Politechnika Łódzka

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Instytut Informatyki

Dariusz Jędrzejczak, 201208

Dual: a web-based, Pac-Man-complete hybrid text and visual programming language

Praca magisterska
napisana pod kierunkiem
dr inż. Jana Stolarka

Łódź 2016

Contents

Chapter 0

Introduction

0.1 Scope

This research spans a fairly broad area of knowledge, connecting – in order of importance – programming language design, web technologies, web application design and development as well as computer game development.

The main focus of this thesis is designing a programming language, which can have multiple deeply integrated editable representations.

I present a way to combine features of visual languages and text-based languages in an integrated development environment, which lets the programmer work with both representations in parallel or intertwine them in various ways.

A proof-of-concept interpreter and development environment for the language is implemented using web technologies.

Practical demonstration of the capabilities of the implementation is presented by writing a Pac-Man clone in the designed language[1]. This also provides a reference for assessing the performance of the implementation.

0.2 Choice of subject

The choice of this particular subject stems from my deep personal interest in programming language design. This research is an opportunity for me to create a project that demonstrates various ideas in this area that I developed over time and to explore and refine them further.

[1]The term "Pac-Man-complete" in the title of this thesis refers to a somewhat humorous description used by the Idris' programming language[66] author, Edwin Brady[104, 116], to describe the language. In the context of this dissertation it means that the designed programming language provides enough features to allow one to write a clone of the classic Pac-Man game in it.

0.3 Related work

With this research, I intend to explore certain aspects of programming language design as well as further the growth of visual programming languages, proposing a solution that improves over any existing comparable technology in terms of simplicity, expressive power of the language and usability.

The first and main part of this research is concerned with designing a programming language, which becomes the basis for the second part. This language, Dual, is mostly based on one of the oldest PL[2] families, the Lisp[107, 48] family. Lisp and its various dialects are consistently regarded as one of the most expressive PLs[86, 62], despite having a very simple syntactic and semantic core[102]. Dual is also influenced by many modern PLs, such as JavaScript (as defined by the ECMAScript[16]), which is the implementation language of its interpreter. Similarly to Lisp, Dual has a minimal syntax, although with some modifications and improvements (described in Chapter 2).

The second part of this research builds on top of theoretical[91] as well as practical[101] achievements in the field of Visual Programming Languages (VPLs), focusing on examining the latter. VPLs can be classified in various ways: [91, Section VPL-II.B], [92, Section Types of VPLs], [59, Section Definition]. I introduce my own classification by examining lanugages enumerated in [101]. Two major categories[3] that emerge from this classification are "line-connected block-based" and "snap-together block-based" VPLs. I design and implement a visual representation for Dual, which combines features characteristic to both of these categories. The development environment that is built for the language provides the ability to edit the text and the visual representation in parallel, with the changes made to one visible in the other immediately.

Visual languages are not especially popular compared to text-based languages. But recently they have been gaining more popularity, particularly in game development. Chief example and the main cause of this is Unreal Engine, the highly popular and mainstream[50, 113] game engine, which in the latest version introduced a visual programming language[94] as its primary scripting language. In fact this is the only scripting language that the engine supports, having dropped the UnrealScript language[109] included in the previous versions. This visual programming language will be compared to Dual to highlight its advantages.

The Dual language, both its representations and its environment – I will further use the terms "Dual system" or simply "system" to refer to these as a whole – are built entirely on top of the open web platform[41], which is ubiquitous. This gives the system great portability and makes the difficulty for the potential user to start working with it minimal. This is how the usability mentioned in the first paragraph of this section is defined.

[2]For the sake of terseness I will sometimes use the acronyms PL and VPL to abbreviate "programming language" and "visual programming language".

[3]By amount of languages that fall into each.

0.4 Goals

In line with the above, the purpose of this work is to introduce innovation as well as show a practical application of the developed solution. The concrete goals are:

- To design a programming language, which meets the criteria of expressiveness and usability outlined in the previous section.
- To provide a *general* design of the development environment for the language. This design must include an editor for a visual representation of the language, which must be directly mappable to the text form. Both forms must be designed to be used interchangeably.
- To implement a prototype of the designed environment, including an interpreter for the language, a text editor and a visual editor that conform to the main design requirements.
- To evaluate the practical usability and performance of the prototype by creating a clone of Pac-Man and examine the process as well as the results.
- To present possible ways of improving existing visual language systems.

0.5 Structure

This thesis is structured as follows:

Chapter 0 is this introduction.

Chapter 1 briefly describes technologies and tools used in developing any software described here as well as discusses the essential elements of the theoretical framework upon which the language was built.

Chapter 2 describes the design of the Dual programming language: its syntax, semantics, primitives, core functions and values. It also elaborates on programming language design in general.

Chapter 3 talks about the design of the language's visual representation and its development environment.

Chapter 4 describes the prototype implementation based on the designs. This includes the language and its intrepreter as well as the environment.

Chapter 5 contains a case study of a more-than-trivial application developed with Dual: a Pac-Man clone. Performance of the prototype implementation is assessed and possible adjustments and improvements are discussed.

Chapter 6 compares Dual to existing visual programming languages.

Chapter 7 summarizes and concludes.

Appendix A describes the contents of the DVD attached to this thesis and provides a short instruction on running the prototype.

Appendix B contains additional design ideas that may be implemented in the future.

Chapter 1

Background

This chapter briefly introduces the theoretical and practical components involved in design and implementation of the Dual programming language and its environment.

1.1 Web technologies

As stated in the introduction, one of the main design goals of the system is usability. This is accomplished in practice by building on top of the most accessible and ubiquitous platform – the web platform[119][1].

The language's interpreter and development environment are intended to work with and are built on web technologies: JavaScript, HTML5 and CSS. The prototype implementation makes use of Node.js, a server-side JavaScript runtime[69] and CodeMirror, a JavaScript library which provides basic facilities for the text-based code editor part of the system[73]. This part is modeled after modern web-oriented code editors with similar design philosophy[42], such as Visual Studio Code[79], Brackets[65], Atom[71] and many others.

1.1.1 Document Object Model

The Document Object Model (DOM) is an interface that lets JavaScript manipulate HTML documents as tree structures. The interface aims to be independent of a platform or programming language and is not limited to JavaScript and HTML[? ? ?].

In the DOM, each part of the document is represented by a node in a tree manipulable by JavaScript. All changes applied to the tree can be reflected in the document immediately and displayed.

The DOM structure can be manipulated with JavaScript in the following ways[?]:

- All of the document's elements and their attributes can be changed or removed.

[1]Also referred to as the *open* web platform[41]

- New elements and attributes can be added.
- The whole tree structure can be easily traversed, as each node contains references to many other nodes: its parents, children (with separate references to the first and last child) or siblings.
- All the CSS styles assigned to the document and individual elements can be changed. This means manipulating the details of how the elements are laid out and displayed. An enormous number of different display properties can be customized[**?**], such as colors, fonts, sizes, etc.
- JavaScript can react to all existing HTML events, such as clicking on elements, scrolling the page that contains the document, etc.
- JavaScript can create new events.

1.1.2 JavaScript

The JavaScript programming language was created by Brendan Eich for Netscape[16, Introduction], the company which created the Netscape Navigator web browser. There is a line of evolution that leads from Netscape and its browser to Mozilla and Firefox[87, 100]. The language was developed in 10 days in April 1995[8].

Despite significant design flaws, JavaScript has became *one of* the most[127, 121], if not *the most*[125, Section Most Popular Technologies per Dev Type][124] popular programming languages in the world. In this section I will briefly look at some of the probable reasons for that from a programming language design perspective.

From a programming language design perspective, JavaScript has many great features, borrowed from excellent languages[16, Section 4 Overview], most notably:

- Scheme, one of two main dialects of Lisp[19]. It is a minimalist, but very extensible functional programming language. The features drawn from this language include first-class functions (treating functions as values), anonymous functions (also known as lambdas or function literals) and lexical closures.
- Self, a pioneering prototype-based Object-Oriented Programming language[11], which evolved from Smalltalk-80[5]. It introduced the concept of prototypes, which is an approach to OOP, where inheritance is implemented by reusing existing objects instead of defining classes. Prototype-based programming is the feature that JavaScript adopted from this language.

The two above languages are characterized by a very minimalist nature. Both languages as well as JavaScript[26] are dynamically typed – types can be checked only at runtime.

The final advantage of JavaScript is the fact that it is distributed with a ubiquitous environment of the web browser. This makes the language straightforward for developers to use. Easy to get started – attract novice developers Reach billions of users[123]

The above mixture turns out to create a very powerful and usable language.

In the recent years JavaScript's popularity has been steadily growing[122]. This translated to significant improvements in the language's standarization efforts. Since 2015, ECMA International's[76] Technical Commitee 39[75], the commitee which defines ECMAScript – the official standard for JavaScript – adopted a new process. Under this process, a new version of the standard is released annually[16, 15, 14]. The documents and proposals are publicly available at GitHub[77].

Static type checking for JavaScript is also possible with Flow[74] – a static type checker, which works either as a syntax extension or through comment annotations – or TypeScript[78] – which is a superset of JavaScript.

Concurrency model

In the context of the concurrency model, the JavaScript runtime conceptually consists of three parts: the call stack, the heap and the message queue. All these are bound together by the event loop[23], which is the crucial part of this model. An iteration of this loop involves the following steps:

1. Take the next message from the queue or wait for one to arrive. At this point the call stack is empty.

2. Start processing the message by calling a function associated with it. Every message has an associated function. This initializes the call stack.

3. Processing stops when the stack becomes empty again, thus completing the iteration.

This means, at least conceptually, that messages are processed one by one, in a single thread and an executing function cannot be preempted by any other function before it completes. In practice this is more complicated and there are exceptions to these rules. But this explanation is sufficient for further discussion.

This model makes reasoning about the program execution very straightforward, but is problematic when a single message takes long to execute. The problem is observed e.g. when web applications cause browsers to hang or display a dialog asking the user if she wishes to terminate an unresponsive script.

For this reason it is best to write programs in JavaScript that block the event loop for as short as possible and divide the processing into multiple messages.

This concurrency model is called the *event* loop, because the messages are added to the queue any time an event occurs (an has an associated handler), such as a click or a scroll. In general input and output in JavaScript is performed asynchronously, through events, so it does not block program execution.

1.2 Design and implementation of Lisp

A very important family of programming languages and one which had the most influence on the design of Dual is the Lisp family. In this thesis I use the singular form “Lisp” to refer to the whole family rather that a concrete dialect or implementation – such as Common Lisp[18], Scheme[19], SBCL[84] or Racket[82] – unless otherwise noted.

Lisp is characterized by a very minimal syntax, which relies on Polish (prefix) notation for expressions and parentheses to indicate nesting. There are only expressions and no statements in the language. This means that every language construct represents a value. There is also no notion of operator precedence.

The two core components of a Lisp interpreter are the `apply` and `eval` functions[1, 61, Section 4.1]. The former takes as arguments another function and a list of arguments and applies this function to these arguments. The latter takes as arguments an `expression` and an `environment` and evaluates this expression in this environment. The typical implementation of `eval` distinguishes between a few types of expressions. The essential are:

- Symbols (also known as identifiers or names) – e.g. `velocity` – these are evaluated by looking up the value corresponding to the symbol in the environment, so `velocity` might evaluate to `10` if it is defined as such in the `environment`
- Numbers (or number literals) – e.g. `3.2` – these evaluate to a corresponding numerical value
- Booleans (boolean literals) – `true` or `false` – evaluate to a corresponding boolean value
- Strings (string literals) – e.g. `"Hello, world!"` – evaluate to a corresponding string value
- Quoted expressions – e.g. `'(+ 2 2)` – a quoted expression evaluates to itself; in other words a quote prevents an expression from being evaluated
- *Special forms* or *primitives*, which are expressions that have some special meaning in the language. These are the basic building blocks of programs. For example:
  - `if`, the basic conditional expression and other flow control expressions; the special meaning of these is that they evaluate their arguments depending on some condition
  - *lambda* expressions – essentially function literals, which consist of argument names and a body
  - *definition* and *assignment* expressions; these modify the environment; usually they treat their first argument as a name of the symbol in the

environment, so it is not evaluated; the second argument is evaluated and its value is associated with the symbol

A Lisp expression can look like this:

```
(+ 2 (* 3 5))
```

Words are delimited by white space. Each sequence of words between parentheses can be viewed as a list[2]. Lists can be nested inside each other. Each list represents an expression, where the first element of the list is the expression's operator and the following elements are its arguments.

This parenthesized notation is called S-expressions[106, Section Recursive Functions of Symbolic Expressions; Section The LISP Programming System]. These are used to represent both code and data. For example the code from Listing 1.2 can be represented (in Common Lisp[3]) as data:

```
(list '+ 2 (list '* 3 5))

; or shorter equivalent:
'(+ 2 (* 3 5))
```

Where `list` is a function that produces a list that contains the values of its arguments.

`'` is a *quote* symbol. It prevents an expression from being evaluated.

`;` begins a comment that extends to the end of current line.

`+` and `*` are functions that perform their corresponding arithmetic operations: addition and multiplication respectively.

This data representation can now be manipulated. For example:

```
; set the `expression` variable to hold the same list value
; as in the above listing:
(setf expression '(+ 2 (* 3 5)))

; the variable `expression` now represents
; the expression `(+ 2 (* 3 5))`

; replace `*` with `exp` in the `expression`
; since it is just an ordinary list,
; this is done with ordinary data manipulation functions:
(setf (first (third expression)) 'expt)

; the variable `expression` now represents
; the expression `(+ 2 (expt 3 5))`
```

Where `setf` evaluates its second argument and stores it in a variable represented by its first argument. The first argument can be – among other things[13, Section 11.15.1] – the name of the new variable or a place in an existing list.

[2]Originally the name Lisp was an acronym, which stood for "LISt Processing"[13, Section 1.2][7, Chapter 12]

[3]All the remaining listings in this section contain code in Common Lisp.

`first` and `third` take a list as an argument and extract the first or the third element from that list accordingly.

`expt` is a function that performs exponentiation: it takes two numerical arguments and returns the value of the first raised to the power determined by the value of the second.

We can now evaluate the `expression`:

```
; returns 245, which is the result of
; evaluating (+ 2 (expt 3 5)):
(eval expression)
```

`eval` here is a function of one argument – an expression to be evaluated in the current environment. This is "a user interface to the evaluator"[18, Section 3.8, Function EVAL] (which can be understood as the internal function `eval` described at the beginning of this section).

The property of representing code and data in essentially the same form is known as homoiconicity[44, 60, 6]. In the case of Lisp an S-expression can be very straightforwardly mapped to a corresponding Abstract Syntax Tree (AST) node.

1.2.1 Abstract syntax tree and program representation

The term AST refers to a tree data structure that is built by parsers of programming languages to represent syntactic structure of source code in an abstract as well as easily traversable and manipulable way. In the simplest form, in expression-only languages such as Lisp each node of such tree represents a single expression. The tree is abstract in the sense that it does not necessarily contain all the syntax constructs that occur in the source code or encodes them in some *abstract* way. In case of Lisp, there's no need to store or represent bracketing characters `()` in the AST, as nesting is inherent in the data structure itself.

In theory, a programming language does not require a text representation and could be defined only in terms of a data structure such as a syntax tree. Practically, for a language to be useful, it needs to come with an editable representation that provides a convenient way for a programmer to construct programs. Currently the most successful representation for that is the human-readable text-based representation, which evolved from more primitive and less convenient representations, such as punched cards[98, 43].

1.2.2 Text-based code editors

Constructing programs with text representation can be done with any text editor. This means that the representation is largely independent of a tool, which is an advantage. Any application capable of editing text can theoretically be used to edit any source code (ignoring details such as encoding, etc.). Such applications are universally available, so source code stored in text files can be edited freely on any platform with any tool.

But for complex programs a simple text editor quickly becomes inconvenient and a more specialized one is preferable. Such code editors introduce various features that greatly improve the convenience of working with a text-based representation of a programming language. For example:

- Automatic structuring of the text to emphasize blocks of code (auto-indentation).
- Highlighting of different syntactic constructs with different colors.
- Context-based auto-completion.
- Automatic correction of errors.
- The ability to fold distinct blocks of code.
- Advanced navigation through the code: jumping to declarations, definitions, other modules or files.

Most of these features require that the editor makes use of a parser to recognize the syntactic structure of a program.

Other advantages of a text representation, that stem from the multitude of ways that raw text can be manipulated and processed and are not related to any particular syntax:

- Find and replace with regular expressions.
- Selecting/processing many lines or even blocks of text.
- Editors often treat the source as a 2D grid of characters; each row and column of such grid can be numbered.
- Debuggers, compilers and other elements of a programming language system can use row and column numbers in error messages.
- Version control systems can easily compare (diff) and keep track of changes in text files.

1.2.3 Visual programming languages

An alternative representation is the one employed by visual programming languages. By a visual programming language I mean a language that "lets users create programs by manipulating program elements graphically rather than by specifying them textually"[59, 92].

Such languages are usually tied to a particular editor, which allows the programmer to edit the source code with a mouse rather than the keyboard. That is instead of typing in streams of characters to be parsed and assembled into a structural form, the programmer inserts, arranges and connects together distinct visual elements to produce such a structure. Thus I contend that visual programming can be defined at the lowest level as manipulating a visual form of a language's syntax tree.

The design of the visual representation for my language involved a rough survey of visual programming languages. In this section I will describe the results obtained from this survey.

I classified each of nearly 160 languages listed in [101], according to type of their visual representation into several categories.

Additionally, I associated each language with a number $s \in [0, 3]$, which descirbes its "structure factor". This quantifies my subjective assessment of the readability of the representation compared to familiar text representation ($s = 3$). For example, if it appears that the representation consists of scattered blocks, connected by lines and the layout seems to be arranged by the user, with no editor-support for automatic structuring, s will be low. In other words, the greater the number, the better structured the representation.

This analysis is not strict and systematic, but rather heuristic-based. A language is classified based solely on the screen shots from its environment. Its purpose is to assess general trends and determine which solutions gained the greatest adoption in practice. This is to aid the further design process.

Below I present the results of this classification in the form of a list. The items are organized as follows:

- «Name of category» – «percentage of languages that fall into the category» – «the average "structure factor" s for the category»

  «short description»

Here are the compiled results:

- Line-connected block-based – 66% – 0.61

  Blocks or boxes connected with lines or arrows.

- Snap-together block-based – 11% – 2.4

  Resembles familiar text representation, except that the structure is produced by snapping together blocks, as in jigsaw puzzles.

- Other representations – 23% – 1.39, notably:

  – List-based – 2.5% – 2

    Nested lists, possibly with icons.

  – GUI-based – 2.5% – 1

    Buttons with icons that represent various components.

  – Nested – 2.5% – 2

    Nested windows, boxes, circles or other "scopes". A border of each scope is clearly distinguishable.

- Enhanced text – 2.5% – 2.75

  Similar to text representation, but with differing font sizes, embedded widgets, or other enhancements.
- Timeline-based – 2% – 1.17

  Specialized for animations or music. Elements are placed on a timeline.
- Others – 11% – *varying*

  The remaining 11% are various other representations: experimental, in-game or game-based VPLs, hybrid, specialized, esoteric, etc. A few examples are presented at the end of this chapter, in Section 1.3

The section 1.3 at the end of this chapter contains screenshots from editors and environments for VPLs in each of the categories, in order in which they appear in the above list.

The above results help set possible design directions. We may conclude that in practice there are basically two main types of visual representations: "line-connected block-based" and "snap-together block-based".

I used two more heuristics to verify this conclusion:

- I analyzed the top hits when searching the phrase "visual programming language" with popular search engines, especially by images. Most results link to websites with information about VPLs based on these two main representations. I searched the phrase in Bing, Google, Yahoo and DuckDuckGo and out of the top 20 hits I counted 14-18 (depending on the search engine), which would qualify.

- I analyzed the Wikipedia article about VPLs[59]. The first paragraphs of the definition state:

  > [M]any VPLs (known as dataflow or diagrammatic programming)[89] are based on the idea of "boxes and arrows", where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations.

  This is essentially a description of the "line-connected block-based" representation.

  The example screenshot at the top of the article presents the MIT Scratch programming language, which falls into the "snap-together block-based" category. [91, Section VPL-II.B].

1.2.4 A note on history of VPLs

The prime example of a VPL that uses the second-most popular representation according to my classification is MIT Scratch. It is possibly the most popular educational VPL[53, Section Community of users] – even referred to as "the most popular VPL"[95]. One classification even calls the VPLs that use the "snap-together block-based" representation "Scratch & friends"[92, Section Types of VPLs].

However Scratch was not the first language to use this representation. It was preceded by logoBlocks and earlier similar projects[115].

In fact, if we go far back in history of VPLs, we eventually arrive at the Logo programming language, which was a dialect of Lisp[97, 51]. It was not a VPL by the definition quoted at the beginning of Section 1.2.3, as it did not have a way to manipulate *program elements* visually. Nonetheless, one author calls it "the first real mainstream VPL"[95]. This is because it pioneered many ideas related to visual programming and was inclined in that direction. As is reflected in its many derivatives, which were indeed VPLs[103], such as the visual programming environment of LEGO Mindstorms[90, 47, 83].

1.2.5 Common criticisms of VPLs

Among the most common general criticisms of visual programming languages are[88, 114, 117**?**], [46, Section Criticism]:

- There is a barrier of entry for programmers used to text-based languages.
- Essential programming tools are unavailable or cannot be applied: version control, side-by-side (or diff) comparison, change tracking, testing frameworks, build systems.
- Visual primitives take up significantly more space than text.
- Existing tools are of low quality.
- Performance is overall slow.
- There are no extensibility mechanisms.
- The target group seems to be novice users.
- There is no universal visual representation.
- VPLs create closed ecosystems.

I will address some of these criticisms in Chapter 3.

1.2.6 The problem with structure

The next few paragraphs outline the major problem with the most popular (as the results presented in Section 1.2.3 suggest) "line-connected block-based" visual representation.

