | IDENTIFIER OPERATOR_RECORD_GET INTEGER
408        { $$ = new Get_entry_record($1, $3); }
409     | OPERATOR_CELL_GET IDENTIFIER
410        { $$ = new Get_cell_value($2); }
411     ;
412
413 exp
414     : INTEGER
415       { $$ = new Expression(Number(yytext)); }
416     | IDENTIFIER
417       { $$ = new Identifier($1); }
418     | FLOAT
419       { $$ = new Expression(Number(yytext)); }
420     ;
```

Listing A.1. Grammar of Glass Cat for the kernel language of Oz written in `oz.jison`.

A.3 How to add a new rule?

Glass Cat can construct the result directly if there is no procedure, but if there is one, it must first keep track of every steps inside the procedure. This is why, working with procedures is a little bit tricky but I tried to make it easy enough to add new rules. This section describes the few steps that must be done if you want to add a new rule, *e.g.* `my_rule` in the grammar.

A.3.1 `oz.jison`

Lexer

Make sure that tokens you need in your new rule are defined in the lexical grammar for the lexer, otherwise, add them. Take care that the first character of your token is not yet the first of another token, otherwise the first one to be defined will be chosen even if it is not the right one. *E.g.* if the token `:` (`COLON`) is defined, it must me after `:=` (`OPERATOR_CELL_ASSIGNMENT`), the scanner sees a colon but the `:=` cannot match with the second one.

Statement rules

If the rule you want to add is one that can be followed by another statement, add it in the `block` section with the code defined in A.2 which is as generic as possible, *i.e.* you have to change `my_rule` at line 1 with the name of your new rule.

```
1 | my_rule block
2       { $$=new Block();
```

```
3        if(!$2.isEmpty()) {
4           $$.add_semantics(@1, '', 4);
5           $2.add_semantics(@2, '', 5);
6        }
7        $$.eval_semantics($1.get_semantics(), $2.get_semantics());
8       }
```

Listing A.2. Add a new rule in a statement.

The first line of this code says that `my_rule` is followed by another `block`, line 2 means that the result of this sentence (`$$`) is a `new Block()` to which we add a new semantics (line 4) `add_semantics = function(state_loc, stack, type)` with the state localisation `state_loc` given by `@1`, the stack does not have a new entry `''` and the `type` is an integer which has a value for each rule [1]. Line 7 evaluates semantics as explained in Section 2.2.

Rule definition

Add your rule definition according to the mid-rule property defined in Listing 3.6 if you need to do something before the end of the sentence. Listing A.3 shows an example for the variable initialization.

```
1 variable_initializer
2    : variable_creation_id OPERATOR_ASSIGNMENT variable_assign
3        { $$ = new VariableInitializer($1, $3, cur_decl); }
4    ;
5
6 variable_assign
7    : expression
8    ;
```

Listing A.3. Define a new rule in the grammar (here adding a variable initializer).

Let us have a look at this code, we are getting used to the first line which is the name of the rule. Line 3 is by far the most interesting as others are grammar sentences. The third line means that the result of this sentence is a new object which takes the `variable_creation_id`, `variable_assign` and the `cur_decl` variable as parameter. The latter is employed in procedures in order to retrieve the semantics of the current sentence as we will see in the following steps.

[1] Actually, only the value 2 is reserved for an initialization because it means that another node in the stack must be skipped in order to browse only the identifier and its value and not the identifier alone when it was created.

A.3.2 ast_nodes.js

Global variables

While constructing AST nodes, we must be able to know a lot of things on what is happening, *e.g.* the identifiers that are declared, which one are cells....

Every global variable is enumerated and explained in the bullet list below.

- `var identifiers_list = {}`: Dictionary of declared identifiers (in their scope)
- `var cells_list = new Array()`: List of declared cells identifiers (in their scope)
- `var identifiers_nbr = {}`: Dictionary of identifiers and their number of declaration
- `var cur_condition = {}`: Dictionary of conditions
- `cur_condition[0] = true;`
- `var proc = new Array()`: List of procedures
- `var cur_proc = ""`: Stores everything that happens while parsing and executing
- `var cur_id = 1`: Identifier of the current IF THEN condition
- `var state_decl = new Array()`: List of states
- `var cur_decl = 0`: counter of the number of declaration (used with `state_decl`)

Create AST nodes

When the parser has found the correct sentence it applies the operation defined inside the brackets which is a new object in Glass Cat.

This is mandatory for the new object to take as argument `cur_decl`.