The reason for its popularity might be that such diagrammatic representation is a very natural way of showing relationships between objects, often used when designing on a whiteboard[85] or with tools like Unified Modeling Language[**?**].

Nonetheless when used naively to visualise a program source code it has a major disadvantage.

Editors which use this representation usually leave the layout of the program source completely to the user, providing no automatic structuring. This may easily result in disorder and true “spaghetti-code”, where free-floating blocks are scattered around, connected by many intersecting lines. This is especially true for complex programs (See Figure 1.1).

This lack of support for automatic structuring, which is an essential feature of modern text-based code editors is clearly a regression.

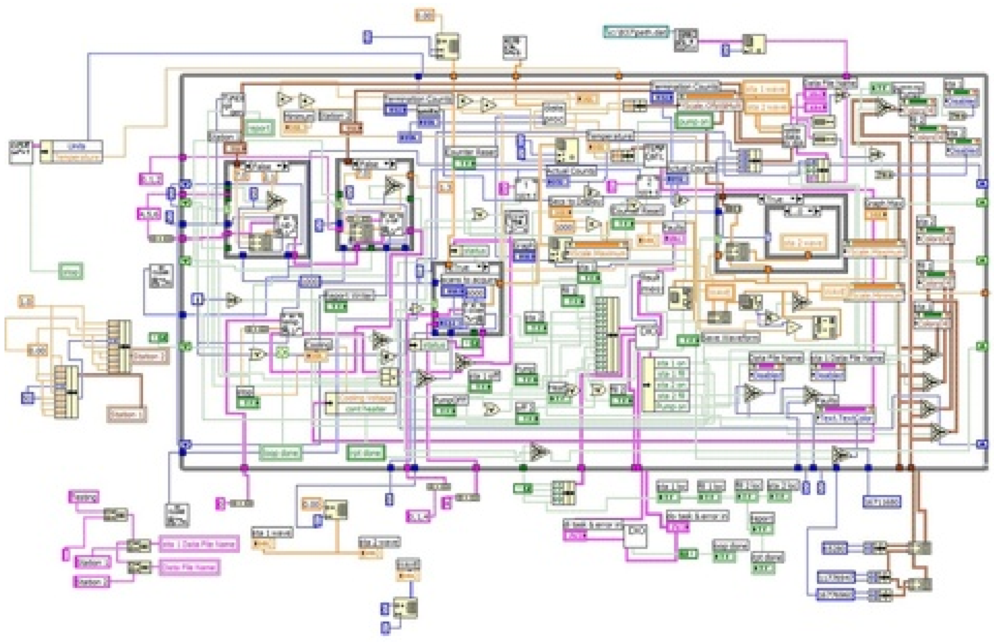


Figure 1.1: An example of a complex program represented with blocks connected by lines; screenshot from [137]

This problem does not occur in the second most popular VPL representation: the “snap-together block-based”.

There, the code is presented and manipulated in terms of visual blocks, which can be dragged and dropped by mouse and snapped together like jigsaw puzzle pieces. This representation is self-structuring and designed to resemble the familiar text-based, indent-structured representations.

1.3 Screenshots

This section presents screenshots that show examples of visual programming languages that fall into each of the categories discussed in Section 1.2.3.

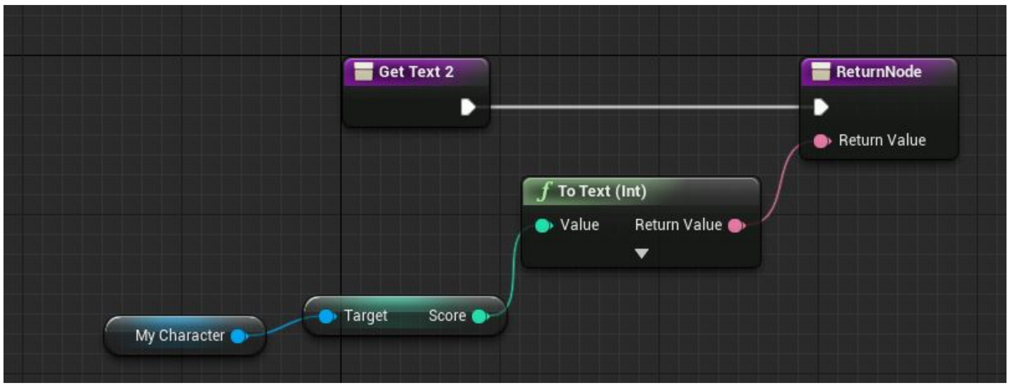
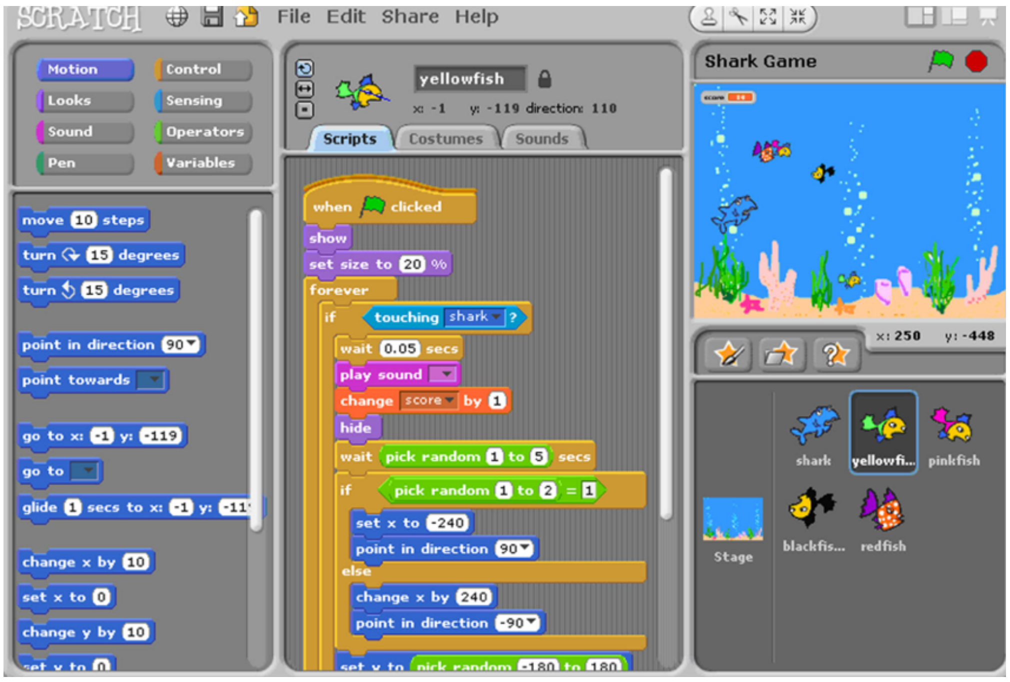
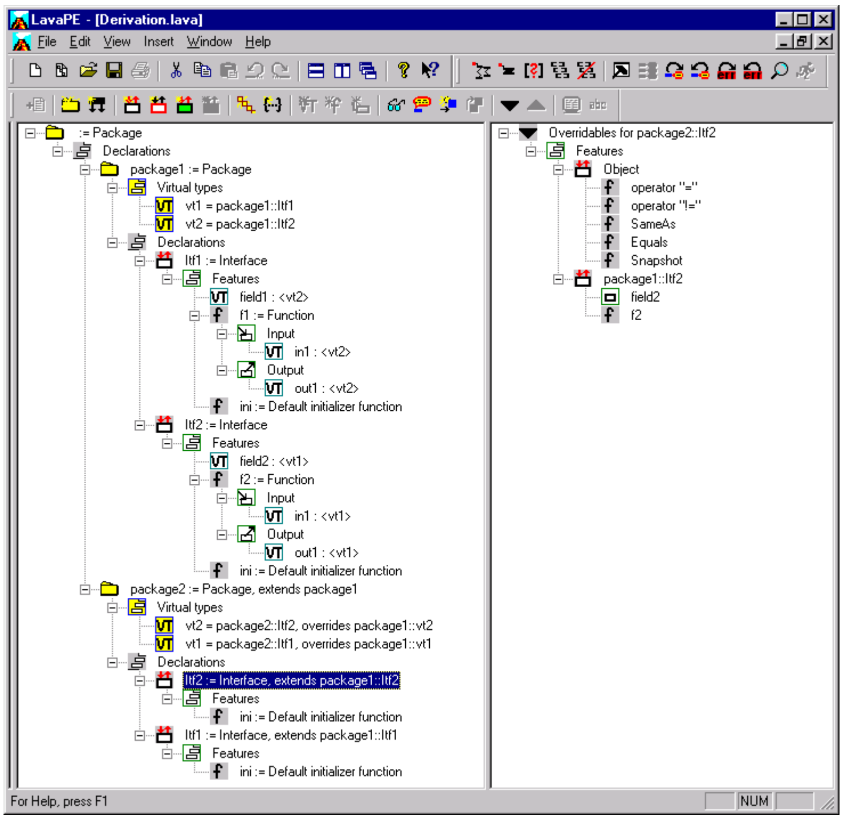


Figure 1.2: Blueprints Visual Scripting system; An example of a "line-connected block-based" VPL; screenshot from [129]



```
when clicked
show
set size to 20 %
forever
  if touching shark ?
    wait 0.05 secs
    play sound
    change score by 1
    hide
    wait pick random 1 to 5 secs
    if pick random 1 to 2 = 1
      set x to -240
      point in direction 90
    else
      change x by 240
      point in direction -90
    set y to pick random -180 to 180
```

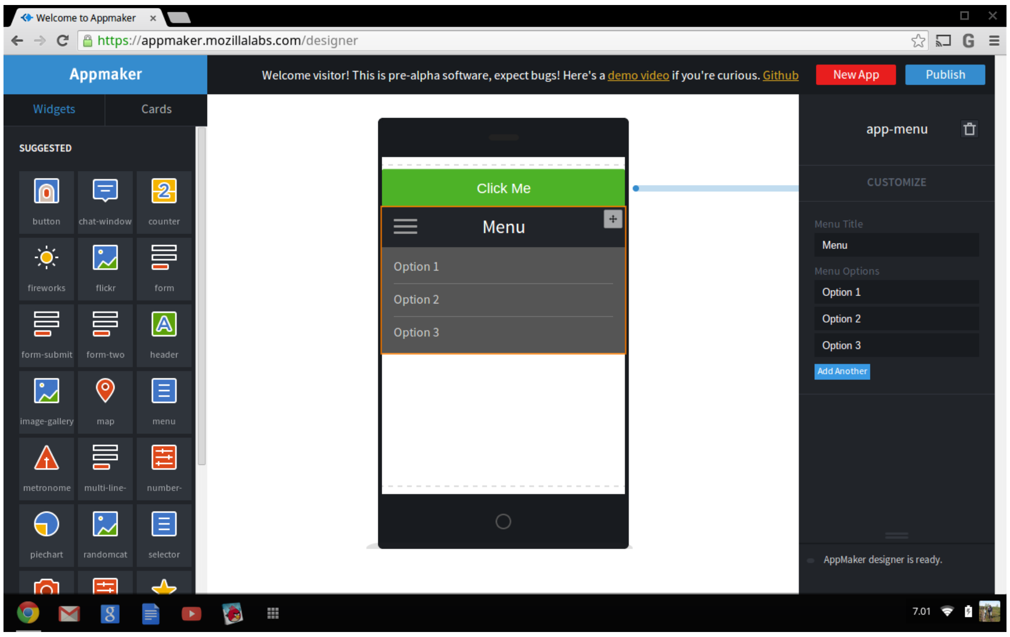
Figure 1.3: MIT Scratch programming language editor; An example of a "snap-together block-based" VPL; screenshot from [128]



```
:= Package
  Declarations
    package1 := Package
      Virtual types
        vt1 = package1::Itf1
        vt2 = package1::Itf2
      Declarations
        Itf1 := Interface
          Features
            field1 : <vt2>
            f1 := Function
              Input
                in1 : <vt2>
              Output
                out1 : <vt2>
            ini := Default initializer function
        Itf2 := Interface
          Features
            field2 : <vt1>
            f2 := Function
              Input
                in1 : <vt1>
              Output
                out1 : <vt1>
            ini := Default initializer function
    package2 := Package, extends package1
      Virtual types
        vt2 = package2::Itf2, overrides package1::vt2
        vt1 = package2::Itf1, overrides package1::vt1
      Declarations
        Itf2 := Interface, extends package1::Itf2
          Features
            ini := Default initializer function
        Itf1 := Interface, extends package1::Itf1
          Features
            ini := Default initializer function
```

```
Overridables for package2::Itf2
  Features
    Object
      operator "="
      operator "!="
      SameAs
      Equals
      Snapshot
    package1::Itf2
      field2
      f2
```

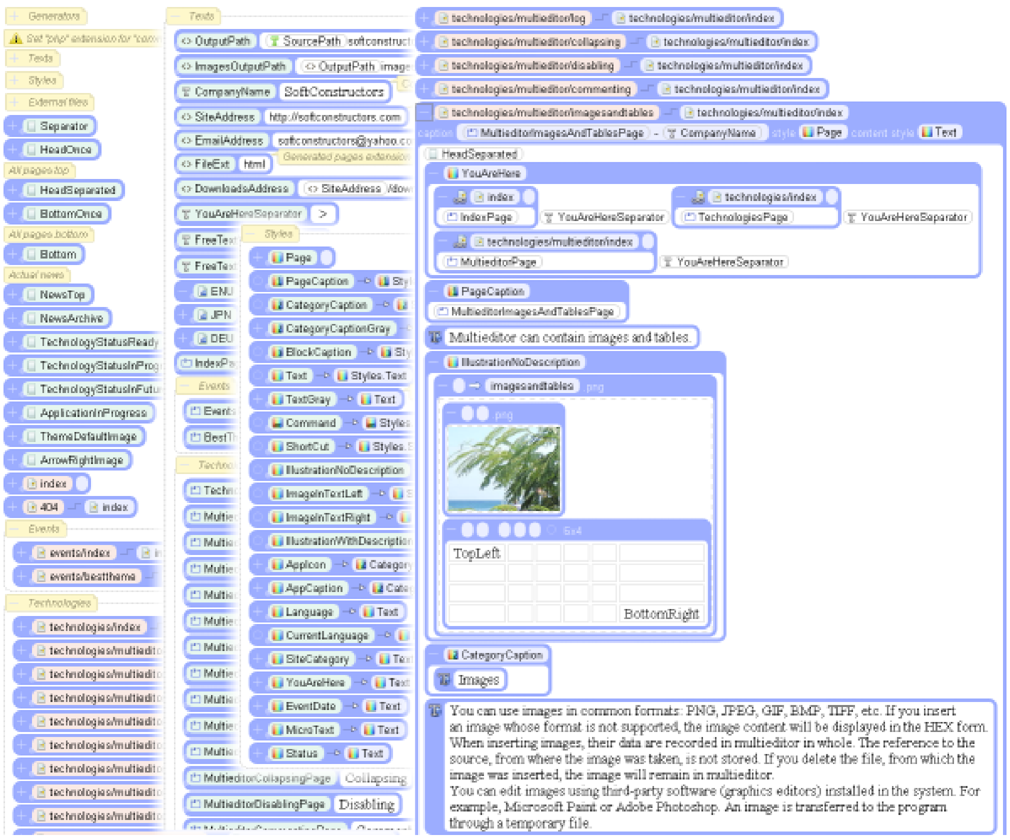
Figure 1.4: Lava programming language editor; An example of a “list-based” VPL; screenshot from [136]



Welcome visitor! This is pre-alpha software, expect bugs! Here's a demo video if you're curious. Github

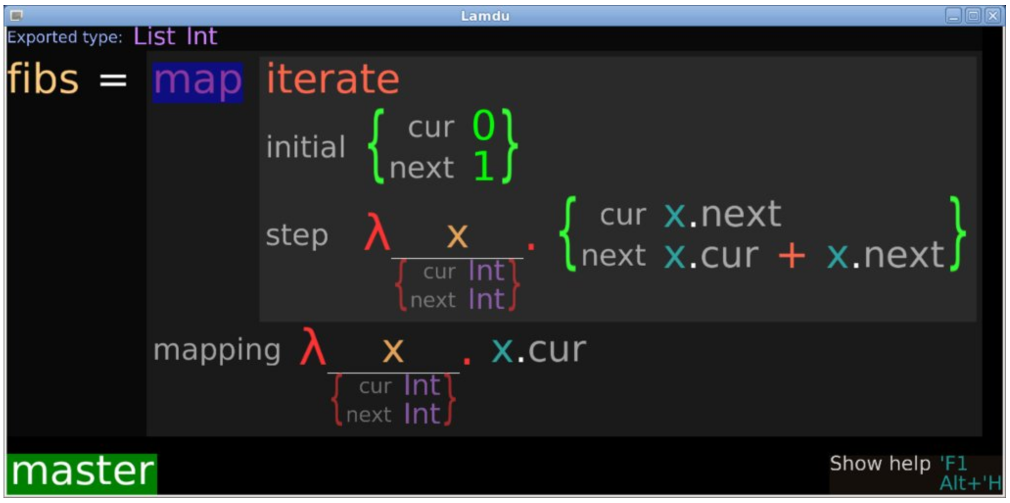
AppMaker designer is ready.

Figure 1.5: Mozilla Appmaker; An example of a “GUI-based” VPL; screenshot from [134]



You can use images in common formats: PNG, JPEG, GIF, BMP, TIFF, etc. If you insert an image whose format is not supported, the image content will be displayed in the HEX form. When inserting images, their data are recorded in multieditor in whole. The reference to the source, from where the image was taken, is not stored. If you delete the file, from which the image was inserted, the image will remain in multieditor.
You can edit images using third-party software (graphics editors) installed in the system. For example, Microsoft Paint or Adobe Photoshop. An image is transferred to the program through a temporary file.

Figure 1.6: StroyCode editor; An example of a “nested” VPL; screenshot from [135]



```
List Int
fibs = map iterate
               initial { cur  0 }
                       { next 1 }
               step λ x          . { cur  x.next          }
                      { cur  Int }   { next x.cur + x.next }
                      { next Int }
           mapping λ x          . x.cur
                    { cur  Int }
                    { next Int }
```

Figure 1.7: Lamdu visual environment; An example of an “enhanced text” VPL; screenshot from [?]

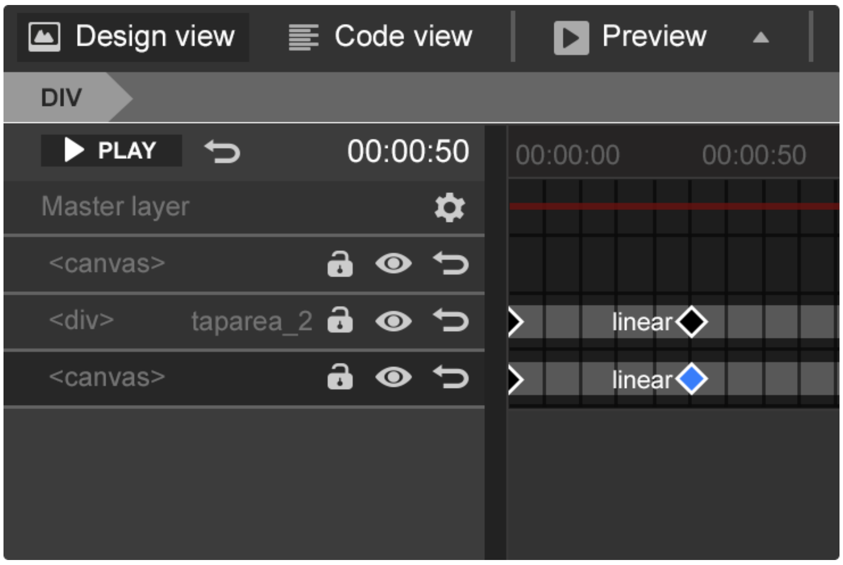


Figure 1.8: Google Web Designer; An example of a “timeline-based” VPL; screenshot from [133]

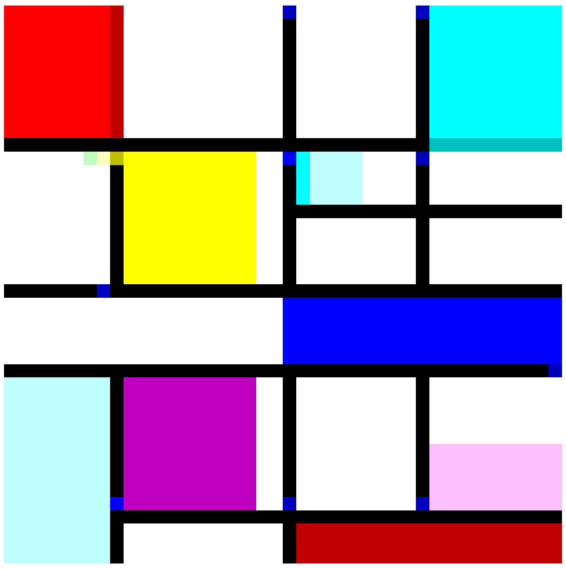


Figure 1.9: The esoteric programming language Piet; An example of an esoteric VPL; screenshot from [130]

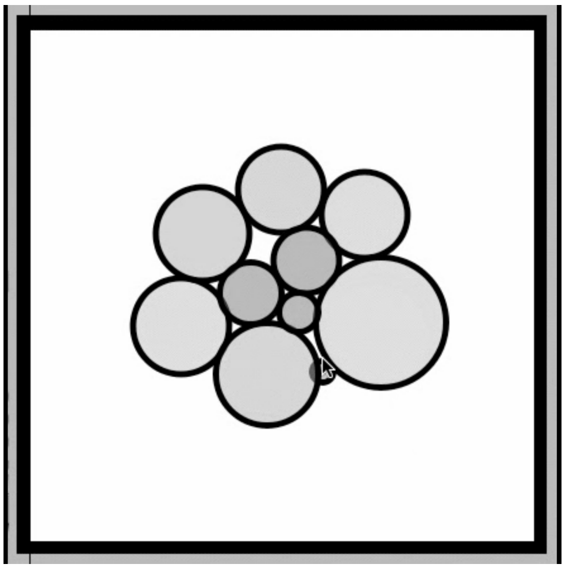


Figure 1.10: "Lily was a browser-based, visual programming environment written in JavaScript."[110]; An example of an experimental VPL; screenshot from [131]

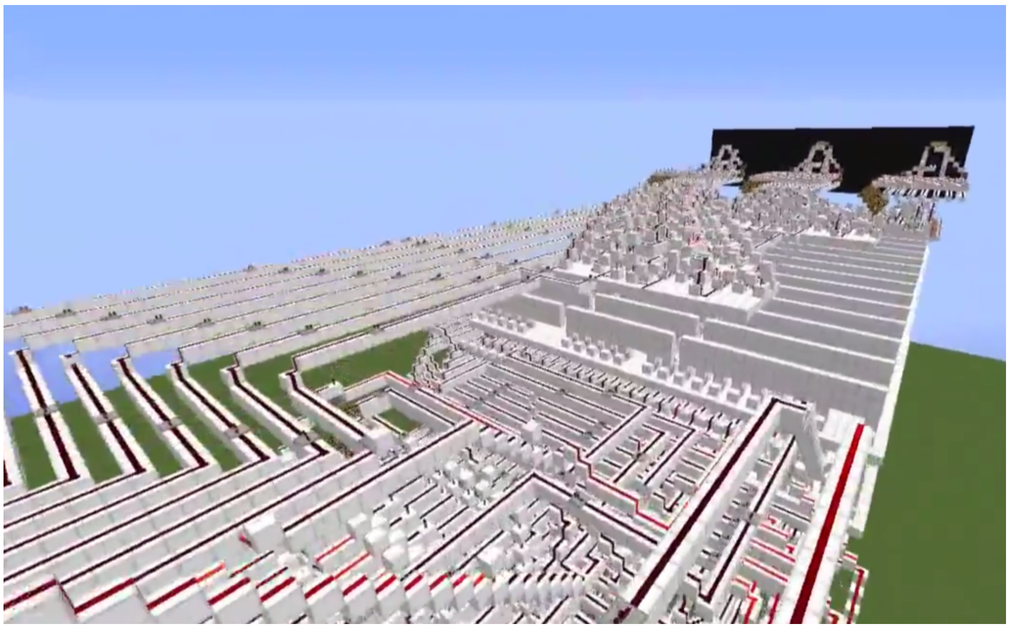


Figure 1.11: Minecraft[80] can be considered a visual programming lanugage. It's an example of a "game-based" VPL; "[S]omeone has created a fully programmable computer using Minecraft"[101]; screenshot from [132]

Chapter 2

Dual programming language

This chapter describes the design of the Dual programming language with focus on its text representation and its executable representation – the syntax tree. The most important concept, the Enhanced Syntax Tree, which enables complete integration of any number of additional representations with the language is also discussed.

2.1 Introduction

The evolution of programming languages is a gradual process. And so is the process of designing a single language. The approach that I found effective was iterative refinement, addition, testing, and sometimes subtraction of features. In practice this translates to intermediate designs and implementations being rearranged into new forms, with some discarded. I did not arrive at something that I could call the final form of the language, so a lot of the features described here are subject to change and improvement. I intend to work on this project further beyond the scope of this thesis.

The language was not originally intended to be a Lisp-like language or clone thereof, but throughout the research I ended up learning a lot about Lisp, sometimes by reinventing parts of this language. A somewhat philosophical interpretation of this would be that Lisp is built on fundamental principles that are (re)discoverable rather than invented.

In this and the following chapters I cover a lot of "design surface", only delving deep into some features that are relevant to core ideas that I wanted to convey in this thesis.

2.2 Syntax and grammar

Among the main design goals for the prototype of the language were simplicity and clarity. I wanted a language that is easy to parse and transform to a different representation. This restriction suggests that the syntax should be as minimal as possible.

I choose Lisp's syntax as the starting point. It is indeed almost as simple as one could imagine. But because of its almost complete uniformity it is often criticized. Some of the major criticisms are:

- In general it is hard to teach, because complex code gets easily confusing[10].
- The more nested the syntax tree, the harder it is to keep track of and balance parentheses; there tends to be a lot of closing parentheses next to each other in the source[64].

I made a few simple adjustments to the syntax in order to address these concerns, at least to some degree. These modifications do not significantly increase the complexity of a parser, but may considerably improve the syntax in terms of ease of use and readability for a human.

2.2.1 Basic syntax

Below I present the definition of Dual's grammar in left-recursion-free Backus–Naur Form. It is included here only for the sake of formality. I believe that for such a simple grammar BNF introduces more noise and is unnecessarily more complex than a textual description, possibly with the help of regular expressions or simply verbatim parser source code. For these reasons any extensions to this basic grammar will later on be introduced in these ways.

This is the BNF definition, a bit verbose for clarity:

```
<expression> ::= <word> | <call>
<call> ::= <operator> <argument-list>
<operator> ::= <word> <argument-lists>
<argument-list> ::= "[" <arguments> "]"
<word> ::= /[^\s\[\]]+/
<argument-lists> ::= <argument-list> <argument-lists> | ""
<arguments> ::= <expression> <arguments> | ""
```

BNF here is extended with the addition of a regular expression (between `/` delimiters) in the definition of `<word>`. The regular expression can be read as "any character which is not white space, `[` or `]`". This means that aside from white space, which acts as expression separator there are only two special characters that the parser has to worry about – the square brackets.

The above grammar definition is obviously very similar to Lisp's BNF description[105, 99].

The following expression in Lisp:

```
(+ 2 (expt 3 5))
```

has an equivalent expression in Dual:

```
+[2 expt[3 5]]
```

Comparing these, we may observe that in Dual:

- The primary bracketing characters are square brackets (`[]`) instead of parentheses. The reason for that design choice is that these are easier to type than parentheses or curly brackets (as they do not require holding the shift key), which matters considering the ubiquity of these characters in the source code.
- Expression's operator name is written *before* the opening bracket that precedes the list of arguments, as in `operator[argument-1 argument-2 ... argument-n]`.

Other than these two differences, Dual's notation is equivalent to S-expressions. Its advantages are:

- It is easier to parse by a human. Operators are clearer distinguished from operands. This is arguably because this notation is more familiar, bearing a similarity to the general mathematical notation (as in `f(x)`) and the most popular programming language syntax – the C-like syntax[1]
- If an expression has another expression as its operator, it is written as `op[args-1][args-2]`, which reduces the amount of nesting and thus the amount of identical bracketing characters appearing next to each other in the source code. Compare the equivalent S-expression: `((op args-1) args-2)`; and with multiple levels: `op[args-1][args-2][args-3][args-4]` vs `((((op args-1) args-2) args-3) args-4)`.

An interesting property of this syntax that, depending on the context, could be classified as an advantage, disadvantage or neither is that the sequence of characters `[[` is not legal, whereas in Lisp the analogous sequence `((` is.

Alas, this simple notation doesn't do away with a lot of other problems inherent in all minimal syntaxes, related to their homogenity. Later in this chapter I will introduce extensions and syntax sugar, which make the notation a little bit more diverse. Keep in mind that every special character that is introduced, is taken away from the set of possible `<word>`-characters, which implies that the regular expression for `<word>` is changed accordingly.

2.3 Comments

Comments are a basic and indispensable syntax feature of any programming language. I chose to include a comment syntax similar to the one found in Ada, Haskell or Lua:

```
-- a comment that extends until the end of the line

-- an expression that computes square root of 81:
sqrt[81]
```

[1]11 out of the top 20 languages as of June 2016[127] have C-based syntax (by this classification: [49]). If we extend the syntax family to Algol-like, its virtually 20 out of 20 – [126]. There are no languages with Lisp-based syntax among the most popular ones.

```
--[
    this is a multiline comment
    --[
        multiline comments can be nested

        as long as [ and ] are balanced, anything can be nested
           within
        multiline comments

        for example:
        --[
            this is a comment that includes a piece of code:
            *[7 7]

            which would evaluate to 49
        ]
    ]
]
```

2.4 Numbers

Numbers in the language are represented as JavaScript numbers. This means that there's only one number type – 64-bit floating point[2]. They are implemented as follows:

- When a word is tokenized by the parser, it is converted to a JavaScript number with a Number type constructor, which returns either the corresponding value (if the word is parsable to a number) or the value `NaN`. In the former case, the numerical value is stored in the appropriate syntax tree node, as its `value` property.
- Upon evaluation, a syntax tree node is checked for the `value` property. If it has one it is given as the result of the evaluation.
- The fact that a number is stored as a syntax tree node, which contains the its string representation and its raw value, both obtained from the source code during parsing means that conversion from a number literal to string is zero-cost, which could be useful for optimization.

Thus all of the following JavaScript number literals are valid in Dual:

```
1
357
3.14
0x11 // hexadecimal
0b11 // binary
0o11 // octal
5e-2 // exponential notation
```

[2]Defined by the ISO/IEC/IEEE 60559:2011 (IEEE 754) standard: [17, 112]

2.5 Escape character

An escape character \ is introduced. It allows special characters to be included in variable names.

For example:

```
word\ with\ spaces\ and\ braces\[\]
```

would be a single valid word and could be used as an identifier for a variable.

2.6 Strings

String values are introduced in Dual as follows:

```
'[A quick brown fox jumps over the lazy dog]
```

' is a special operator that produces a string value. It takes any number of arguments, which must be valid expressions.

Strings support variable substitution (also known as string interpolation[56]). Assuming we have a variable `animal-0` with the string value `"bear"` and another variable `animal-1` with the string value `"duck"`, this string:

```
'[A quick brown {animal-0} jumps over the lazy {animal-1}]
```

would evaluate to:

```
"A quick brown bear jumps over the lazy duck"
```

Special characters inside string can be escaped with the escape character \. Note that balanced square brackets that are part of syntactically valid Dual expression do not have to be escaped.

The implementation of strings is explained in detail in Section 2.8. String interpolation is explained in Section **??**.

2.7 Basic primitives and built-ins

This section enumerates and briefly describes Dual's basic primitives and built-in functions and values.

The items in Sections 2.7.1 and 2.7.2 are structured as follows:

- `«name» [«arguments»]`

  «description»

Where `«name»` is the name of the function/primitive and `«arguments»` are either the names that describe the arguments of the function/primitive or its arity. That is, the number of arguments that the function/primitive is defined for. This can be a fixed value (e.g `1`), a fixed range of values (e.g. `0..3`) or a range of values without an upper bound (e.g. `0..*`, which means 0 or more). An argument name can optionally contain a colon character `:`, which is followed by the type that the argument is expected to have.

«description» is a brief description of the function/primitive.

2.7.1 Functions

The following basic functions are defined in the language:

- `list [0..*]`

  Returns a JavaScript `Array`[21], which contains the values of its arguments.

- `$ [0..*]`

  An alias for `list`.

- `apply [f args]`

  Works like Lisp's `apply`: it takes a function and a list of arguments and returns the result of applying the function to the arguments.

- `log [0..*]`

  Wraps JavaScript's `console.log` method[24]. It outputs the values of its arguments to JavaScript's standard output – web browser's console.

- `typeof [arg]`

  Wraps JavaScript's `typeof` operator. "[R]eturns a string indicating the type of the unevaluated operand."[32]

- `or [a b]` and `and [a b]`

  The basic logical operators – analogous to `||` and `&&` in JavaScript.

- `any [0..*]` and `all [0..*]`

  Like the above, but accept variable number of arguments. These return either `true` or `false`.

- `not [arg]`

  The logical negation operator (`!`).

- `mod [arg]`

  The modulus (`%`) operator.

- `-# [arg]`

  The unary minus operator. It negates its argument.

- `sum [0..*]` and `mul [0..*]`

  Perform summation and product operations on any number of arguments.

- `to-int [arg:number]`

  Converts its argument to an integer value, by truncating the decimal part.

- `strlen [str:string]`

  Returns the length of `str`.

- `str@ [str:string n:integer]`

  Returns the `n`th character of `str`.

Moreover, all basic binary inequality operators `<, >, <=, >=` as well as all basic binary arithmetic operators `+, -, *, /` are supported. Comparison operators: `=` and `<>` are equivalent to JavaScript's `===` and `!==`, which means they perform strict comparison, without implicit type conversion[22, Section Equality operators].

2.7.2 Language primitives

The Dual language supports the following primitives:

- `bind [name value]`

  Evaluates its second argument and binds this value to the name of the first argument. This name is bound within the current scope. This is a basic construct for defining variables, like `var` or `define` in other languages. Significant semantics here are that new scopes are introduced by function bodies, macro bodies and match expression bodies. The primitive also supports pattern matching to deconstruct the value and bind its components to possibly several variables. In that regard it works a lot like JavaScript's destructuring assignment[25] or similar features in other languages, such as Perl or Python. This primitive can be used only for binding names that don't exist in the scope at the point of its invocation. There are other constructs for mutating and modifying existing variables. There is no hoisting[34, Section var hoisting], as definitions are processed in order in which they appear in code.

  For example:

  ```
  bind [greeting '[Hello]]
  ```

- `if [condition consequent alternative]`

  This primitive serves as a basic conditional evaluation construct. Its semantics are like those of the analogous construct in Lisp. It accepts 3 arguments: first the `condition` expression, then the `consequent`, that is, the expression to be evaluated if the value of the condition is *not false* (note that this is a strict rule; any other value than `false` is interpreted as `true`; every conditional construct in the language follows this rule). The third argument, the `alternative` is the expression that is evaluated otherwise.

- `do [0..*]`

  Evaluates its arguments in order and returns the value of the last argument. Fulfils the role of a block of expressions.

  For example:

  ```
  if [<[a 2] do [
      bind [b -[2 a]]
      log['[difference b:] b]
  ] do [
      bind [c -[a 2]]
      log ['[difference c:] c]
  ]]
  ```

- `while [condition body]`

  A basic loop construct. If `condition` is equivalent to *not false*, evaluates `body`. Repeats these steps until `condition` evaluates to `false`. Returns the value of the last evaluation of `body` or `false` if the body was not evaluated.

- `mutate [name value]`

  If a variable identified by `name` is defined within the current scope or any outer scope, changes (mutates) its value, so it now refers to the result of evaluating the `value` argument. The scopes are searched from the innermost to the outermost, in order. If the `name` argument doesn't identify any variable, an error is thrown. Returns the scope (environment), in which the primitive was evaluated.

- `dict [0..*]`

  Creates and returns a JavaScript object. It takes an even number of arguments. Arguments are considered in twos, as key-value pairs. These pairs determine properties for the new objects. Keys, which must be words, are property names and their corresponding values, which can be arbitrary expressions, are the values of these properties.

  For example:

  ```
  -- creates an object with four properties and assigns it
  -- to variable ‘car‘:
  bind [car dict [
      id 0
      brand '[Ford]
      model '[Mustang]
      year 1969
  ]]
  ```

- `assign [2..*]`

  A wrapper for JavaScript's Object.assign()[27]. It copies the values of all properties from one or more source objects to a target object. Returns the target object. The first argument is the target object, the following arguments are the source objects.

- `code' [arg]`

  Returns its argument without evaluating it. Used in macros, which return unevaluated code, which is substituted in the syntax tree and only then evaluated.

- `macro [1..*]`

  Returns a macro value. The last argument is the macro's body. The preceding arguments are the patterns for the macro's arguments. See Sections 2.10 and 2.12 for details.

- `of [1..*]`

  Returns a function value. The last argument is the function's body. The preceding arguments are the patterns for the macro's arguments. See Section 2.10 for details.

- `of  [2..*]`

  Returns a function value. Treats its penultimate argument as the function's body and all the preceding arguments as patterns for the function's arguments. See Section 2.10 for details.

  The last argument is used when the function is called and the values supplied as arguments do not match the patterns. If the argument is a function, it will be called with the same values as arguments and if it is a value it will be returned.

  This enables chaining functions together like so:

```
bind [f
    of~ [a b c log['[called with three arguments]]
        of~ [a b log['[called with two arguments]]
            of~ [a log['[called with one argument]]
                log['[called with zero or more than three
                   arguments]]
            ]
        ]
    ]
]

-- will log "called with two arguments":
f[3 2]
```

- `procedure [body]`

  Returns a function value. Its only argument is the function's body.

- `match [2..*]`

  Performs pattern matching. Its first argument is an expression to be matched. The following arguments are two-element lists, where the first element is the pattern to be matched and the second the expression to be evaluated if it matches. See Section 2.10.2 for details.

- `cond [1..*]`

  It works like a nested `if-else`s and similarly to Lisp's `cond`[18, Section 5.3, Macro COND]. Its arguments are two-element lists, where the first element is a condition that should evaluate to a boolean value and the second is the expression to be evaluated if it the condition is true. It evaluates at most one expression: the one that has a true condition. Conditions are checked in order.

- `. [2..*]`

  Property accessor. Essentially works like JavaScript's `.` operator[28]:

  For example:

  ```
  .[window Date now][]
  ```

  translates to JavaScript as:

  ```
  window.Date.now();
  ```

  If a property cannot be accessed, an error is thrown.

- `: [3..*]`

  Works symmetrically to `.` – it sets a property to a value specified by its last argument.

  For example:

  ```
  :[game-state hero ammo 5]
  ```

  translates to JavaScript as:

  ```
  gameState.hero.ammo = 5;
  ```

  If a property cannot be accessed, an error is thrown.

- `@ [arg]`

  Identity operator. Returns the value of its argument.

- `async [1..*]`

  Its first argument should be an asynchronous JavaScript function, such as `requestAnimationFrame`[38]. It applies this function to its remaining arguments.

2.7.3 Values

The following values are also defined:

- `true` and `false`

  Evaluate to their respective boolean values. `_` is an alias for `true` when used outside of pattern-matching. This enables a convenient compatibility between `match` and `cond`: if we're matching a single value and want to have a default case, then `_` is used to match any value. Similarly, if `_` is given as a condition in the last alternative of `cond`, it will evaluate to `true` and work as the default case.

- `undefined`

  Evaluates to JavaScript's `undefined` value. It is a primitive type that is used by the language to mark values that have not been assigned a value. Also, functions that do not explicitly return a value, return `undefined`[33].

- `window`

  Provides access to JavaScript's global `window` object[36].

2.8 Enhanced Syntax Tree

In order to enable full mapping between any number of program representations at the syntax-level, a modification of an AST was designed as a data structure representation of Dual's syntax. I call this structure the Enhanced Syntax Tree (EST). This crucial element in the language's design is described in this section.

The primary representation of a program in Dual is the EST. Although itself not directly editable, it can contain references to any number of editable representations, such as the text and visual ones.

These other representations contain back-references to the EST. Thanks to this, a change to any of the representations can be propagated to every other representation.

Every representation must come with:

- A way to translate it to an EST.
- A way to generate it for a given EST.
- A way or ways to manipulate it.

While translating, generating and manipulating, it must be ensured that each entity of the representation has a bidirectional association to a corresponding EST node.

For example, for text representation:

- Translation to EST is done with a parser.

- Generation from EST is done with an unparser[57].
- Manipulation is done with a text editor.

To ensure that the associations are kept, there must be objects that represent "text fragments". These objects then must contain references to corresponding EST nodes and vice versa.

White space characters and comments have no semantic significance, unless serving as separators could be considered one. After parsing, bracketing characters also serve no purpose and can be safely discarded, without influencing the meaning of the program. This is indeed done when constructing an AST from text in most programming languages.

In case of Dual though, no characters are discarded. Instead, white space, comments, brackets and any other characters are included in the EST, connected to appropriate nodes. Storing all characters in the EST means that the entirety of text representation, in structural form, is accessible straight from the syntax tree. This allows an unparser to recreate it *exactly.*

Such design greatly simplifies the implementation of and integrates with the language features such as:

- Automatic indentation. The EST contains all white space. If a new node is inserted into it, it can be initialized with white space of its siblings and/or ancestors, etc..
- Documentation comments. Comments can easily be associated with corresponding code blocks (EST nodes), which can be useful for automatically generating documentation in any format.
- Any expression can be unparsed to its original form straight from syntax tree, which can be used for debugging. For example, if a Dual program is built by manipulating visual representation entirely without the use of text and an error occurs while interpreting it, a single EST node – the one that contains the erroneous expression – can be unparsed and presented by the debugger in editable form. This may allow the user to fix the error quicker than manipulating blocks, without the need to unparse the entire program.
- Any expression can be stringified (serialized) on-the-fly and this string can be used as a value in the program or stored in a file.

2.8.1 Structural representation of strings

The last feature from the above list provides an interesting way of implementing strings in the language. Instead as streams of characters, they could be kept in structural form – as syntax trees. In combination with pattern matching this enables language-native structural manipulation of strings[3]. For example we could write:

[3]See: [120] and [52, Section Pattern matching and strings] for similar concepts.

```
bind [str '[A quick brown fox jumps over the lazy dog]]

bind [words [_ _ third-word {rest}] str]

bind [characters [_ _ third-letter {rest}] third-word]

-- logs "o" to the console:
log [third-letter]
```

Where `words` deconstructs a string into individual words and binds these words to identifiers provided as its arguments. `characters` performs an analogous operation on the character-level. The notation `{rest}` matches zero or more arguments (see Section 2.11 for details). `log` outputs the values of its arguments to the JavaScript console.

A downside here is that such representation of strings is not very efficient. A simple optimization would be to keep the raw form of the string (a stream of characters) as a value in the corresponding syntax tree node. So the raw representation is extended instead of replaced. Having these two forms alongside each other would enable the programmer to use the familiar string manipulation methods as well as structural manipulation without significant performance impact.

2.9 Syntax sugar for function invocations

In order to reduce the amount of *closing* brackets appearing next to each other in program's text, two additional simple notations were introduced. The first is addition of the pipe special character (`|`). It is used for single-argument functions.