```
1  var VariableInitializer = function(identifier,value,cur_decl_local){
2     this.type = 'VariableInitializer';
3     this.identifier = identifier;
4     this.value = value;
5     this.semantic = new Array();
6     this.decl_nbr = cur_decl_local;
7     cur_decl++;
8
9     /* MORE CODE */
10 };
```

Listing A.4. Define a new AST node.

Every parameter must be put as a `this` variable to be accessible to parent calls. In order to be generic, a new node must have all the `this` variables which appears in the Listing A.4.

Moreover, if you use a reference to an identifier, you have to be sure that it is the last one created in the current scope. This can be done with a call to `get_last(id)`.

If you have to add more methods to this object, [Bak12] recommends to use the keyword `prototype` to define class methods in ordrer to minimize the size of the memory used.

A.3.3 Working with procedures

When you add a new rule, it must work even if there are procedures in the code. Therefore, we must keep track of everything that happens while the procedure is created and stored in a variable.

The global variable `cur_proc` contains a string of everything in the current procedure. When the keyword `end` is met at the end of the procedure definition, the `cur_proc` variable is stored in an array which key is the name of the procedure `proc[id]` then it is reset. When the procedure is called, the entry which corresponds to the name of the procedure is evaluated, *i.e.* the procedure is replayed.

Additionally, we have to add a new semantics for this function (lines 5-7). As this part is explained in the next section, the only thing you have to understand by now is that the state of the sentence must be stored in a global variable (`state_decl`) as when the function is replayed, new objects are created but do not have the power of the parser to add semantics except for the store.

Listing A.5 shows how we can add a new call in `cur_proc`.

```
1 cur_proc += "new Case('"+identifier+"', "+JSON.stringify(pattern)
  +","+cur_decl_local+");";
2
3 cur_proc += "if(is_good_pattern('"+identifier+"',"+JSON.stringify(
  pattern)+")){";
4
5 if(typeof state_decl[this.cur_decl] != 'undefined') {
6    cur_proc += "this.add_semantics("+JSON.stringify(state_decl[this.
     cur_decl])+",'',7);";
7 }
```

Listing A.5. Add a new call in `cur_proc` in order to replay it when a procedure is called .

A.4 How to add semantics?

Semantics is added to each object by the parser simply with the function `add_semantics` called on the object you created, *e.g.* `$1.add_semantics(@2, '', 4)`. Listing A.6 shows the content of this function for the `Block` object but it is the same everywhere.

```
1 Block.prototype.add_semantics = function(state_loc, stack, type) {
2    var sem = new Object();
3    sem.state_loc = state_loc;
4    sem.stack = stack;
5    sem.type = type;
6    this.semantic.push(sem);
7 };
```

Listing A.6. Add semantics to an object.

The function takes as parameters the state location in the format of the value given by `@n` in Jison, this is an object which contains information about the location in the code (see Listing A.7):

```
1 {"first_line":1,"last_line":6,"first_column":0,"last_column":3}
```

Listing A.7. Content of a `state_loc`.

The `stack` parameter is a string to add to the stack and the `type` an integer to know which operation adds semantics.

As explained in Chapter 6, we must evaluate the parse tree in a specific order (right, left, parent) to browse the semantics. This is done at each "root" note, *i.e.* once there is a `block` in the sentence. So you have to add a `$1.eval_semantics($1.get_semantics(), $2.get_semantics())` in the parser.

A.4.1 Working with procedures

In procedures, the parser is not there any more to add the right semantics so we have to store it in a global variable with the function `set_loc`. This method adds a new entry inside `state_decl` with the key set at `cur_decl`. When there is a procedure call, it knows to which state it corresponds and can browse it.

Moreover, we can now explain the code which was first presented in Listing A.5 and reminded in Listing A.8.