If a function is invoked with only one argument, we can omit the closing bracket `]` and replace the opening bracket `[` with `|`. `|` can be viewed as a right-associative "invocation operator". For example:

```
foo | bar | baz
```

is equivalent to:

```
foo [bar [baz]]
```

The parser produces equivalent syntax trees for the above cases.

Below I present more examples to illustrate the utility of this syntax. Note that <=> symbol used in comments means "equivalent to":

```
-- compute factorial of 32:
-- <=> factorial[32]
factorial|32

-- find 9th Fibonacci number:
-- <=> fibonacci[9]
fibonacci|9

-- compute sine of pi:
-- <=> sin[pi]
```

```
sin|pi

-- compute cosine of the number that is
-- the result of multiplication of pi and 5!:
-- <=> cos[*[pi factorial[5]]]
cos|*[pi factorial|5]

-- convert 33.2 to an integer (truncate .2):
-- <=> to-int[33.2]
to-int|33.2

-- construct a list with one item,
-- which is a string "hello":
-- <=> list['[hello]]
list|'|hello
```

Another special character (`!`) was introduced for analogous use in zero-argument expressions (procedures):

```
-- invoke a procedure that changes
-- some state variables in its outer scope:
-- <=> set-initial-state[]
set-initial-state!

-- sum two random numbers:
-- <=> +[random[] random[]]
+[random! random!]

-- bind a value returned by an immediately
-- invoked procedure to an identifier:
-- <=> bind [forty-two procedure [42][]]
bind [forty-two procedure [42]!]

-- evaluates to 42:
forty-two
```

2.10 Pattern matching

A simple, yet powerful pattern-matching facilities were added to the language, which further extend its expressive power.

Pattern matching works with bindings, functions, `match` primitive and macros.

The pattern matching works in a way similar to most other languages that support this feature (e.g. ML family). The general rules are:

- A literal (strings or numbers are supported) value matches itself:

  ```
  -- computes factorial of a number:
  bind [factorial
       of~ [0 1
       of~ [n *[n factorial[-[n 1]]]]]
  ]
  ```

```
-- logs ‘120‘:
log [factorial|5]
```

- An identifier (word) matches any value, which is then bound to the identifier:

```
bind [simple-print of [x log|x]]

-- logs ‘3‘:
simple-print[3]
```

- A wildcard pattern (_) matches any value, but does not bind:

```
-- returns its third argument, discards the rest:
bind [get-third of [_ _ x x]]

-- logs ‘3‘:
log [get-third[1 2 3]]
```

  As such it can be useful for discarding some values, depending on other values or extracting some values from a structure (see next point).

Also the following expression-patterns are supported:

- `list` or `$` is used to destructure lists:

```
bind [$[_ _ third-element] $[0 1 2]]

-- logs ‘3‘:
log [third-element]

-- it works for arbitrarily nested lists as well:
bind [
    $[  _ $[  _  pick  _   _]   _]
    $['|a $['|b  '|c '|d '|e] '|f]
]

-- logs ‘c‘:
log [pick]
```

- Comparison operators (`= < <= >= <>`) match if a value passes the comparison; it can be viewed as a shorthand notation for simple guards[9, Chapter Pattern Matching Basics, Section Using Guards within Patterns]:

```
-- returns the sign of a number
-- note: ‘-#‘ is the unary ‘-‘ operator:
bind [sign
    of [=|0 0
    of [<|0 -#|1
    of [>|0 1]]]
]

-- logs ‘-1‘:
log [sign|-77]
```

Other pattern-expressions are not supported and using them will result in a mismatch.

2.10.1 Destructuring

Pattern matching works with bindings, the language allows destructuring assignments and definitions[25]. An example of a such definition would be:

```
bind [
    $[a b $[c d]]
    $[1 2 $[3 4]]
]

bind [
    $[ _  x  y  {  rest  }]
    $['|a '|b '|c  '|d '|e  '|f]
]

-- logs ‘1 2 3 4‘:
log [a b c d]

-- logs ‘b c ["d", "e", "f"]‘:
log [x y rest]
```

2.10.2 `match` primitive

The above examples show pattern matching used in function definitions and for destructuring values by binding their components to identifiers. There is also the `match` primitive, which can serve the role of a `switch` statement from C-like languages. It is however much more powerful, as any complex values supported by the pattern matching system can be matched, including lists. This allows switching on multiple values and in any combination.

The `match` primitive's first argument is a value to match and all subsequent arguments are two-element lists, where the first element is the pattern to match and the second is the expression to evaluate in case of a match. The primitive tries the matches in order and only evaluates the expression related to the first successful match. The subsequent matches are not evaluated.

Here are example uses for `match`:

```
bind [state '|game-on]

-- will execute the ‘play‘ procedure:
match [state
    $['|game-on play!]
    $['|game-paused display-pause-menu!]
    $['|game-screenshot capture-screenshot!]
]

-- ...
-- note: . is the access operator
-- .[a b c] is equivalent to a.b.c in other languages
bind [$[x y] .[player postion]]

-- we can easily replace complex conditions:
match [$[x x y y]
    $[
        $[>|0 <|screen-width >|0 <|screen-height]
        log|'[player visible]
    ]
    $[_ log|'[player not visible]]
]
```

2.11 Rest parameters and spread operator

Another syntax extension that I introduced involves two additional special bracketing characters: `{` and `}`, which serve several purposes:

- If used in function definitions, they indicate that a function or macro is variadic – it accepts a variable number of arguments[58].

  If a function is invoked with an equal or greater number of arguments than stated in its definition and the last argument's name in this definition is specified between `{` and `}` then any extraneous arguments are available in the function's body in a list with the name that was specified inside the curly braces. The order of arguments is preserved. For example:

  ```
  bind [variadic-function of [a b {args}
      log [a b args]
  ]]

  -- logs `1 2 [3, 4, 5, 6]`:
  variadic-function[1 2 3 4 5 6]
  ```

  This is essentially the same as the "rest parameters" mechanism known from Lisp[13, Section 12.2.3], recently also adopted in JavaScript (as of the ECMAScript 2015 standard[29]).

- The above extends beyond function definitions. It works in any place, where pattern matching works:

  ```
  bind [$[a b {rest}] $['|a '|b '|c '|d '|e]]

  -- logs `["c", "d", "e"]`:
  log [rest]
  ```

  This enables non-exact matching. If only the first few elements of a list are important and a list with variable number of elements is acceptable as a match, the extra elements can be dropped by using this syntax.

- If used outside pattern matching, curly braces act as an universal list splicing and flattening operator. If an argument is given to a function surrounded by curly braces and this argument is a list then it is treated as if every individual element of that list was provided in its place.

  If an argument in curly braces is not a list, then curly braces behave like the identity operator and return it unchanged.

  There can be multiple arguments inside one pair of curly braces or multiple curly-braced arguments given to a function. All of these arguments will be expanded in the way described.

  For example:

```
bind [f of [a b c d e f log [a b c d e f]]]
bind [args $[8 7 6]]

-- logs ‘9 8 7 6 5 4‘:
f[9 {args} {$[5 4]}]

-- or alternatively,
-- surrounding all arguments with curly braces
-- logs ‘9 8 7 6 5 4‘:
f[{9 args $[5 4]}]

-- curly braces can surround any argument
-- or any sequence of arguments,
-- regardless of position in the invocation list;
-- logs ‘9 8 7 6 [5, 4]‘:
f[{9 args} $[5 4]]
```

  This works similarly to the spread operator from ECMAScript 2015[30] or the splicing syntax `,@` used in Lisp's backquotes[18, Section 2.4.6], [13, Section 9.4], but is much more flexible. It can be used in every function or macro invocation, not just in backquotes. Also, if there is more that one list-argument that should be spliced, they can be grouped together inside curly braces and do not have to be individually "tagged".

  Curly braces can also be used as a nicer syntax for the fundamental function `apply`:

```
    bind [numbers $[1 2 3 4 5]]

    -- evaluates to 15:
    apply [sum numbers]

    -- also evaluates to 15:
    sum[{numbers}]
```

- They also serve as string interpolation notation. When a string is evaluated, all expressions surrounded by `{` and `}` that appear inside this string are evaluated and spliced into its value before the value is returned.

For example:

```
bind [name '|Bill]

-- logs 'Hello, Bill.':
log ['[Hello, {name}.]]
```

This gives us a very convenient notation for string interpolation, similar to e.g. template literals in JavaScript[31].

To escape curly braces inside a string we can double them or use the escape character \:

```
-- logs 'Hello, {name}.':
log ['[Hello, {{name}}.]]

-- also logs 'Hello, {name}.':
log ['[Hello, \{name\}.]]
```

There is also a special type of string – an HTML string, where interpolation notation is the reversed – double braces cause substitution, single braces do nothing:

```
bind [name '|Bill]

-- logs '<h1>Hello, Bill.</h1>':
log [html'[<h1>Hello, {{name}}.</h1>]]

-- logs '<h1>Hello, {name}.</h1>':
log [html'[<h1>Hello, {name}.</h1>]]
```

This is to enable embedding CSS and JavaScript code inside those strings, without having to constantly escape brace characters.

- Related to the above point, curly braces are used in `code'` strings that are returned by macros. They work similarly to curly braces in strings, except that the substituted values should be unevaluated expressions (syntax tree nodes). Also, code strings are never interpreted as streams of characters, and are always stored as syntax trees.

  This use of curly braces is very similar to Lisp's unquote syntax (`,`)[12, Section 1.3.8].

  The next section (2.12) provides examples and explanation of macros in Dual.

2.12 Macros

The experimental approach to Dual's design gave rise to a very interesting feature, which could be described as first-class just-in-time expanded macros.

Macros in Lisp-like languages are different than the macros provided by the C preprocessor or similar macro systems. They are much more powerful[63].

Essentially, macros are a way to transform code from one (usually much terser) form to another. They can extend the language's syntax, create new syntactical constructs or Domain-Specific Languages (DSLs)[4, Chapter 3].

Lisp-like macros are integrated with the language and operate on its code or, more precisely, syntax trees[4]. Such a macro can be described as a function that takes code as arguments and returns code as a result. This code is then expanded into the syntax tree, replacing the macro. A macro can perform arbitrary computation while it is evaluated, just like a function. Macros are written in the same language as the code they transform.

2.12.1 First-class

Unlike traditional Lisp macros, in Dual macros are first-class, because they are treated like any other value.

In order to support first-class runtime macros, a Lisp interpreter can be modified as follows[108][5]:

- Primitives are moved into the top-level environment[6]. They thus are no longer treated as special case by the `eval` function.
- A new primitive, `macro` is added, which is essentially equivalent to `lambda`, except that it produces macro values instead of function values.
- The `apply` function is now responsible for checking the type of an expression's operator, which can be a `primitive`, a `macro` or a normal expression. This determines whether the arguments should be evaluated before application.

This results in a simpler, more uniform and at the same time more powerful interpreter. A major advantage is that:

> Because of their first-class nature, first-class macros make it easy to add or simulate any degree of laziness[108].

A macro in Dual is defined with the `macro` primitive and bound to a name with the `bind` primitive:

```
-- defines an 'unless' macro,
-- which works like the 'if' primitive,
-- except that the provided condition is negated
bind [unless macro [condition body alternative
    code'[if [not[{condition}] {body} {alternative}]]
]]

bind [a 100]
```

[4]For brevity I will sometimes use the term "code" when I mean a syntax tree.

[5]Recall a brief description of an implementation of a Lisp interpreter from Section 1.2.

[6]The top-level environment contains the basic functions and values that are available to every program.

```
-- will log "a greater than 3":
unless [>[a 3]
    log|'[a less than or equal to 3]
    log|'[a greater than 3]
]
```

Because a macro is a first-class value, there is no need for a special primitive for defining macros, such as `defmacro` in Lisp[18, Section 3.8, Macro DEFMACRO].

2.12.2 Just-in-time

Macros in Dual are just-in-time expanded, because the expansion happens at runtime, when a macro is encountered and evaluated by the interpreter. There is no separate macro-expansion time.

For simple macros the expansion works as follows:

- A macro invocation is encountered by the interpreter.
- It is expanded into code by evaluating it.
- The node in the EST containing the macro invocation is permanently replaced by the expanded expression.
- The expanded expression is evaluated and its value is returned as the value of the invocation.
- Next time when the interpreter arrives at the same point in the EST, the macro will already be replaced by the expanded expression. Thus, the cost of macro-expansion is one-time.

Macros in Dual can also return other macro values. If a macro returns a macro value instead of code, this value is evaluated. If, in turn, the result of this evaluation is another macro value, this one is evaluated as well, and so on, until a code value is returned. It then is expanded as described above.

This feature nicely composes with Dual's variation of Lisp's syntax in terms readability, particularly by reducing the amount of adjacent closing brackets in the source code and introducing blocks without the need for explicit use of the `do` primitive.

As an example. the below listing presents a macro named `if*`, which defines a slightly different syntax for the `if` primitive. This syntax wraps the condition, consequent and alternative parts of the `if` in separate blocks delimited by `[]`. The condition is required to be an infix expression in the form `a operator b`. The consequent and alternative blocks take care of wrapping all expressions within them in `do` blocks. This makes it more convenient and less error-prone to write complex `if` expressions:

```
-- defines the ‘if*‘ macro
-- it returns a macro, which returns a macro,
-- which returns another macro
-- arguments of each of these macros
-- are then spliced in the appropriate places
-- in the code that creates the resulting ‘if‘ expression:
bind [if* macro [a op b
    macro [{then}
        macro [{else}
            code'[if [apply[{op} {a} {b}] do[{then}] do[{else}]]]
        ]
    ]
]]

-- the macro is used as follows:
if* [a < b][
    log ['[a is less than b]]
    a
][
    log ['[b is less than or equal to a]]
    b
]

-- the above expands to:
if [<[a b] do [
    log ['[a is less than b]]
    a
] do [
    log ['[b is less than or equal to a]]
    b
]]
```

Note that the macro gets rid of the explicit `do` expressions. It essentially defines a new language construct, which has the following template:

```
if* [--[condition: ] <value-1> <comparison-operator> <value-2>][
    -- consequent block:
    <expression-1>
    <expression-2>
    -- ...
    <expression-n>
][
    -- alternative block:
    <expression-1>
    <expression-2>
    -- ...
    <expression-n>
]
```

2.12.3 In combination with `|` and `!`

The combination of the macro system and the syntax sugar for zero and single argument functions (`|` and `!`) helps reduce the amount of bracketing characters even further.

For example, if we define a `match*` and `of*` macros as follows:

```
bind [match* macro [{args}
        macro [op
                code'[apply [{op args}]]
]]

bind [of* macro [{args}
    macro [{body}
        macro [alternative
            code'[of~ [{args} do[{body}] {alternative}]]
        ]
    ]
]]
```

Then the following expression:

```
bind [x 99]

-- will log "x is greater than one":
match*  [x]
| of* [<|0] [log|'[x is negative]]
| of*   [0] [log|'[x is zero]]
| of*   [1] [log|'[x is one]]
| log|'[x is greater than one]
```

Would be translated into the following:

```
bind [x 99]

-- will log "x is greater than one":
apply [
    of~ [<|0 log|'[x is negative]
        of~ [0 log|'[x is zero]
            of~ [1 log|'[x is one]
                log|'[x is greater than one]
            ]
        ]
    ]
    x
]
```

Notice that the `match*` macro does not use or need the native `match` construct at all.

The resulting syntax is somewhat similar to ML-style[55, Section Algebraic datatypes and pattern matching] languages. And yet it there is no complex grammar that defines this syntax. It is handled with a very simple parser.

This shows that a few simple, but general syntax rules and a powerful macro system, can be a very flexible tool for extending syntax.

Chapter 3

Dual's development environment

This chapter provides a design for Dual's development environment. This is not a formal specification. I try to provide an idea of how I imagine a good development environment should look like to maximize usability and be appealing to experienced as well as semi-experienced programmers.

These are general guidelines that may be applied to improve existing visual programming language systems in terms of usability.

A proof-of-concept prototype implementation that, to a degree, conforms to the design outlined here, is described in Chapter 4. In that implementation I focus on the critical features that are necessary to meet the goals of this thesis.

3.1 Overview

The overall design tries to merge features of modern code editors, such as Brackets[65], Visual Studio Code[79] or Atom[71] with visual programming language editors, such as the Blueprint Editor from Unreal Engine 4[93].

The environment is intended to work online, similarly to Integrated Development Environments such as Codeanywhere[68] or Cloud9[67]. It should also work as offline, like Scratch's offline editor[40].

3.2 Design goals

I set the following general goals that the designed development environment must meet:

- In terms of usability, it should be easily usable with minimal effort from the user to install it or set it up.
- The user interface and its idioms and metaphors[118] should be familiar for an experienced user and easy to learn for an inexperienced one.
- The editor should support the text and visual representation.

3.3 Requirements

In order to meet the above goals, I set down the design requirements outlined in this section.

3.3.1 Usability

The environment should be a web application. It should have minimal external dependencies.

It should be separated into the following components:

- A project manager, which should be capable of managing local as well as remote projects. The environment identifies projects by paths, which obviates the need for project files.

  For example, if user selects a path on her local file system, such as `/projects/my-project`, it will be registered as the most recently opened project and opened in the editor. All files contained in the `my-project` folder would be a part of the project.

  This is an idiom used in modern code editors, such as Brackets or Visual Studio Code. However, project files could be optional.

- A code editor, which itself consists of two parts:
  - A text editor with syntax highlighting, auto-completion, auto-indentation, etc. for Dual. The text editor should conform to the design outlined in Section 2.8.
  - A visual editor able to manipulate a visual representation of the EST with autostructuring. It should also conform to the design outlined in Section 2.8.

3.3.2 User interface

The general layout of the editor should conform to universally accepted (*de facto*) standards where possible. As illustrated in Figures 3.2, 3.3 and 3.1, different code edtiors feature a very similar layout. Below I aim to enumerate its general features.

The layout of Dual's environment should include the following:

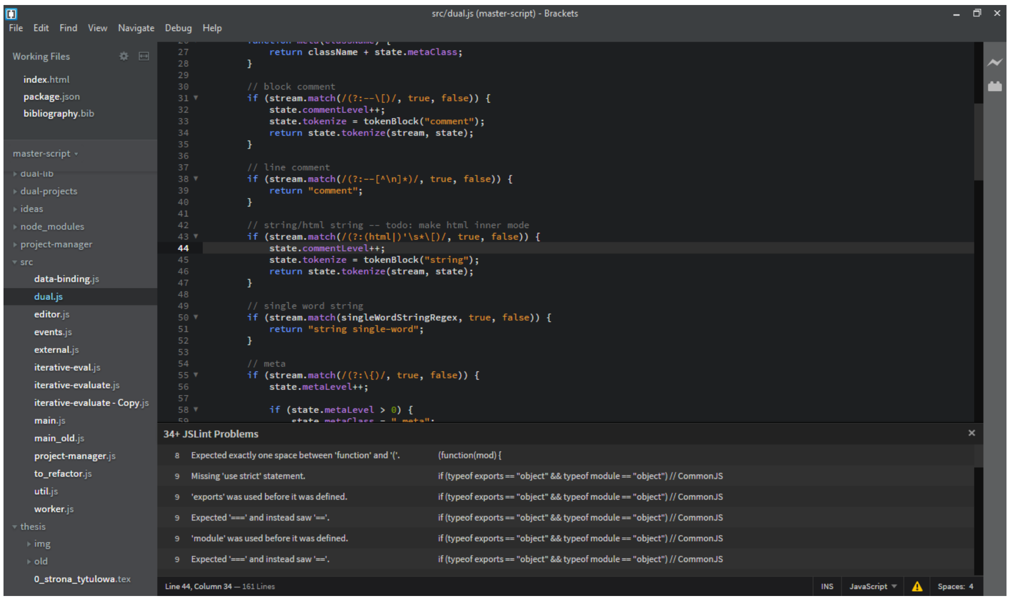
- A title bar, which contains the name of the file that is currently being edited (focused), a path to the file and the name of the editor.
- A menu bar at the very top of the window screen, which contains menus such as File, Edit, View, Help, each with appropriate options and submenus.
- A left panel with a tree view of all the files that belong to the currently open project.

- Next to the left panel, a main area, where the contents of the currently open file are displayed. There may be a split view functionality, where a few files can be displayed at once next to each other. There also may be a tab bar above a file's contents that contains buttons that allow the user to quickly switch between several open files.

- A bottom panel that can contain an interactive console, similar to the JavaScript console in web browsers. The panel may also display various status information.

- A status bar at the bottom of the screen. It may display short information about current position in a file, text encoding or indentation size.

- A text input that is an interface for a flexible global search facility, which can search through all editor options and menus, as well as the library of available programming language constructs: all primitives, basic functions and macros as well as all user-defined functions and macros. The search functionality should be available at all times. It should appear at the top of the screen and may be toggleable.



```
            return className + state.metaClass;
        }

        // block comment
        if (stream.match(/(?:--\[)/, true, false)) {
            state.commentLevel++;
            state.tokenize = tokenBlock("comment");
            return state.tokenize(stream, state);
        }

        // line comment
        if (stream.match(/(?:--[^\n]*)/, true, false)) {
            return "comment";
        }

        // string/html string -- todo: make html inner mode
        if (stream.match(/(?:(html|)'\s*\[)/, true, false)) {
            state.commentLevel++;
            state.tokenize = tokenBlock("string");
            return state.tokenize(stream, state);
        }

        // single word string
        if (stream.match(singleWordStringRegex, true, false)) {
            return "string single-word";
        }

        // meta
        if (stream.match(/(?:\{)/, true, false)) {
            state.metaLevel++;

            if (state.metaLevel > 0) {
```

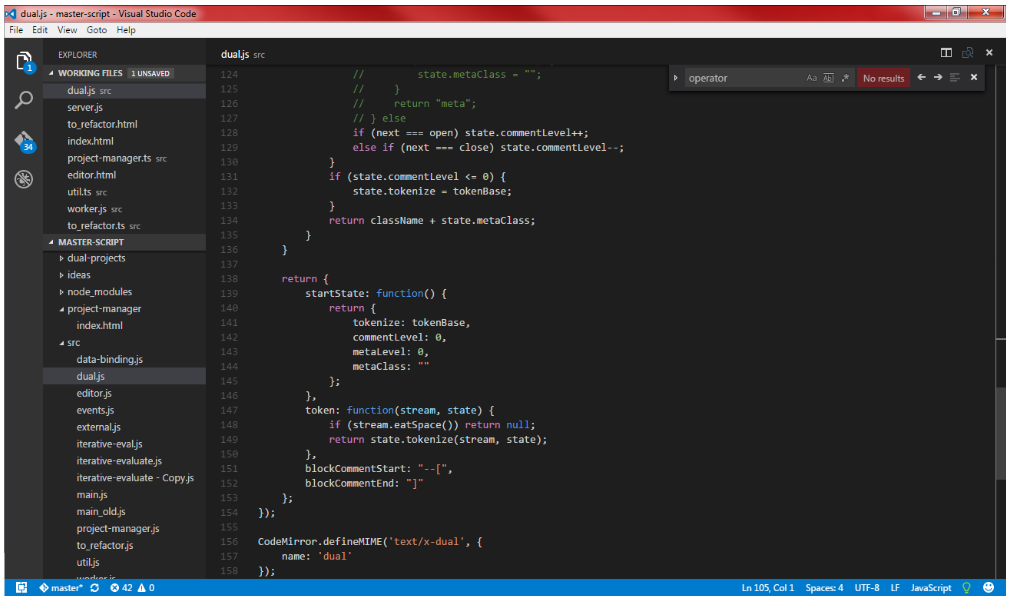
```
34+ JSLint Problems
8  Expected exactly one space between 'function' and '('.   (function(mod) {
9  Missing 'use strict' statement.                          if (typeof exports == "object" && typeof module == "object") // CommonJS
9  'exports' was used before it was defined.                if (typeof exports == "object" && typeof module == "object") // CommonJS
9  Expected '===' and instead saw '=='.                     if (typeof exports == "object" && typeof module == "object") // CommonJS
9  'module' was used before it was defined.                 if (typeof exports == "object" && typeof module == "object") // CommonJS
9  Expected '===' and instead saw '=='.                     if (typeof exports == "object" && typeof module == "object") // CommonJS
Line 44, Column 34 — 161 Lines
```

Figure 3.1: A screenshot from the Brackets editor

3.4 Text editor

The text editor should have all the standard capabilities of modern code editors, like the ones outlined in Section 1.2.2:

- Auto-indentation.

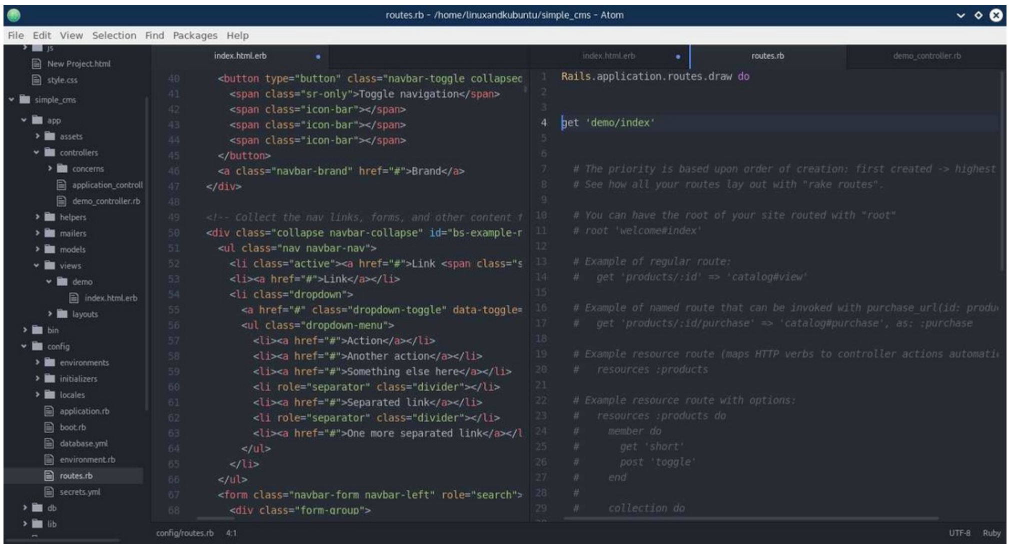


```
                //         state.metaClass = "";
                //     }
                //     return "meta";
                // } else
                if (next === open) state.commentLevel++;
                else if (next === close) state.commentLevel--;
            }
            if (state.commentLevel <= 0) {
                state.tokenize = tokenBase;
            }
            return className + state.metaClass;
        }
    }

    return {
        startState: function() {
            return {
                tokenize: tokenBase,
                commentLevel: 0,
                metaLevel: 0,
                metaClass: ""
            };
        },
        token: function(stream, state) {
            if (stream.eatSpace()) return null;
            return state.tokenize(stream, state);
        },
        blockCommentStart: "--[",
        blockCommentEnd: "]"
    };
});

CodeMirror.defineMIME('text/x-dual', {
    name: 'dual'
});
```

Figure 3.2: A screenshot from the Visual Studio Code editor



```
    <button type="button" class="navbar-toggle collapsec
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="#">Brand</a>
  </div>

  <!-- Collect the nav links, forms, and other content f
  <div class="collapse navbar-collapse" id="bs-example-r
    <ul class="nav navbar-nav">
      <li class="active"><a href="#">Link <span class="s
      <li><a href="#">Link</a></li>
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle=
        <ul class="dropdown-menu">
          <li><a href="#">Action</a></li>
          <li><a href="#">Another action</a></li>
          <li><a href="#">Something else here</a></li>
          <li role="separator" class="divider"></li>
          <li><a href="#">Separated link</a></li>
          <li role="separator" class="divider"></li>
          <li><a href="#">One more separated link</a></l
        </ul>
      </li>
    </ul>
    <form class="navbar-form navbar-left" role="search">
      <div class="form-group">
```

```
Rails.application.routes.draw do


get 'demo/index'


  # The priority is based upon order of creation: first created -> highest
  # See how all your routes lay out with "rake routes".

  # You can have the root of your site routed with "root"
  # root 'welcome#index'

  # Example of regular route:
  #   get 'products/:id' => 'catalog#view'

  # Example of named route that can be invoked with purchase_url(id: produ
  #   get 'products/:id/purchase' => 'catalog#purchase', as: :purchase

  # Example resource route (maps HTTP verbs to controller actions automati
  #   resources :products

  # Example resource route with options:
  #   resources :products do
  #     member do
  #       get 'short'
  #       post 'toggle'
  #     end
  #
  #     collection do
```

Figure 3.3: A screenshot from the Atom editor

- Syntax highlighting.
- Code folding. Any code between a matching pair of brackets should be collapsible.
- Code navigation. Any identifier or module path in the source code should be a link to an appropriate declaration or definition, if such is available.
- Context-sensitive auto-completion. This includes automatic pairing of brackets and correction of typing errors.

- Find and replace functionalities with support for regular expressions.
- The source text should be processed by the editor as a two dimensional grid of characters, with each row (line) and column numbered. Any rectangular area in this grid should be selectable.
- Related to the above: editing of multiple lines in parallel. If an area is selected that spans multiple rows, each row has its own text cursor.

All of the above features should be integrated with the language and make use of its parser and the syntax tree generated by it.

3.5 Visual representation and its editor

In order to address some of the most common criticisms of VPLs, outlined in Section 1.2.5 I set the following general design requirements for the visual representation:

- It should combine the familiar appearance of the "line-connected block-based" VPLs with the structure of "snap-together block-based" VPLs (see Section 1.2.3). The former family is exemplified by the Blueprints Visual Scripting system of Unreal Engine 4[94] and the latter by MIT Scratch[39, 53].
- It should be no harder to use than the text representation. Ideally a visual editor should add useful capabilities, without taking away these provided by text editors.
- Its appearance should be fully customizable.

Moreover, Dual's visual representation should satisfy all the requirements outlined in 2.8:

- It should be fully mappable to the EST. There should be a distinguishable and manipulable visual element for every EST node. Such an element should contain a reference to the node (and vice versa).
- There should be ways to perform the following actions with the visual editor:
  - Insert new nodes into EST.
  - Remove existing nodes from the EST.
  - Modify existing nodes in the EST.
  - Replace existing nodes and subtrees in the EST.
  - Move nodes and subtrees to different locations in the EST.
  - Select and manipulate multiple arbitrary nodes and subtrees in the EST.

  In short, the visual editor should provide means to perform the same abstract operations on the EST that are possible when editing text and possibly more.

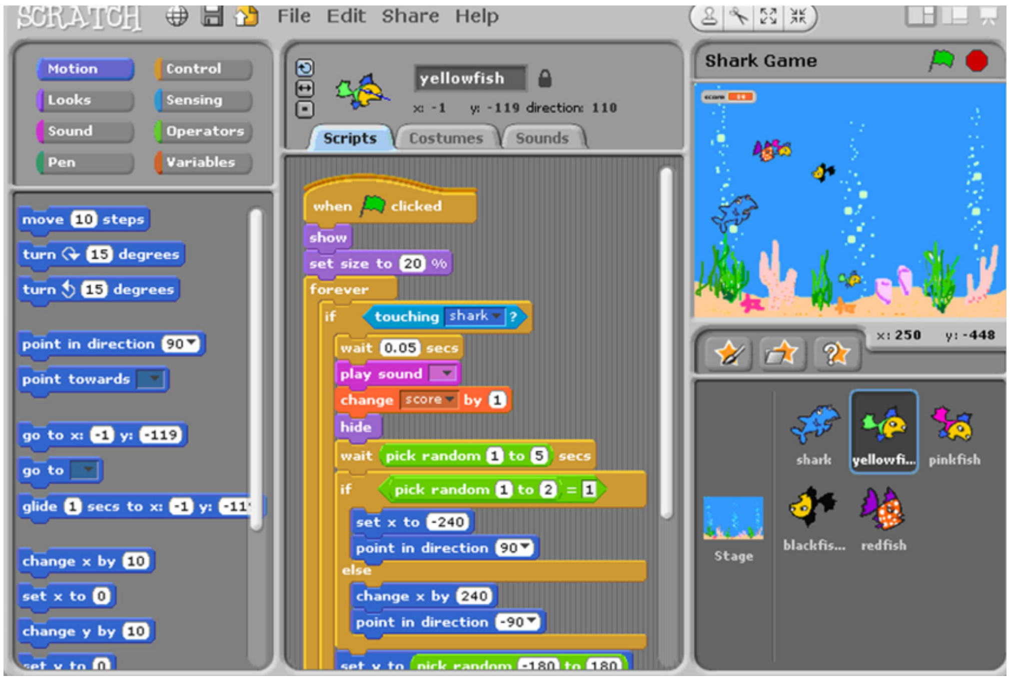
Some operations on raw text should also be reflected in the visual representation. For example the Find option. Searching through text should highlight the visual elements that correspond to the EST nodes that are connected to the text that contains the searched phrase. Regular expression based searching should also be possible. Replacing text with the Replace option should be reflected in the visual representation.

There should be a possibility to temporarily disable a representation by disconnecting it from the EST. Any changes made to other representations will then not be propagated to this representation. A disabled representation should become read-only until it is enabled. When it is enabled it should be updated to reflect the current state of the EST. At least one representation must be enabled at all times.

There should be a context-sensitive auto-completion feature and an easily accessible library of functions and primitives with documentation. User-defined functions should be added to the library and auto-completion database upon definition and removed from it when they are removed from the source code.

The library could look similarly to the one in Scratch, where the user can select puzzle pieces from several categories (Figure 3.4). The categories should be the names of modules, where each function is defined. Similarly, submodules should have their respective subcategories.

The auto-completion context menu could resemble the one from the UE4's Blueprint editor (Figure 3.5). If there is many auto-completion possibilities, they should be presented as a tree, with categories from the library as root nodes.



```
when clicked
show
set size to 20 %
forever
  if touching shark ?
    wait 0.05 secs
    play sound
    change score by 1
    hide
    wait pick random 1 to 5 secs
    if pick random 1 to 2 = 1
      set x to -240
      point in direction 90
    else
      change x by 240
      point in direction -90
    set y to pick random -180 to 180
```

Figure 3.4: MIT Scratch programming language editor; the left panel contains a library of language constructs. These are ordered into categories (selectable with buttons at the top of the panel); screenshot from[128]

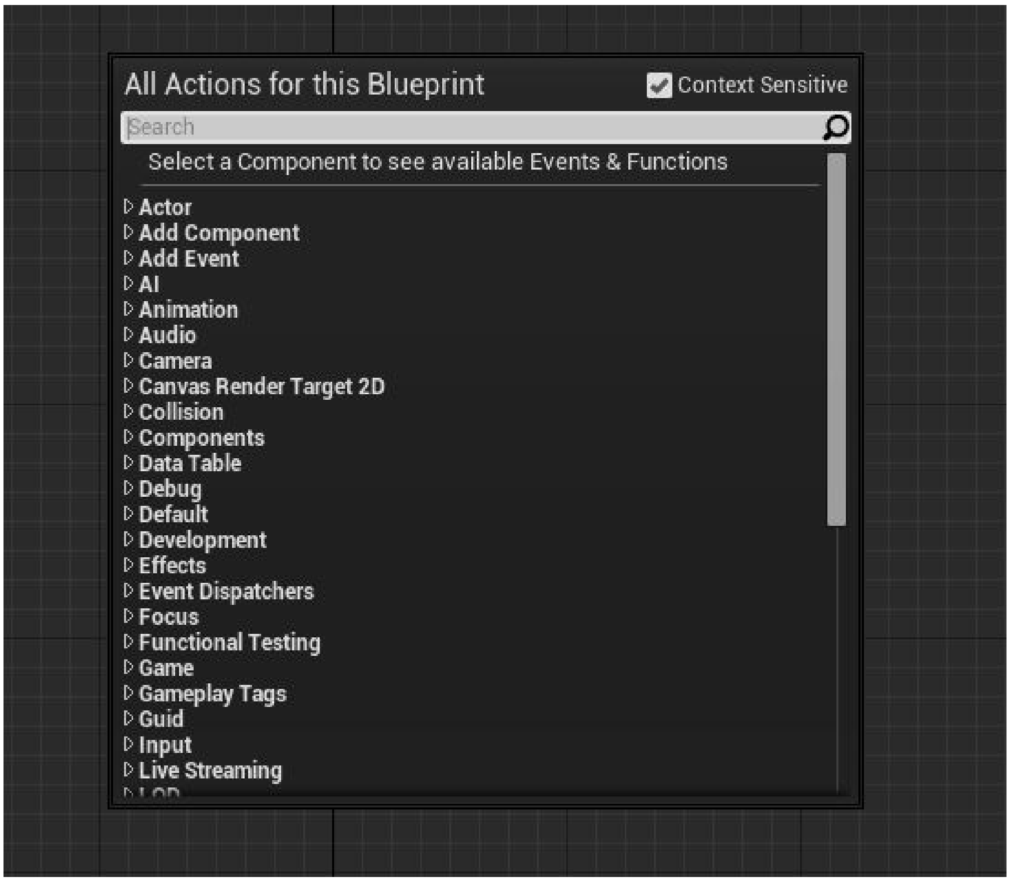


Figure 3.5: The context menu from the UE4's Blueprints editor; each item is a category, which can be expanded by clicking on it into subcategories, which eventually lead to individual nodes. Clicking on a node name inserts it into the program; screenshot from[?]

3.5.1 The design process

Figures 3.6, 3.7, and 3.8 show mock-ups that were produced when designing the visual representation.

The colored squares with letters inside are place-holders for icons. The user should be able to click on those icons and fold the blocks into a more compact form, hiding the names and excessive text. This should be possible at the level of individual blocks, whole subtrees or the entire program – similarly to code folding in text editors. This allows to have a big picture and general relationships between nodes always visible and at the same time gives an ability to focus on the details of the part at hand.

The design contains the following visual elements:

- Rectangular blocks, which represent expressions or individual nodes of the EST. Those in turn consist of:
  - A header, which contains an icon and the name of the expression's op-

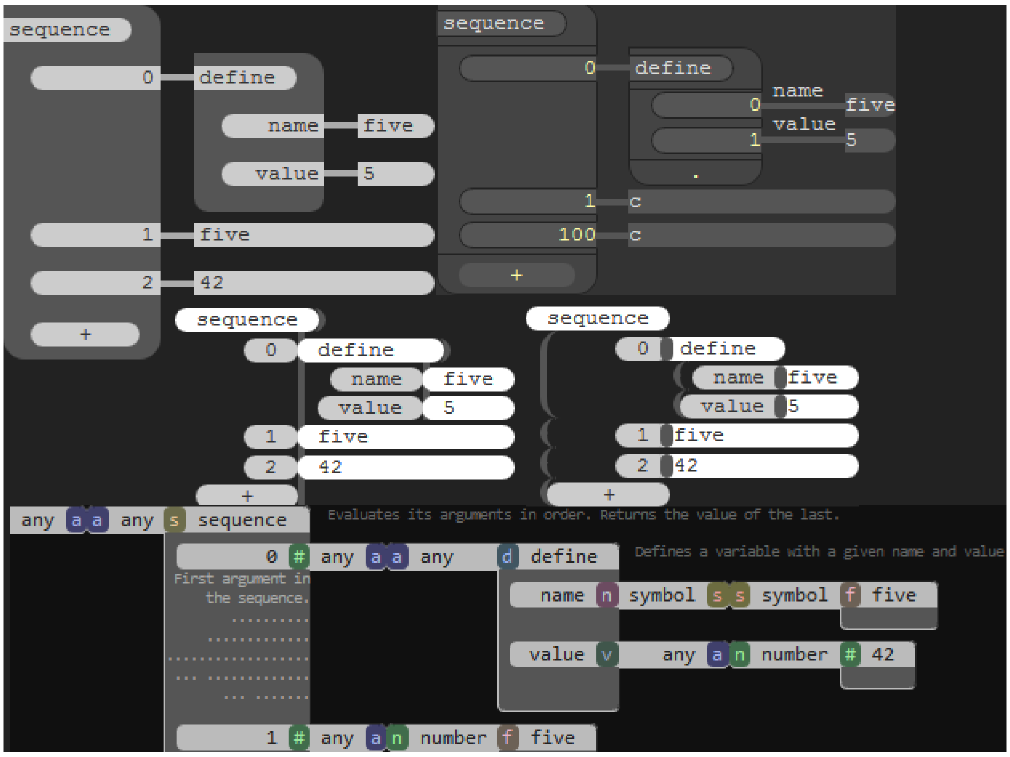
erator. Next to the header a documentation comment may be displayed.

  – Slots, which are the numbers or names of the arguments followed by an icon. Below these documentation comments may be displayed.
  – Additional buttons, which may be used for example to add more slots to variadic expressions.

- Connections between slots and blocks, which could also contain some useful annotations. The proposed design places type annotations there. These consist of the name of the type followed by an icon that represents this type. Connections actually have two parts: one extending from a slot, which in this case would contain the argument's type annotation, and one extending from a block header, which would contain a type annotation of the expression's return value.

The visibility of the documentation comments below the slots associated with the arguments could be toggleable by clicking on them. This should work on an individual, subtree or global scale, similarly to icons. This idiom of individual-subtree-global should be applied to all options, where it is sensible.

The user should have an easy way of configuring whether or not and what information should be displayed. By folding enough of the text elements into icons the visual representation could actually be made more compact than the text form.

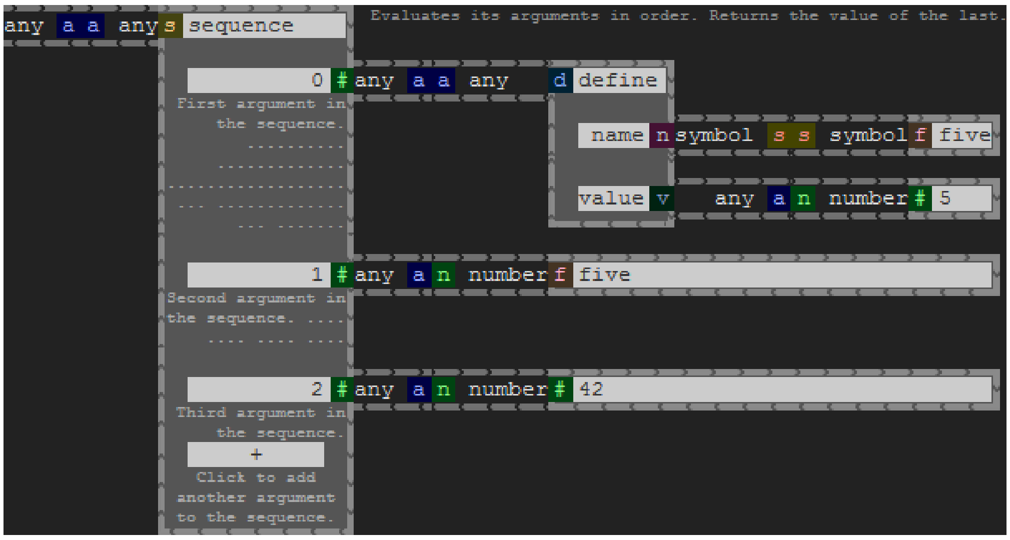


Evaluates its arguments in order. Returns the value of the last.

Defines a variable with a given name and value

First argument in the sequence.

Figure 3.6:


Evaluates its arguments in order. Returns the value of the last.

First argument in the sequence.

Second argument in the sequence.

Third argument in the sequence.

Click to add another argument to the sequence.
any a a any s sequence
0 # any a a any d define
name n symbol s s symbol f five
value v any a n number # 5
1 # any a n number f five
2 # any a n number # 42
+

Figure 3.7: This design has the interesting property of visually illustrating the program flow with arrows.