```
1 if(typeof state_decl[this.cur_decl] != 'undefined') {
2     cur_proc += "this.add_semantics("+JSON.stringify(state_decl[this.
      cur_decl])+",'',7);";
3 }
```

Listing A.8. Add semantics in procedures.

Once the state location is defined (after that the procedure has been parsed and so at the first call), semantics can be added.

A.5 Add tests

1. Open 'js/glass_cat.js' and add the following lines at the end of `generate_semantics` (below `cnt_exe++;`):

```
1       }
2       a = state.split("</p>").join("\\n");
3       a = a.split("<p>").join("");
4       console.log("add_test('"+a.substring(0,a.length-2)+"', '"+
        JSON.stringify(sem)+"');");
5   }
```

Listing A.9. Capture the store of a code to create a new test.

2. Load 'users/index-test.html' and write your code then press Execute. Open the console and copy the text from `add_test` to the end of the string.
3. Paste this string at the end of 'js/anim_dev.js'.

Appendix B

Users

Welcome in Glass Cat ! This chapter explains the different areas you can see on the website and codes you can type in.

Glass Cat is available at http://pbouilliez.github.io/glass-cat/, then you choose the user version.

B.1 A brief tour

The website is captured in Figure B.1.

There are many areas, each of them corresponds to a number:

1. A menu: access code examples, information about the author and Glass Cat.
2. A textarea: this is where you can type your Oz kernel language.
3. A program control: use to manage steps of the semantics (previous, next and execute to start parsing, GO TO step is a plus as the user can go quicker to a specified step).
4. States: display the states from right to left.
5. Stores (single-assignment and cell store): display the stores (up is the single-assignment one, down is the cell store).

Concretely if you want to use Glass Cat:

1. Write your code in the textarea (number 2).
2. Click on EXECUTE, if your code is correct, no error appears. Otherwise you will see an explanation of why Glass Cat cannot execute your code.
3. Click on NEXT to see the next semantics statement.
4. Click on PREVIOUS to go back on the previous one.
5. Click on GO TO if you want to go to a specific step.

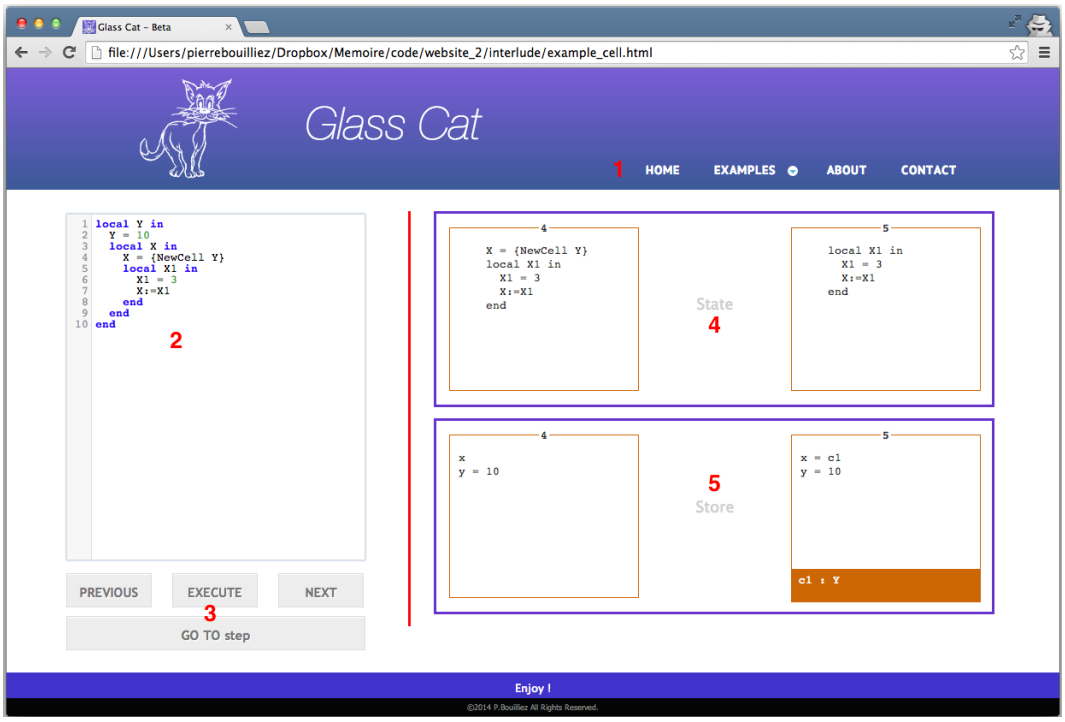


Figure B.1. Design of Glass Cat.

B.2 What can you do?

Glass Cat can only understand a part of the Oz kernel language. Therefore, you have to respect some rules.

Keywords

Identifiers Identifiers must start with a capital letter, followed by lower case, capital letter or numbers but cannot have an underscore in it. The following regex defines an identifier.