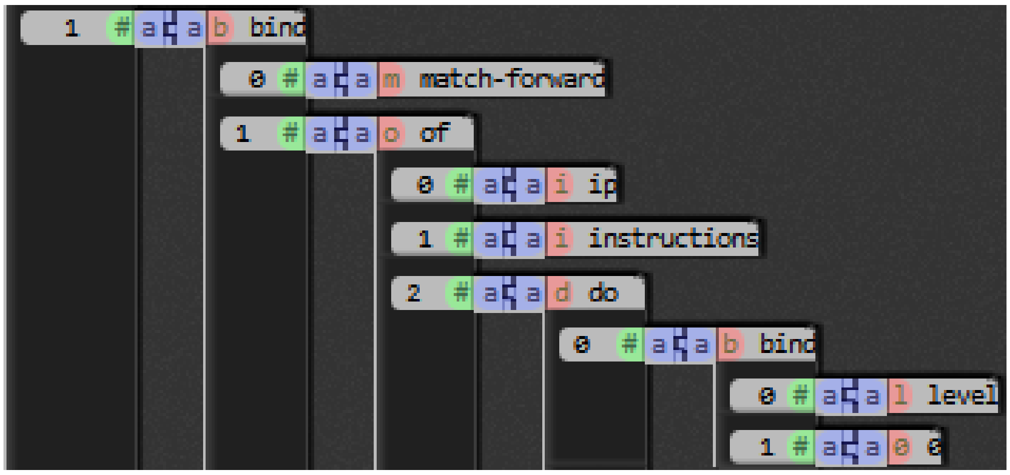


Figure 3.8:

3.6 Additional features

Both, the text and the visual editor should provide an interface for debugging programs. This entails the abilities to:

- Set breakpoints on individual lines of code (applicable only to the text editor).
- Set breakpoints on individual expressions (applicable to both).
- Pause program execution and step over, in and out of individual expressions.
- Inspect program state. It should be possible to easily access the values of individual expressions and possibly change them.

- Signal errors in a specific and readable manner.

Other features that aid debugging could also be useful. For example:

- The ability to record program execution and rewind it, inspecting individual steps.
- Support for exceptions.
- Support for remote debugging.
- Profiling facilities.

The editor should also provide the following additional features:

- Integration with version control systems and services that provide it, such as GitHub.
- Integration with text comparison and diff tools.
- Integration with linting tools.
- An Application Programming Interface for plugin creation.
- "Intelligent" mechanisms that display contextual hints and suggestions.

Chapter 4

Prototype implementation

This chapter describes how the essential parts of the designs outlined in previous chapters were fulfilled in practice by implementing a proof-of-concept interpreter and development environment.

4.1 Programming discipline

The prototype was implemented largely in the spirit of exploratory programming: "the kind where you decide what to write by writing it."[72].

This approach in combination with a dynamic and flexible language like JavaScript enables one to quickly transform ideas to working prototypes and shape them as one goes along. But the usefulness of this method is limited, as it may quickly produce fairly low-quality code, as it is not focused on future maintainability.

Most of the features of the prototype system are implemented as a proof-of-concept, were the main focus is making them work. Performance and other considerations are of low priority. Some features are more refined than others in order to fulfil the major goals of this thesis, one of which was to implement a working non-trivial application in the language.

4.2 The language

The language's parser and interpreter are implemented in JavaScript. The parser conforms to the grammar specification described in Section 2.2.1 with all of its extensions defined in Chapter 2. The parser emits events while processing every language construct. These events carry individual syntax tree nodes with information about the current position in the source string. The events are then captured by the environment to attach additional information to EST nodes, such as references to elements of the visual representation (DOM nodes), and objects that represent text fragments.

The prototype implementation of the language contains all the features described in Chapter 2, with the following exceptions:

- Macros are not implemented.[1]
- There are two primitives, which produce function values: `of` and `of-p`. The second has the same meaning as `of` described in Chapter 2. The first has the same meaning, except that it does not use pattern matching when binding names to arguments. This primitive requires that all the names must be words.
- Destructuring is implemented only for definitions (it works in `bind`) and not for assignments (it does not work in `mutate`). Pattern matching could easily be extended to mutation, although I have found it sufficient to be usable only in definitions and ended up not implementing it for assignments in the prototype.
- Comments are treated and attached to EST nodes as streams of characters. Nesting and balancing of brackets is taken into account in multi-line comments, but their tree-like structure is not preserved.
- Strings are not treated specially by the parser. They are stored and manipulated as syntax tree nodes, not as streams of characters. This means that the optimization described in Section 2.8.1 is not applied. The performance penalty is acceptable in the prototype implementation.
- The escape character \ has no special meaning. To substitute a special character in a string, the following built-in values are defined:

  ```
  (left-bracket) -- escapes "["
  (right-bracket) -- escapes "]"
  (left-brace) -- escapes "{"
  (right-brace) -- escapes "}"
  (pipe) -- escapes "|"
  (bang) -- escapes "!"
  ```

  So `'[(left-bracket)hello(right-bracket)]` would evaluate to: `"[hello]"`.

The interpreter is implemented naively. Its main part is the recursive `evaluate` function. The function accepts an `expression` and an `environment`. The expression can be a number literal, an identifier or an invocation. Each of these types is evaluated differently:

- If it is a number literal, it evaluates to its corresponding numerical value.
- If it is an identifier (a name), it is looked up in the environment and the corresponding value is returned. If it is not found in the environment, an error is thrown.

[1]In fact, they partially *are* implemented, but are not usable. For example, there is a `macro` primitive available, which produces macro values. But it should not be used, as these macro values are not treated specially by the interpreter, so using them will not have the desired effect.

- If the expression is an invocation then it is evaluated according to its operator:
  - If the operator is a string operator `'` or `html'`, then first any variable substitutions are performed and then the corresponding string value is returned.
  - If the operator is a primitive, then the expression's arguments are not evaluated and passed to this primitive, which may or may not evaluate them. The result of the primitive's application is returned.
  - If the operator is an expression, then it is evaluated. Then its arguments are evaluated. If any of those arguments is a substitution expression (i.e. it is wrapped in curly braces `{}`) then it is expanded before proceeding. When all of the arguments are evaluated, the operator is applied to them and the result of this application is returned.

4.3 The environment

The development environment's prototype implementation provides only the few critical features that are necessary to demonstrate how complete integration and interchangeability of the text and visual representations of the language can be achieved.

The prototype of the environment has elements of a web application in its implementation. However, it is prepared only to work offline, on the user's machine.

The system is implemented with minimal dependencies, so it can be easily installed and so that a greater level of integration can be achieved by having more control over every part of the system.

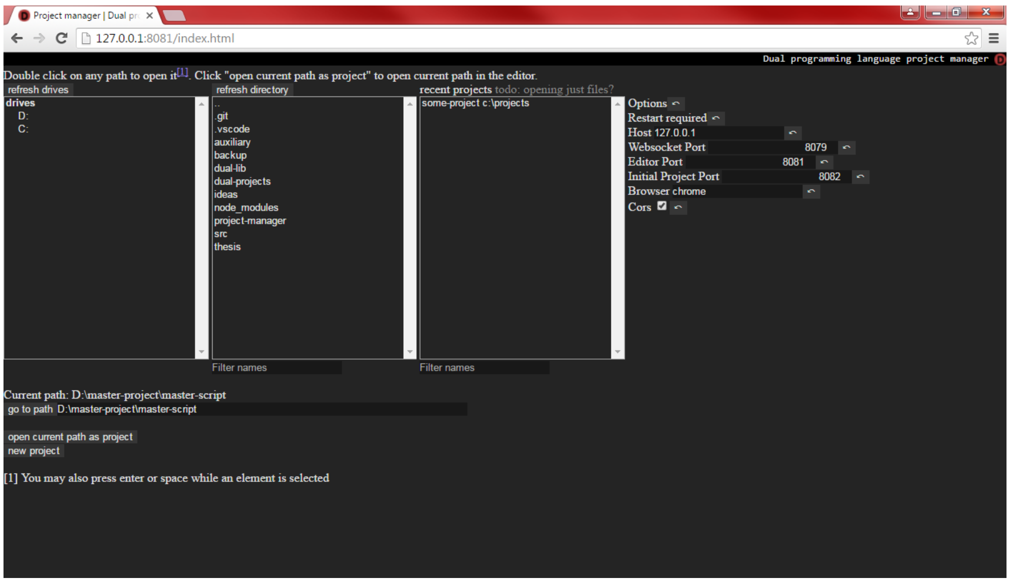
The only required dependencies for the basic functionalities of the prototype to work is a web browser and the CodeMirror library. An additional dependency is the Node.js environment – for running the stub of the project manager.

4.3.1 General architecture

The development environment's web-application-like architecture is reflected in its three main components:

- The server part, implemented in JavaScript on top of Node.js. This part's function is mainly to enable access to the user's file system, so that any local folder can be opened as a project. Modern web browsers restrict access to the local file system, as dictated by their security policy. The server part also handles persisting changes to files and configuration.

- The project manager part, which communicates directly with the server part. The connection is maintained over a WebSocket[35]. This part provides access to user's file system via a custom folder selection interface. Basic configuration of server communication, such as changing the address and ports is also


Project manager | Dual pr
127.0.0.1:8081/index.html
Dual programming language project manager
refresh drives
refresh directory
Filter names
Filter names
go to path
open current path as project
new project

Double click on any path to open it[1]. Click "open current path as project" to open current path in the editor.

drives
D:
C:

..
.git
.vscode
auxiliary
backup
dual-lib
dual-projects
ideas
node_modules
project-manager
src
thesis

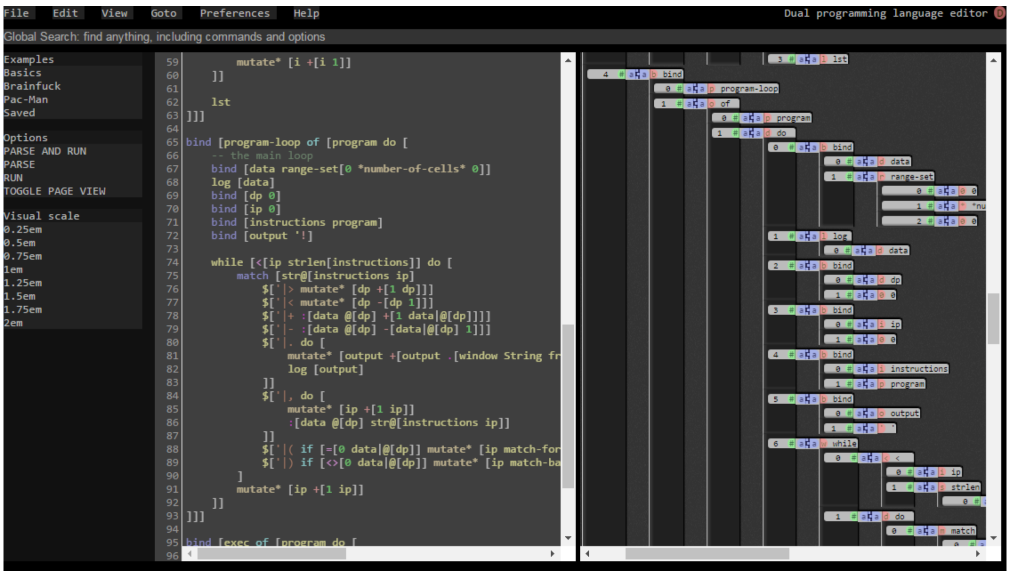
recent projects todo: opening just files?
some-project c:\projects

Options	
Restart required	
Host	127.0.0.1
Websocket Port	8079
Editor Port	8081
Initial Project Port	8082
Browser	chrome
Cors	☑

Current path: D:\master-project\master-script
go to path D:\master-project\master-script

[1] You may also press enter or space while an element is selected

Figure 4.1: Dual's project manager



```
        mutate* [i +[i 1]]
    ]]

    lst
]]]

bind [program-loop of [program do [
    -- the main loop
    bind [data range-set[0 *number-of-cells* 0]]
    log [data]
    bind [dp 0]
    bind [ip 0]
    bind [instructions program]
    bind [output '!]

    while [<[ip strlen[instructions]] do [
        match [str@[instructions ip]
            $['|> mutate* [dp +[1 dp]]]
            $['|< mutate* [dp -[dp 1]]]
            $['|+ :[data @[dp] +[1 data|@[dp]]]]
            $['|- :[data @[dp] -[data|@[dp] 1]]]
            $['|. do [
                mutate* [output +[output .[window String fr
                log [output]
            ]]
            $['|, do [
                mutate* [ip +[1 ip]]
                :[data @[dp] str@[instructions ip]]
            ]]
            $['|( if [=[0 data|@[dp]] mutate* [ip match-for
            $['|) if [<>[0 data|@[dp]] mutate* [ip match-ba
        ]
        mutate* [ip +[1 ip]]
    ]]
]]]

bind [exec of [program do [
```

Figure 4.2: Dual's editor

possible. Once a project is selected, the user may open it in the editor part. Figure 4.1 demonstrates the interface of the project manager.

- The editor part, which is the main component and can function as a standalone application. It can communicate with the server indirectly, through the `localStorage` mechanism[37].

The project manager and the editor, which can be considered the front-end parts of the system are designed to be Single-Page Applications[54]. The project manager exchanges JSON messages with the server through a WebSocket. This is used for updating the view with dynamic data. In order to facilitate the manipulation of the HTML structure of the page, which is the main application's view, I implemented a very simple web application framework, which binds the data from the server with the data on the client and the Document Object Model[3, Chapter 13].

Figure 4.2 shows an overview of the editor prototype's window. The basic layout is modeled after the aforementioned code editors. At the top of the window is the menu bar, below it a mockup of a global search input (not implemented). The left panel contains basic controls for selecting examples, invoking the parser and interpreter, toggling application view and adjusting the scale of the visual representation.

The browser's JavaScript console is used as the standard output. There is no built-in console.

The following options are implemented in the prototype:

- Available from the menu bar:
  - File->Save, which saves the current content of the text editor to a file named `save.dual` in editor's root directory. This only works if the server-side part of the environment is running. Otherwise the source will be saved only to browser's internal storage.
  - In the Edit menu: Undo, Redo, Cut, Copy, Paste and Select All options are supported. Note that by default web browsers restrict the access to the user's clipboard, so for Copy and Paste the standard key shortcuts should be used (Ctrl-C, Ctrl-V). All other conventional keyboard shortcuts are also supported, thanks to the CodeMirror library.
- Available from the left panel:
  - The options in the Examples submenu cause a corresponding source file to be loaded into the editor. This is for demonstration for the purposes of this thesis.
  - The Options submenu allows the user to invoke the parser and the interpreter separately or in combination as well as toggling between the "page" (also known as "application") and visual editor views. The application view contains an embedded web page (iframe), which can be manipulated by a Dual application. This is used to display the game view in the Pac-Man clone example.
  - The Visual scale submenu changes the size of the blocks in the visual editor. This demonstrates how manipulating one CSS property influences the rendering of the visual representation.

Some options have descriptive captions available that appear when the mouse cursor hovers over them.

4.3.2 Text editor

The text editor is built on top of the CodeMirror framework[73]. This provides all basic features of a text editor, such as automatic indentation, syntax highlighting (a custom syntax highlighting mode for Dual is defined), line numbering, block selection or parallel editing of multiple lines.

The text editor is integrated with the environment. The fragments of text corresponding to EST nodes in the text representation are tracked by CodeMirror's TextMarker objects. These facilitate tracking and propagating any changes to and from this representation, as well as highlighting of the currently focused expression.

If a position of the text cursor in or the contents of the source change, a fragment of text corresponding to the appropriate EST node is highlighted. Also the corresponding subtree in the visual editor is highlighted. It works also in the other direction – when a node in the visual editor is selected, it is highlighted along with the corresponding text fragment.

This demonstrates the core functionality of the system: it is "aware" at all times of currently focused meaningful part of the code, corresponding to an EST node. This is reflected in both representations associated with the EST.

Because every node in the EST is linked in both directions with a corresponding abstract element in a representation, any change to the element can be reflected in the node and, through the EST, in all other associated representations. This makes the system accurate and fast, as every change happens in an isolated context, which does not have to be reestablished every time a modification is made.

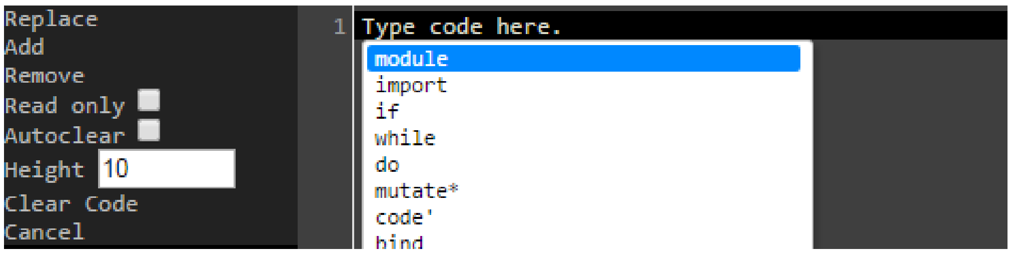


Figure 4.3: Visual editor's context menu

4.4 Visual editor and representation

The visual representation is implemented in a in terms of a DOM tree, which mirrors the EST: every EST node has a corresponding set of DOM nodes. Thanks to this, any actions performed on the DOM can be tracked through the standard browser-implemented interface. This is done by attaching `click` event handlers to relevant nodes. Such an event triggers the following:

- A corresponding EST node is "focused" by the system.
- The visual node is highlighted.

- A context menu appears similar to that depicted in 4.3.

The context menu depicted in Figure 4.3 has all the basic options for manipulating the visual representation. These perform their corresponding action on the currently focused node and propagate it to the text representation. The options are named Replace, Add, and Remove.

The Remove option simply removes the selected node and its subtree from the DOM, the EST, as well as the associated text fragment.

The Add and Replace options make use of the small text-editor area next to the context menu. It contains a predefined list of names of some of the possible nodes that can be inserted. Selecting any of the names causes a template for the new node – in the form of an editable code snippet – to be inserted into the text-editor area. Such a template can be quickly adjusted by the user before inserting.

The predefined list of names is a stub implementation of the context menu feature described in Section 3.5

The user may also type in raw code into the text box, without selecting any templates. After entering the code and selecting the appropriate option, the text is parsed, transformed into TextMarker, EST, and DOM representations. Then all the versions of the fragment are inserted in appropriate places.

The list of possible nodes displayed along with the context menu is implemented in terms of a simple auto-complete functionality on top of CodeMirror. Every item in the auto-complete list is associated with a fragment of code, which is basically a signature of the corresponding function. User-defined functions could be easily automatically added to this list by extracting their signatures from definitions.

The visual representation is composed purely out of HTML and CSS, which makes its appearance fully customizable.

Chapter 5

Case study

In order to examine and demonstrate the capabilities of the prototype implementation, I implemented a Pac-Man clone in Dual. This chapter describes the results of this test.

Implementation such non-trivial application allowed me to test the language design and establish which features are the most useful in practice.

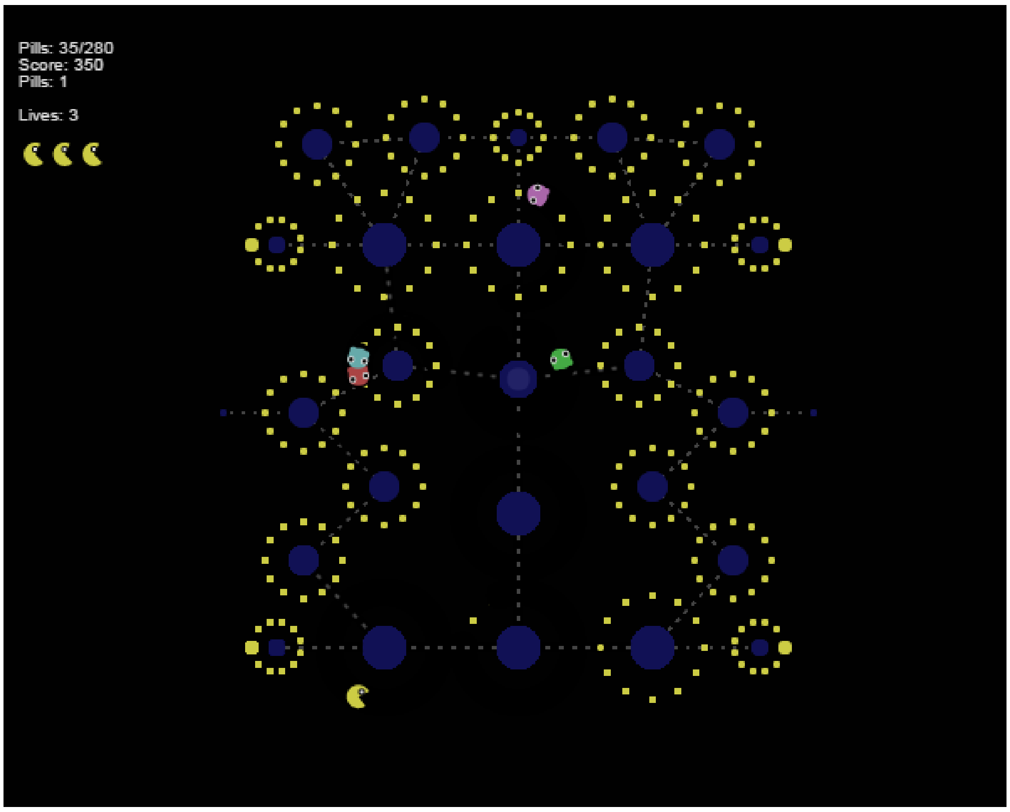


Figure 5.1: A screenshot from the game

5.1 The game

The implementation described here is a port of my earlier clone of the game, which was written in Links[70], a functional language.

The mechanics of the game are slightly changed compared to the original. The maze of corridors is replaced by a maze of "planets", which are connected to each other. Pac-Man and the ghosts are in constant motion, as they orbit the planets, because there are no walls and 90-degree turns, which can stop them. Figure 5.1 shows a screen shot from the gameplay.

5.1.1 Main loop

A typical game loop in a modern JavaScript game[20] relies on the `requestAnimationFrame` method[38]. This method takes a single argument, which is a function callback. This function is invoked by the browser before it repaints the contents of the window. Thus, it allows the game rendering to be synchronized with the browser.

The callback invoked by `requestAnimationFrame` receives a timestamp, which specifies the time of the repaint event. Ideally and under the most common circumstances this happens 60 times a second.

I implemented the game loop in Dual as follows[1]:

```
-- [1] the amount of milliseconds between game state updates:
bind [tick-length 50]

-- the main loop function:
--[
    arguments:
        -- current game state:
        game-state
        -- time of the last game state update:
        last-tick
        -- the time of the current invocation of the loop:
        current-time
]
bind [main-loop of [
    game-state
    last-tick
    current-time

    do [
        -- [2] schedule the next iteration of the loop
        -- using requestAnimationFrame:
        async [
            .[window requestAnimationFrame]
            of [next-time
                main-loop[
                    game-state
                    last-tick
```

[1] Only the essential parts are shown in the listing, for the complete implementation, see the `src/pillman.dual` file in the DVD attached to this thesis.

```
                next-time
            ]
        ]
    ]

    -- the timestamp of the tick after the last tick:
    bind [next-tick +[last-tick tick-length]]

    -- counts how many state updates should be
    -- performed in this iteration:
    bind [tick-count 0]

    -- if the current time is past the timestamp
    -- of the next tick, the above counter should be
    -- incremented; possibly more than by one,
    -- if the difference is a multiply of tick-length:
    if [>[current-time next-tick] do [
        bind [time-since-tick -[current-time last-tick]]
        mutate [
            tick-count
            .[math floor][
                /[time-since-tick tick-length]
             ]
        ]
    ] _]

    -- [3] perform the calculated amount of game state updates
        :
    bind [i 0]
    while [<[i tick-count] do [
        -- keep track of the time of the last tick:
        mutate [last-tick +[last-tick tick-length]]

        -- [4] invoke the function that updates the game-state
            :
        mutate [
            game-state
            main-game-logic[game-state input-queue]
        ]

        mutate [i +[i 1]]
    ]]

    -- [5] if there were game state updates, redraw game
        screen;
    -- else do nothing:
    if [>[tick-count 0]
        draw[
            game-state
            current-time
            .[window performance now]!
        ]
        _
    ]
    ]
]]
```

The places in the above listing that require closer examination are marked with comments that contain a number in brackets, such as `[1]`. These comments are placed above the first line of the important fragment.

Each item on the below numbered list refers to the appropriately marked piece of code, as in 1. refers to `[1]`, 2. refers to `[2]` and so on:

1. The global variable `tick-length` is set to 50. It determines the frequency of game updates. 50 milliseconds translates to 20 Hz or 20 FPS (Frames Per Second). It turns out that the value of this variable very significantly determines the performance of the application and influences it in interesting ways, described later in ths chapter.

2. The first thing that happens in the main loop is a call to the `async` primitive, which schedules the next call to the main loop to happen in the next animation frame. This asynchronous recursive invocation ensures that the simulation runs continuously. The last argument to the next main loop invocation is `next-time`. It contains the timestamp of the next frame, provided by `requestAnimationFrame`.

3. The while loop performs game state updates. The longer the period since the last call to main loop, the more times it will execute.

4. The game state updates are performed by the `main-game-logic` function.

5. Drawing happens only if there were game state updates in the current main loop iteration. Otherwise a frame is essentially dropped.

5.2 Performance

I encountered significant problems with performance of the application. In order to avoid unnecessary inflation of the volume of this thesis, I will not go into details, but outline the main points that explain the issues.

The game showed problems with performance fairly early into the implementation. Initially I set the value of the `tick-length` variable to 16.67 (milliseconds), to achieve a frame rate of 60 FPS. When I it with the `draw` function redrawing every yellow dot on the screen every frame, it rendered the application unresponsive.

In order to make the game run smoothly, I had to introduce an optimization where only the changed portion of the screen was redrawn each frame. I also had to decrease the value of `tick-length`, lowering the expected frame rate. This turned out to be different for different browsers.

For Chrome the value of `tick-length`, which allowed the game to run smoothly was 50. This means that an updates were happening every 50 milliseconds or at a frame rate of 20 FPS. Lower values caused the frame rate to destabilize and drop significantly on average.

In Firefox the application ran smoothly when `tick-length` was set to 75. This gives a frame rate of about 13 FPS. The performance in this browser was overall significantly worse than in Chrome.

I investigated the reason for the slow performance and the difference between Chrome and Firefox browsers with their built-in profilers. This showed that the slowdowns were caused by JavaScript's garbage collector, which was cleaning up large amounts of memory very often, thus pausing the execution.

The reason for this lies in the simple architecture of the interpreter. It works by recursively invoking a function that evaluates an expression. First the topmost expression (the root of the syntax tree) is evaluated. Then its each of its arguments, which are themselves expressions that have other expressions as arguments. This quickly results in a big tree of recursive calls on the call stack.

While the function that evaluates an expression is running, JavaScript's event loop is blocked, preventing any events from being handled and any other code from running.

Moreover, each of the stack frames contains references to various data, even though most of it is not relevant. But since it is referenced, the garbage collector cannot release the memory that the data takes up.

The collection can happen only when a whole syntax tree is evaluated and the stack is again empty.

The more calls in a single frame to the function that updates the game state, the more garbage to collect. This results in the collection pauses being longer. Because of that the frame rate drops.

The different patterns between Firefox and Chrome are caused by different garbage collection strategies[**? ?**].

The general pattern seems to be that in Chrome the collections are more frequent, regular and predictable than in Firefox, which results in better performance, higher frame rate and smoother appearance of the game.

5.3 Possible improvements

To fix the described performance issues, the language's interpreter should be implemented in a way that takes into account the characteristics of the host language's environment. In case of JavaScript some of these are:

- The event-loop-based concurrency model. In JavaScript an iteration of event loop should run as fast as possible to achieve the best performance.

  An interpreter should not block the loop by recursive evaluation. The evaluation function should instead be interruptible. This could be achieved in multiple ways. One, which I experimented with[2], is transforming the function into an iterative version by emulating the call stack. In this implemen-

[2]The file `dist/iterative-evaluate.js` contains an implementation of `evaluate` function that was used in earlier prototypes of the interpreter.

tation, the function contains a variable that holds the stack frames for its own recursive invocations.

The function can thus pause evaluation between calls to itself, which should be asynchronous (similarly to the `main-loop` described in Section **??**) and not recursive. This releases the event loop.

Explicit control of the call stack allows trivial implementation of debugging facilities, since evaluation can now be paused at any time.

- Automatic memory management with a garbage collector. An interpreter should create as little garbage as possible, should be free of memory leaks and possibly should manage memory "manually", using object pools or similar patterns.

An entirely different approach to improve performance would be compilation of the language to bytecode, straight to JavaScript or even to asm.js, a highly-optimizable low-level subset of JavaScript[**?**].

Interruptable eval

Continuation-passing style State machine Anyway, explicit stack

Additional benefits: can pause and debug the application, step through can record the state and rewind

5.4 Conclusion

Dynamic languages with a garbage collector allow a programmer to write code without worrying much about memory management or other low level considerations. But when it comes to performance and robustness this approach shows its downsides very quickly.

The performance issues that I have encountered when implementing the Pac-Man clone, which are described in this chapter are very much related to the characteristics of the JavaScript environment. Notably the event loop and the garbage collector, which has different implementations with varying performance profiles across browsers. This shows in the performance differences when comparing different web browsers.

An interpreted language implemented without caring about low-level mechanisms, such as memory management is not suitable for writing non-trivial, performance-intensive applications, such as simulations or computer games.

This case study shows one of the disadvantages of exploratory programming.

Chapter 6

Comparisons to other VPLs

This chapter compares Dual to the Blueprints Visual Scripting system of Unreal Engine 4 and to MIT Scratch to better illustrate the improvements that the presented design provides.

6.1 VPLs: scripting languages

All VPLs are either DSLs or scripting languages. There is no general purpose VPL[? ?].

General characteristics of a scripting language are[? ? ? ?]:

- It is flexible and in terms of being able to perform
- It is usually dynamically typed
- It has a library of basic functions
- Because of the above, is suitable for rapid prototyping
- It complements a non-scripting language

6.2 MIT Scratch

Scratch is an educational programming language. Although it is not explicitly called a scripting language, it has characteristics of one and uses the term "script" to refer to the programs created in it[?].

6.2.1 Issues

Being intended for educational purposes, Scratch has a very limited set of features. Arguably too limited. Some of its main shortcomings are:

- No support for first-class or higher order functions.

- Limited file I/O.
- Implemented in ActionScript, which limits its portability and usability.
- Does not support complex data structures. Only one-dimensional arrays, known as "lists" are supported.
- String manipulation capabilites are limited.
- Limited support for object-orientation.
- No text representation.

All in all, Scratch is not really usable outside of education. Which is not a problem in itself, since it is designed as strictly for that. But still, it is far from perfect even in this application.

The validity of this statement is reflected in the fact that there exists a less popular derivative, called Snap!, which does partially address Scratch's issues. It adds first class procedures, first class lists, and "first class truly object oriented sprites with prototyping inheritance, and nestable sprites"[53, Section Features and derivatives].

6.3 Blueprints Visual Scripting system

The Blueprints Visual Scripting system is a part of Unreal Engine 4, the commercial game engine. As its name implies, it is also intended for *scripting*. Its purpose is to complement C++, which is the implementation language of the engine and is used for all other purposes, such as extending it.

6.3.1 Issues

Some of the main disadvantages of the Blueprints system are[**?** **?** 94**?**]:

- Blueprint scripts consume more memory and are slower than C++ programs.
- Blueprint scripts are not portable.
- The visual editor does not support automatic structuring. It is hard to manage complex scripts.
- There is no support for first-class or higher-order functions.
- There is no usable text representation.
- The type system is static. This is not a disadvantage in itself, but it does negatively impact the aspects of complementing C++ (which also has a static type system) and flexibility.

- Basic functions are missing from the "standard library". E.g. there is no sorting function available out of the box[?]. The user has to either implement basic functions herself, which can result in partial or broken implementations or use external libraries[?].
- Version control is very difficult. Because blueprints are stored in binary format without a text representation, they cannot be automatically merged or compared by standard tools. Dedicated tools exist, but are more limited, harder to use.

Overall, as a major part of a commercial game engine, the Blueprints system is significantly lacking. Programming in Blueprints can often feel rigid and cumbersome. It does a poor job at complementing C++, being similarly static and rigid. It can be rather described as intersecting C++'s feature set, with some improvements, but significantly poorer performance. Basic functionality is missing and external libraries are needed to fix that. Because of all this reasons it is not very suitable for rapid prototyping. There is a lot of room for improvement.

6.4 In comparison to Dual

In comparison with the above languages, Dual features the following:

- It is highly usable, portable and is intended to be open-source.
- It has a highly integrated text representation interchangeable with the visual representation. This fixes all problems related to tools that operate only on text, such as version control or comparison and diffing tools.
- It is dynamically typed – its type system relies on JavaScript's type system.
- It is highly expressive and extensible, having support for first-class and higher-order functions. The designed first-class JIT macros are much more powerful than the macro system in the Blueprints system.
- The visual representation in Dual can be fully customized with CSS. This feature can be combined with the Lisp-quality expressive power of the language for example to better articulate the semantics of a DSL.

Chapter 7

Summary and conclusions

The language and its development environment presented in this thesis is by no means a complete design. It should be viewed as a snapshot from a design process that I intend to continue in the future.

However the effort invested in the project was sufficient to achieve the general goals, listed in Chapter 0. This is reflected in:

- The core language, which is Turing-complete and thus capable of implementing any algorithm[1]. A simple Brainfuck interpreter is included as an example program (see Appendix **??**) to demonstrate this[96]. Moreover, the language is based on Lisp, inheriting a lot of its expressive power and extensibility.
- The above is also demonstrated in implementing a non-trivial application, which is the clone of Pac-Man described in Chapter 5. This exercise also showed significant flaws in the prototype related to performance and thus helped set directions for future improvement.
- Design and implementation of a mechanism, which enables the direct correspondence and interchangeability of the visual and text representations. In principle any number of representations could be associated.
- Implementation of the prototype of the language's development environment with integrated text and visual editors. This demonstrates practically the main ideas outlined in Chapter 3.

Visual programming systems that provide the ability to work with text and visual representations were developed in the past[101]. But none of them seems to have a similar degree of integration as the system presented here.

[1]In the same sense as JavaScript or C. No existing language is Turing-complete in the absolute sense, because of physical hardware limitations.

I am convinced that some of the ideas presented here are a valuable contribution in the area of (visual) programming language design and development and can be of use for current and and future designers in improving the existing as well as creating new, better[2] visual language systems.

I learned that programming language design is a tremendous and heroic task, especially if the language being designed is intended to be of any real-world use. Designing and implementing such a language absolutely from scratch, while introducing useful innovation cannot be done within the time limits of research for a thesis, unless perhaps by an experienced language designer. But such experience has to be gained somehow this was an excellent opportunity.

[2] While doing this research for this thesis I found a VPL project named Luna, that may be built around a similar idea of highly integrated text and visual representations. The project's website claims (emphasis mine): [81]:

> Luna is the *world's first* programming language featuring two exchangeable representations: textual and visual, which you can freely switch at any time.

However the project is in "private alpha" stage – it is not publicly available as of July 2016. Nonetheless this may be taken as an indication that the ideas presented in this research are a step on a path to better future VPL systems.

Bibliography

Books and articles

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996. Also available online: `https://mitpress.mit.edu/sicp/`.

[2] E.W. Dijkstra. Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60. Technical report, Stichting Mathematisch Centrum, 1961.

[3] Marijn Haverbeke. *Eloquent JavaScript.* No Starch Press, 2nd edition, December 2014. Also available online: `http://eloquentjavascript.net/`.

[4] Doug Hoyte. *Let Over Lambda.* Lulu.com, 2008.

[5] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, March 1993. Also available here: `http://worrydream.com/EarlyHistoryOfSmalltalk/`.

[6] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3(4):214–220, April 1960.

[7] Peter Seibel. *Practical Common Lisp.* 2005.

[8] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

[9] Various Wikibooks users. F# Programming. `https://en.wikibooks.org/wiki/F_Sharp_Programming`. A Wikibooks project: `https://en.wikibooks.org/wiki/Main_Page`.

[10] Philip Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Notices*, 22(3):83–94, March 1987. Available online: `https://www.cs.kent.ac.uk/people/staff/dat/miranda/wadler87.pdf`.

Documentations, standards and specifications

[11] Russell Allen et al. Self Handbook for Self 4.5.0 documentation. http://handbook.selflanguage.org/4.5/, January 2014.

[12] Matthew Flatt and PLT. The Racket Reference. https://docs.racket-lang.org/reference. “[D]efines the core Racket language and describes its most prominent libraries.”.

[13] Free Software Foundation, Inc. Emacs Lisp. https://www.gnu.org/software/emacs/manual/html_node/elisp/index.html. The latest version of the GNU Emacs Lisp Reference Manual.

[14] Ecma International. ECMAScript® 2017 Language Specification. https://tc39.github.io/ecma262/.

[15] Ecma International. ECMAScript® 2015 Language Specification. http://www.ecma-international.org/ecma-262/6.0/, June 2015.

[16] Ecma International. ECMAScript® 2016 Language Specification. http://www.ecma-international.org/ecma-262/7.0/index.html, June 2016.

[17] ISO/IEC/IEEE. ISO/IEC/IEEE 60559:2011. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469, July 2011. “[S]pecifies formats and methods for floating-point arithmetic in computer systems.”.

[18] LispWorks Ltd. Common Lisp HyperSpec (TM). http://clhs.lisp.se/Front/index.htm. “[O]nline version of the ANSI Common Lisp Standard[.]”.

[19] Alex Shinn, John Cowan, Arthur A. Gleckler, and et al. The Revised^7 Report on the Algorithmic Language Scheme. trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs.pdf, July 2013. The latest version of the de facto standard for the Scheme programming language.

Mozilla Developer Network

The following sources are articles from Mozilla Developer Network (https://developer.mozilla.org.):

[20] Mozilla Developer Network and individual contributors. Anatomy of a video game. https://developer.mozilla.org/en-US/docs/Games/Anatomy. An article from Mozilla Developer Network (https://developer.mozilla.org).

[21] Mozilla Developer Network and individual contributors. Array. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array. An article from Mozilla Developer Network (https://developer.mozilla.org).

[22] Mozilla Developer Network and individual contributors. Comparison operators. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators. An article from Mozilla Developer Network (https://developer.mozilla.org).

[23] Mozilla Developer Network and individual contributors. Concurrency model and Event Loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. An article from Mozilla Developer Network (https://developer.mozilla.org).

[24] Mozilla Developer Network and individual contributors. Console.log(). https://developer.mozilla.org/en-US/docs/Web/API/Console/log. An article from Mozilla Developer Network (https://developer.mozilla.org).

[25] Mozilla Developer Network and individual contributors. Destructuring assignment. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment. An article from Mozilla Developer Network (https://developer.mozilla.org).

[26] Mozilla Developer Network and individual contributors. JavaScript data types and data structures. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures. An article from Mozilla Developer Network (https://developer.mozilla.org).

[27] Mozilla Developer Network and individual contributors. Object.assign(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign. An article from Mozilla Developer Network (https://developer.mozilla.org).

[28] Mozilla Developer Network and individual contributors. Property accessors. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors. An article from Mozilla Developer Network (https://developer.mozilla.org).

[29] Mozilla Developer Network and individual contributors. Rest parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters. An article from Mozilla Developer Network (https://developer.mozilla.org).

[30] Mozilla Developer Network and individual contributors. Spread operator. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator. An article from Mozilla Developer Network (https://developer.mozilla.org).

[31] Mozilla Developer Network and individual contributors. Template literals. https://developer.mozilla.org/en-US/docs/Web/JavaScript/

Reference/Template_literals. An article from Mozilla Developer Network (https://developer.mozilla.org).

[32] Mozilla Developer Network and individual contributors. typeof. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof. An article from Mozilla Developer Network (https://developer.mozilla.org).

[33] Mozilla Developer Network and individual contributors. undefined. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined. An article from Mozilla Developer Network (https://developer.mozilla.org).

[34] Mozilla Developer Network and individual contributors. var. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var. An article from Mozilla Developer Network (https://developer.mozilla.org).

[35] Mozilla Developer Network and individual contributors. WebSockets. https://developer.mozilla.org/en-US/docs/WebSockets. An article from Mozilla Developer Network (https://developer.mozilla.org).

[36] Mozilla Developer Network and individual contributors. Window. https://developer.mozilla.org/en-US/docs/Web/API/Window. An article from Mozilla Developer Network (https://developer.mozilla.org).

[37] Mozilla Developer Network and individual contributors. Window.localStorage. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. An article from Mozilla Developer Network (https://developer.mozilla.org).

[38] Mozilla Developer Network and individual contributors. window.requestAnimationFrame(). https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame. An article from Mozilla Developer Network (https://developer.mozilla.org).

Wikis

[39] Scratch Wiki. Scratch. https://wiki.scratch.mit.edu/wiki/Scratch. Description of the language from the Scratch Wiki. See also the language's homepage: https://scratch.mit.edu/.

[40] Scratch Wiki. Scratch 2.0 Offline Editor. https://wiki.scratch.mit.edu/wiki/Scratch_2.0_Offline_Editor. From the Scratch Wiki: https://wiki.scratch.mit.edu/wiki/.

[41] W3C Wiki. Open Web Platform. https://www.w3.org/wiki/Open_Web_Platform. "The Open Web Platform is the collection of open (royalty-free) technologies which enables the Web.".

[42] Wikipedia, the free encyclopedia. Comparison of JavaScript-based source code editors. https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_source_code_editors. Wikipedia comparison article.

[43] Wikipedia, the free encyclopedia. Computer programming in the punched card era. https://en.wikipedia.org/wiki/Computer_programming_in_the_punched_card_era. Wikipedia article about programming in the punched card era.

[44] Wikipedia, the free encyclopedia. Homoiconicity. https://en.wikipedia.org/wiki/Homoiconicity. Wikipedia definition of homoiconicity.

[45] Wikipedia, the free encyclopedia. JSDoc. https://en.wikipedia.org/wiki/JSDoc. Wikipedia definition of JSDoc.

[46] Wikipedia, the free encyclopedia. LabVIEW. https://en.wikipedia.org/wiki/LabVIEW. A Wikipedia article.

[47] Wikipedia, the free encyclopedia. Lego Mindstorms. https://en.wikipedia.org/wiki/Lego_Mindstorms. Wikipedia article about Lego Mindstorms.

[48] Wikipedia, the free encyclopedia. Lisp (programming language). https://en.wikipedia.org/wiki/Lisp_(programming_language). Wikipedia definition of Lisp.

[49] Wikipedia, the free encyclopedia. List of C-family programming languages. https://en.wikipedia.org/wiki/List_of_C-family_programming_languages. A list from Wikipedia.

[50] Wikipedia, the free encyclopedia. List of Unreal Engine games. https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games. "[A] list of notable games using a version of the Unreal Engine." From Wikipedia.

[51] Wikipedia, the free encyclopedia. Logo (programming language). https://en.wikipedia.org/wiki/Logo_(programming_language). Wikipedia article about the Logo programming language.

[52] Wikipedia, the free encyclopedia. Pattern matching. https://en.wikipedia.org/wiki/Pattern_matching. Wikipedia definition of pattern matching.

[53] Wikipedia, the free encyclopedia. Scratch (programming language). https://en.wikipedia.org/wiki/Scratch_(programming_language). Wikipedia definition of Scratch.