```
[A-Z][a-zA-Z0-9]*
```

Strings Strings must be surrounded by `" "`.

Variable declaration It must be written:

`local X in` ⟨s⟩ `end`

You cannot define more than one identifier at a time.

Variable initializer It must be written:

```
X = Y
X = Y+Z
X = Y+1
X = 1+1
```

You cannot define more than 2 items in an addition.

Records It must be written:

```
X = '|'(1:X1 2:X2 3:X3)
X = '#'(a:X1 2:X2 b:X3)
X = person(age:X1 sex:X2)
```

The label can be something between ' ' or a string starting with a lower case. Features must be integers or a string starting with a lower case.

Condition It must be written:

```
X = (Y == Z)
X = (4 == Z)
X = (Y == 4)
X = (4 == 4)
X = true
X = false
```

It must me surrounded by parenthesis.

The `if` statement It must be written:

`if` ⟨x⟩ `then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`

The identifier ⟨x⟩ must be a condition.

Pattern matching It must be written:

`case` ⟨x⟩ `of person(d:D) then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`
`case` ⟨x⟩ `of D then` ⟨s⟩$_1$ `else` ⟨s⟩$_2$ `end`

It can take records and identifiers as pattern.

Procedures It must be written:

```
Y = proc{$ X Z}
Y = proc{$ X ?R}
```

It must be anonymous procedures and if there are returned values, they must be written with a `?` preceding them.

Bibliography

[Bak12] Gregory Baker. Optimizing javascript code. https://developers.google.com/speed/articles/optimizing-javascript, 2012. [Online; accessed 15-May-2014].

[Bau13] L Frank Baum. *The Patchwork Girl of Oz*. Courier Dover Publications, 2013.

[Byn10] Mathias Bynens. Jison vs. peg.js. http://jsperf.com/jison-vs-peg-js, 2010. [Online; accessed 20-October-2013].

[Car09] Zach Carter. Jison. http://zaach.github.io/jison/, 2009. [Online; accessed 10-October-2013].

[Car13] Zach Carter. Jison - github (mid-rules). https://github.com/zaach/jison/issues/173, 2013. [Online; accessed 26-February-2014].

[CBR14] Sébastien Combéfis, Adrien Bibal, and Peter Van Roy. Recasting a traditional course into a MOOC by means of a SPOC. In *Proceedings of the European MOOCs Stakeholders Summit 2014*, pages 205–208, February 2014.

[CIAD12] Bill Campbell, Swami Iyer, and Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. CRC Press, 2012.

[CLCdSM12] Sébastien Combéfis and Vianney Le Clément de Saint-Marcq. Teaching programming and algorithm design with pythia, a web-based learning platform. *Olympiads in Informatics*, 6, 2012.

[CMS99] Stuart K Card, Jock D Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[Dev14] Google Developers. Closure compiler. https://developers.google.com/closure/compiler/, 2014. [Online; accessed 15-May-2014].

[Die07] Stephan Diehl. *Software Visualization*. Springer, 2007.

[For14] Bryan Ford. The packrat parsing and parsing expression grammars page. http://bford.info/packrat/, 2014. [Online; accessed 20-May-2014].

[FSF12a] Inc. Free Software Foundation. Bison - GNU parser generator. http://www.gnu.org/software/bison/, 2012. [Online; accessed 20-February-2014].

[FSF12b] Inc. Free Software Foundation. Bison - GNU parser generator. http://www.gnu.org/software/bison/manual/html_node/Table-of-Symbols.html, 2012. [Online; accessed 21-February-2014].

[GBWS11] Roger Gajraj, Margaret Bernard, Malcolm Williams, and Lenandlar Singh. Transforming source code examples into programming tutorials. In *ICCGI 2011, The Sixth International Multi-Conference on Computing in the Global Information Technology*, pages 160–164, 2011.

[GMT+05] Paul Gries, Volodymyr Mnih, Jonathan Taylor, Greg Wilson, and Lee Zamparo. Memview: A pedagogically-motivated visual debugger. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*, pages S1J–11. IEEE, 2005.

[Guo13] Philip J Guo. Online python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[Hav14] Marijn Haverbeke. Code mirror. http://codemirror.net/, 2014. [Online; accessed 15-February-2014].

[Hel09] Juha Helminen. Jype–an education-oriented integrated program visualization, visual debugging, and programming exercise tool for python. *Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology*, 2009.

[Hun02] Christopher D Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach. *Computers & Education*, 39(3):237–260, 2002.

[KP05] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.

[Lev09] John Levine. *Flex & bison*. O'Reilly Media, Inc., 2009.

[Maj10] David Majda. Peg.js. http://pegjs.majda.cz/, 2010. [Online; accessed 15-October-2013].

[Mig14] Matt Might. The language of languages. http://matt.might.net/articles/grammars-bnf-ebnf/, 2014. [Online; accessed 14-May-2014].

[MMC02] Jonathan I Maletic, Andrian Marcus, and Michael L Collard. A task oriented view of software visualization. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40. IEEE, 2002.

[MMSBA04] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces*, pages 373–376. ACM, 2004.

[Nel14] Randal C. Nelson. Context-free grammars. http://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html, 2014. [Online; accessed 14-May-2014].

[Nor08] Chris Northwood. Lexical and syntax analysis of programming languages. http://www.pling.org.uk/cs/lsa.html, 2008. [Online; accessed 14-May-2014].

[NRA+02] Thomas L Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, et al. Exploring the role of visualization and engagement in computer science education. In *ACM SIGCSE Bulletin*, volume 35, pages 131–152. ACM, 2002.

[Pos12] Jan Paul Posma. *jsdare: a new approach to learning programming*. PhD thesis, University of Oxford, 2012.

[Pro08] The Flex Project. flex: The fast lexical analyzer. http://flex.sourceforge.net/, 2008. [Online; accessed 14-May-2014].

[RLKS08] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. Effectiveness of program visualization: A case study with the ville tool. *Journal of Information Technology Education*, 7, 2008.

[S+12] Juha Sorva et al. Visual program simulation in introductory programming education. 2012.

[SJR+13] Piotr Sobieski, Christine Jacqmot, Benoît Raucent, Pascale Wouters, Jean-Marc Braibant, and Vincent Wertz. Dynamique des groupes – APP0, 2013.

[SK] Jorma Sajaniemi and Marja Kuittinen. Planani: A program animator based on the roles of variables.

[Uni11] Jive. http://www.cse.buffalo.edu/jive/, 2011. [Online; accessed March 2014].

[VR09] Peter Van Roy. Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music*, page 9, 2009.

[VR11] Peter Van Roy. Nsp. http://www.info.ucl.ac.be/~pvr/VanRoyHorizonsVol2.pdf, 2011. [Online; accessed March 2014].

[VR12] Peter Van Roy. LFSAB1402 - informatique 2, 2012.

[VRH04] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[Wik13] Wikipedia. Glass cat. http://en.wikipedia.org/wiki/Glass_Cat, 2013. [Online; accessed July 2013].

[Wik14a] Wikipedia. Comparison of parser generators — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Comparison_of_parser_generators, 2014. [Online; accessed 13-May-2014].

[Wik14b] Wikipedia. Parsing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Parse, 2014. [Online; accessed 14-May-2014].