[54] Wikipedia, the free encyclopedia. Single-page application. https://en.wikipedia.org/wiki/Single-page_application. Wikipedia definition of Single-page application.

[55] Wikipedia, the free encyclopedia. Standard ML. https://en.wikipedia.org/wiki/Standard_ML. Wikipedia definition of Standard ML.

[56] Wikipedia, the free encyclopedia. String interpolation. https://en.wikipedia.org/wiki/String_interpolation. Wikipedia definition of string interpolation.

[57] Wikipedia, the free encyclopedia. Unparser. https://en.wikipedia.org/wiki/Unparser. Wikipedia definition of an unparser.

[58] Wikipedia, the free encyclopedia. Variadic function. https://en.wikipedia.org/wiki/Variadic_function. Wikipedia definition of a variadic function.

[59] Wikipedia, the free encyclopedia. Visual programming language. https://en.wikipedia.org/wiki/Visual_programming_language. Wikipedia definition of a visual programming language.

[60] WikiWikiWeb. Definition Of Homoiconic. http://c2.com/cgi/wiki?DefinitionOfHomoiconic. An entry from WikiWikiWeb http://c2.com/cgi/wiki/FrontPage.

[61] WikiWikiWeb. Eval Apply. http://c2.com/cgi/wiki?EvalApply. An entry from WikiWikiWeb http://c2.com/cgi/wiki/FrontPage.

[62] WikiWikiWeb. Lisp Is Too Powerful. http://c2.com/cgi/wiki?LispIsTooPowerful. An entry from WikiWikiWeb: http://c2.com/cgi/wiki.

[63] WikiWikiWeb. Lisp Macro. http://c2.com/cgi/wiki?LispMacro. An entry from WikiWikiWeb http://c2.com/cgi/wiki/FrontPage.

[64] WikiWikiWeb. Lost Ina Seaof Parentheses. http://c2.com/cgi/wiki?LostInaSeaofParentheses. An entry from WikiWikiWeb http://c2.com/cgi/wiki/FrontPage.

Homepages

[65] Adobe and Brackets' community. Brackets - A modern, open source code editor that understands web design. http://brackets.io/. Brackets website.

[66] Edwin Brady and Idris' community. Idris | A Language with Dependent Types. `http://www.idris-lang.org/`. Homepage of the Idris programming language.

[67] Cloud9 IDE, Inc. Cloud9 - Your development environment, in the cloud. `https://c9.io/`. Cloud9 webpage.

[68] Codeanywhere, Inc. Codeanywhere · Cross Platform Cloud IDE. `https://codeanywhere.com/`. Codeanywhere webpage.

[69] Node.js Foundation. Node.js. `https://nodejs.org`. Node.js website. "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.".

[70] Simon Fowler, Sam Lindley, Garrett Morris, Philip Wadler, et al. Links: Linking Theory to Practice for the Web. `http://groups.inf.ed.ac.uk/links/`. Links programming language website.

[71] GitHub. Atom. `https://atom.io/`. Atom website. Atom is "[a] hackable text editor for the 21st Century".

[72] Paul Graham and Robert Morris. Arc Forum | Arc. `http://arclanguage.org/`. Arc programming language website.

[73] Marijn Haverbeke and CodeMirror's community. CodeMirror. `http://codemirror.net/`. CodeMirror website. "CodeMirror is a versatile text editor implemented in JavaScript for the browser.".

[74] Facebook Inc. Flow | A static type checker for JavaScript. `https://flowtype.org/`. Flow webpage.

[75] Ecma International. TC39 - ECMAScript. `http://www.ecma-international.org/memento/TC39.htm`. Technical Committee 39 webpage at Ecma International.

[76] Ecma International. Welcome to Ecma International. `http://www.ecma-international.org/`. Ecma International webpage.

[77] Ecma International and Technical Committee 39. tc39/ecma262: Status, process, and documents for ECMA262. `https://github.com/tc39/ecma262`. Official ECMAScript's GitHub repository.

[78] Microsoft. TypeScript - JavaScript that scales. `https://www.typescriptlang.org/`. TypeScript webpage.

[79] Microsoft. Visual Studio Code - Code Editing. Redefined. `https://code.visualstudio.com/`. Visual Studio Code website.

[80] Mojang. minecraft.net - Home. `https://minecraft.net`. Minecraft website.

[81] New Byte Order. Luna. Visual and textual functional programming language. http://www.luna-lang.org/. Luna programming language website.

[82] PLT et al. The Racket Language. https://racket-lang.org/. Racket programming language website. For detailed authorship information see https://racket-lang.org/people.html.

[83] The LEGO Group. Home - LEGO® MINDSTORMS® - LEGO.com - Mindstorms LEGO.com. http://mindstorms.lego.com. LEGO Mindstorms website.

[84] Various. About - Steel Bank Common Lisp. http://www.sbcl.org/. Steel Bank Common Lisp programming language website. For detailed copyright information see http://www.sbcl.org/history.html.

Other

[85] Scott W. Ambler. Software Modeling on Plain Old Whiteboards (POWs). http://agilemodeling.com/essays/whiteboardModeling.htm. From Agile Modeling website: http://agilemodeling.com/.

[86] Donnie Berkholz. Programming languages ranked by expressiveness. http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/, March 2013. A blog post by a RedMonk (A "developer focused industry analyst firm.") analyst.

[87] Craig Bicknell and Chris Oakes. Mozilla Stomps Ahead Under AOL. https://web.archive.org/web/20140603235609/http://archive.wired.com/techbiz/media/news/1998/11/16466, November 1998. An archived Wired (http://www.wired.com/) blog post.

[88] Daniel Bower. Visual Programming vs Text Programming. https://bowerstudios.com/node/742. An article from author's personal website: https://bowerstudios.com/.

[89] S. D. Bragg and C. G. Driskill. Diagrammatic-graphical programming languages and dod-std-2167a. In *AUTOTESTCON '94. IEEE Systems Readiness Technology Conference. 'Cost Effective Support Into the Next Century', Conference Proceedings.*, pages 211–220, Sep 1994.

[90] Jim Bumgardner. The Origins of Mindstorms. http://wayback.archive.org/web/20131221044013/http://www.wired.com/geekdad/2007/03/the_origins_of_/, March 2007. An archived Wired (http://www.wired.com/) blog post.

[91] Margaret M. Burnett. Visual Language Research Bibliography. `http://web.engr.oregonstate.edu/~burnett/vpl.html`. “This page is a structured bibliography of papers pertaining to visual language (VL) research.”. From Oregon State University.

[92] Rémi Dehouck. The maturity of visual programming. `http://www.craft.ai/blog/the-maturity-of-visual-programming/`, September 2015. A blog post.

[93] Epic Games, Inc. Blueprint Editor Reference. `https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Editor/`. From Unreal Engine 4 Documentation.

[94] Epic Games, Inc. Blueprints Visual Scripting. `https://docs.unrealengine.com/latest/INT/Engine/Blueprints/`. From Unreal Engine 4 Documentation.

[95] Martin Exner. Visual Programming Language – Infograph and Introduction | Constructing Kids. `https://constructingkids.com/2013/05/15/vpl/`. From the author’s personal blog: `https://constructingkids.com/`.

[96] Frans Faase. BF is Turing-complete. `http://www.iwriteiam.nl/Ha_bf_Turing.html`. An article from author’s personal website.

[97] Logo Foundation. Logo History. `http://el.media.mit.edu/logo-foundation/what_is_logo/history.html`. From Logo Foundation website: `http://el.media.mit.edu/logo-foundation/`.

[98] Frank da Cruz. IBM Punch Cards. `http://www.columbia.edu/cu/computinghistory/cards.html`. From Columbia University Computing History: `http://www.columbia.edu/cu/computinghistory/index.html`.

[99] Jacques Guyot. BNF rules of LISP. `http://cui.unige.ch/db-research/Enseignement/analyseinfo/LISP/BNFlisp.html`. A BNF formulation of Lisp syntax.

[100] Jim Hamerly, Tom Paquin, and Susan Walton. The Story of Mozilla. `http://www.oreilly.com/openbook/opensources/book/netrev.html`, January 1999. From “Open Sources: Voices from the Open Source Revolution”.

[101] Eric Hosick. Visual Programming Languages - Snapshots. `http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/`, 2014.

[102] Nicholas H.Tollervey. Lisp Concise and Simple. `http://ntoll.org/article/lisp-concise-and-simple`, March 2013. An article from the author’s personal website: `http://ntoll.org/`.

[103] MIT Media Laboratory. Cricket Logo for GoGo board. http://learning.media.mit.edu/projects/gogo/gogo22/cricket_logo.html.

[104] George Leontiev. https://twitter.com/folone/status/494017847585415168. A twitter message with a quote from Edwin Brady.

[105] Barry Margolin. Re: Lisp BNF available? http://www.cs.cmu.edu/Groups//AI/util/lang/lisp/doc/notes/lisp_bnf.txt. An archived message from comp.lang.lisp discussion group.

[106] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. http://www-formal.stanford.edu/jmc/recursive/recursive.html. A paper.

[107] John McCarthy. History of Lisp. http://www-formal.stanford.edu/jmc/history/lisp/lisp.html, February 1979. A draft.

[108] Matt Might. First-class (run-time) macros and meta-circular evaluation. http://matt.might.net/articles/metacircular-evaluation-and-first-class-run-time-macros/.

[109] Christian Nutt. Epic's Tim Sweeney lays out the case for Unreal Engine 4. http://www.gamasutra.com/view/news/213647/Epics_Tim_Sweeney_lays_out_the_case_for_Unreal_Engine_4.php, March 2014. An article from the Gamasutra website.

[110] Bill Orcutt. billorcutt/lily: Lily was a browser-based, visual programming environment written in JavaScript. the project is inactive. https://github.com/billorcutt/lily. A GitHub repository.

[111] Python Software Foundation. The Python Tutorial. https://docs.python.org/3/tutorial/. From Python 3 official documentation: https://docs.python.org/3/index.html.

[112] Axel Rauschmayer. How numbers are encoded in JavaScript. http://www.2ality.com/2012/04/number-encoding.html. An article from author's personal blog http://www.2ality.com.

[113] Guinness World Records. Most successful videogame engine. http://www.guinnessworldrecords.com/world-records/most-successful-game-engine. From Guinness World Records webpage. Record as of 16 July 2014.

[114] Tiago Simões. Visual Programming Is Unbelievable... Here's Why We Don't Believe In It. http://www.outsystems.com/blog/2015/03/visual-programming-is-unbelievable.html. An article from OutSystems blog: https://www.outsystems.com/blog/.

[115] Gerhard Sprung. history of visual programming languages v0.1. https://gsprung.wordpress.com/2010/07/21/history-of-visual-programming-languages-v0-1/, July 2010. From the author's personal blog: https://gsprung.wordpress.com/.

[116] The Type Theory Podcast. Episode 2: Edwin Brady on Idris | The Type Theory podcast. http://typetheorypodcast.com/2014/09/episode-2-edwin-brady-on-idris/.

[117] Various. What are the advantages and disadvantages of visual programming languages compared to regular programming languages? https://www.quora.com/What-are-the-advantages-and-disadvantages-of-visual-programming-languages-c An question at Quora https://www.quora.com.

[118] WAI. Why UX Designers Should Use Idioms Rather Than Metaphors. https://medium.com/@weareignition/why-ux-designers-should-use-idioms-rather-than-metaphors-f0e4718f4960.

[119] WHATWG. The Web platform: Browser technologies. https://platform.html5.org/. A list of browser technologies that are the components of the platform with links to their specifications.

[120] Wolfram. Working with String Patterns. https://reference.wolfram.com/language/tutorial/WorkingWithStringPatterns.html. An article from Wolfram Language Documentation http://reference.wolfram.com/language/.

Rankings, benchmarks and statistics

[121] Pierre Carbonnelle. PYPL PopularitY of Programming Language index. http://pypl.github.io/PYPL.html. A ranking of programming languages by popularity. "[C]reated by analyzing how often language tutorials are searched on Google.".

[122] Andrie de Vries. The most popular programming languages on StackOverflow | R-bloggers. http://www.r-bloggers.com/the-most-popular-programming-languages-on-stackoverflow/, July 2015. An R-bloggers (http://www.r-bloggers.com/) blog post which includes charts illustrating JavaScript's popularity on StackOverflow between 2008 and 2015.

[123] Miniwatts Marketing Group. World Internet Users Statistics and 2015 World Population Stats. http://www.internetworldstats.com/stats.htm.

[124] Stephen O'Grady. The RedMonk Programming Language Rankings: January 2016 – tecosystems. http://redmonk.com/sogrady/2016/02/19/

language-rankings-1-16/. A ranking of programming languages by popularity. "[C]correlates language discussion (Stack Overflow) and usage (GitHub)[.]".

[125] Stack Overflow. Stack Overflow Developer Survey 2016 Results. http://stackoverflow.com/research/developer-survey-2016. Stack Overflow's annual developer survey. "[T]he most comprehensive developer survey ever conducted.".

[126] Martin Rinehart. The Briefest Genealogy of Programming Languages. http://www.martinrinehart.com/pages/genealogy-programming-languages.html.

[127] TIOBE software BV. TIOBE Index | Tiobe - The Software Quality Company. http://www.tiobe.com/tiobe_index. A ranking of programming languages by popularity. Based on "the number of search engine results for queries containing the name of the language" (https://en.wikipedia.org/wiki/TIOBE_index).

Figure sources

[128] Anonymous. http://mypad.northampton.ac.uk/12406702/files/2013/05/Screen-Shot-2013-05-02-at-23.19.19-1s0qp26.png. A screenshot from the MIT Scratch environment. From https://mypad.northampton.ac.uk/12406702/2013/01/17/computer-programming-scratch/.

[129] Unreal Engine 4 Documentation. https://docs.unrealengine.com/latest/images/Engine/Blueprints/HowTo/BPHT_6/GetScore.jpg. A screenshot from Blueprints Visual Scripting system from Unreal Engine 4's official documentation: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html.

[130] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_piet_01.gif. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[131] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_lily_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[132] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_minecraft_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[133] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_webdesigner_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[134] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_appmaker_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[135] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_stroycode_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[136] Eric Hosick. http://blog.interfacevision.com/assets/img/posts/example_visual_language_lava_01.png. A screenshot from Interface Vision's blog post "Visual Programming Languages - Snapshots": http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. Individual sources are linked there.

[137] Tycho Luyben. http://d2o7bfz2il9cb7.cloudfront.net/main-qimg-a2e1e13841b01982fcb2ddcda2f958e9. From a Quora answer: https://www.quora.com/What-are-the-advantages-and-disadvantages-of-visual-programming-languages-c
answer/Tycho-Luyben.

Acronyms

API Application Programming Interface. 54

AST Abstract Syntax Tree. 10, 31, 32

BNF Backus–Naur Form. 22

DOM Document Object Model. 5, 55, 59, 60

DSL Domain-Specific Language. 40, 69, 71, 97

EST Enhanced Syntax Tree. 21, 31, 32, 51, 55, 56, 60

FPS Frames Per Second. 66, 67

GUI Graphical User Interface. 12

IDE Integrated Development Environment. 45

JIT Just-In-Time. 71

OOP Object-Oriented Programming. 6

PL Programming Language. 2

SPA Single-Page Application. 58

UE4 Unreal Engine 4. 70

UML Unified Modeling Language. 14

VPL Visual Programming Language. 2, 13, 16–20

Appendix A

DVD

The attached DVD contains the following directories:

dist – a runnable version of the prototype of the editor described in this thesis as well as all associated applications; also contains source files of all the applications

doc – electronic version of this thesis in PDF format and a presentation from diploma seminar.

ext – Node.js installer. Node.js is required to run the server-side of the application

src – only the source files of the applications developed in Dual

A.1 Running the prototype

It is assumed that you have a modern web browser compatibile with Firefox[1] 47 or Chrome[2] 51 – these were used in developing and testing the application. The source code is written using some ECMAScript2015 features, so it will not work on older browsers. In order to run the version distributed with this thesis, follow these steps:

1. If you want to run the server-side part of the application (it will work without it):
   (a) If you don't have Node.js already, install the latest "Current" version from the official distribution channel (`https://nodejs.org`), or use your operating system's package manager. If running 64-bit Windows, you may also use the installer from the DVD attatched to this thesis (`ext` folder). It was downloaded from `https://nodejs.org/dist/v6.2.2/`.
   (b) Open the `dist` folder in the command line.
   (c) By default, the server-side part of the application is configured to open `chrome` as the web browser that will handle the client-side. If you want to change that, edit the file `server-options.json` and change the `"browser"` property to a command that will open a different browser of your choice – e.g. `"firefox"`. Save the file.
   (d) Run the command `node server.js`. Before doing that you may optionally update all dependencies to the latest versions by running `npm install`.
   (e) By default the server-side part is configured to run on `127.0.0.1` and uses ports in the range 8079-8082, specified in the `server-options.json` file. Make sure these are available. If not, you may change the defaults again by editing the file.

[1]`https://www.mozilla.org/firefox`
[2]`https://www.google.com/chrome/browser/desktop/`

   (f) The project manager view should open in your web browser. You can change the same configuration options as in `server-options.json` here (under “Options”).
   (g) Click the button “open current path as project” at the bottom.
   (h) See 3

2. Alternatively, if you want to just open the editor, open the `editor.html` file from the `dist` folder.

3. A new tab should open in the browser with the editor view. You can start using it as described in Chapter 3.

Appendix B

Design discussion

This appendix contains some ideas that are being designed for the future versions of the Dual programming language.

B.1 Comments

B.1.1 Built-in documentation comments

In principle multi-line comments could be implemented simply with the syntax analyzer checking the operator of the expression being parsed, and if it is -, treating such expression as a comment. The fact that this expression was already parsed and transformed into a structural tree-like form could be taken advantage of while generating documentation from comments. For example we could define a following Domain-Specific Language[1] for documentation:

```
--[
    the below is a documentation comment
    followed by the documented piece of code:

    --[[
        Calculates the circumference of the Circle.

        override!
        deprecated!

        this [circle]

        -- The circumference of the circle:
        return [number]
    --]]

    define [calculate-circumference procedure [
        mul[2 math.pi this.radius]
    ]]
]
```

[1]Inspired by [45])

B.1.2 One-word comments

If multiline comments were implemented as expressions on parser-level then, in combination with `|` special character we could have one-word comments, which could be useful for describing arguments to facilitate reading of expressions. For example we could implement list comprehensions, where:

```
$<-[^[x 2] x range[0 10]]
$<-[$[x y] x $[1 2 3] y $[3 1 4] <>[x y]]
```

would be equivalent to Python's[111, Section 5.1.3]:

```
[x**2 for x in range(10)]
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

As we see this notation is acceptable (if not cleaner) for simple comprehensions, but starts being less readable for complex ones. This could be alleviated by introducing one-word comments:

```
$<-[^[x 2] --|for x --|in range[0 10]]

$<-[$[x y] --|for x --|in $[1 2 3] --|for y --|in $[3 1 4] --|if
    <>[x y]]
```

which are easily inserted inline with code and have a benefit of clearly separating individual parts of an expression, because of being easily distinguished visually from the rest. This can simulate different syntactical constructs from other programming languages, like:

```
if [>[a b] --|then
     log['|greater]
--|else
     log['|lesser-or-equal]
]
```

Except that it is not validated by the parser. But we could imagine a separate or extend the existing syntax analyzer, so it could validate such "keyword" comments or even use them in some way. For example, we could add a static type checker to the language – in a similar manner that TypeScript or Flow[?] extends JavaScript. This would be completely transparent to the rest of the language, so any program that uses this feature would be valid without it and it could be turned on and off as needed.

To reduce the number of characters that have to be typed, we could decide to use a different comment "operator", such as `%`:

```
$<-[^[x 2] %|for x %|in range[0 10]]

$<-[$[x y] %|for x %|in $[1 2 3] %|for y %|in $[3 1 4] %|if <>[x y
   ]]

if [>[a b] %|then log['|greater] %|else log['|lesser-or-equal]]
```

Or even, at the cost of complicating the parser, introduce a separate syntax for one-word comments:

```
-- ‘%:type‘ could be a type annotation
bind [a 3 %:integer]
bind [b 5 %:integer]

-- will print "lesser-or-equal"
if [>[a b] %then
    log['|greater]
%else
    log['|lesser-or-equal]
]
```

In future versions of the language, comments will be stored separately from whitespace in the EST. This enables easy smart indentation – only a prefix of the relevant expression has to be looked at, no need to filter out comments. It also enables using comments structurally, as a metalanguage for annotations, documentation, etc.

B.2 C-like syntax

Throughout this thesis I introduced multiple ways in which the basic, Lisp-like syntax of Dual can be easily extended with simple enhancements, such as adding more general-purpose special characters, macros, single-word comments (as described in Section B.1), etc.

Going further along this path, keeping in mind that a real-world language should appeal to its users we find ourselves introducing more and more elements of C-like syntax. This section describes more possible ways in which the simple syntax could be morphed to resemble the most popular languages. Ultimately all this could be implemented with a conventional complex parser for a C-like language that translates to bare Dual syntax.

Below I present a snapshot from one of designs I have been working on in order to achieve some goals described in this section:

```
fit map" {f; lst} {
    let {i; ret} [0, []];

    while ((i < lst.length)) {
        ret.push f(lst i);
        set i" ((i + 1))
    };

    ret
};
```

This would be equivalent to:

```
bind ['|map of ['|f '|lst do [
    bind ['[i ret] $[0 $[]]]

    while [<[i lst|length] do [
        ret[push][f[lst|@[i]]]
```

```
            mutate ['|i +[i 1]]
      ]]

      ret
]]]
```

Using the notation presented in Chapter 2.
One may observe that:

- The syntax is much richer, somewhat C-like, but with critical differences, reflecting significantly different nature of the language. At a first glance, it has a familiar look defined by blocks of code delimited by curly-braces, inside which statements (actually expressions) are separated by semicolons; there are different kinds of bracketing characters (`{}()[]`) with different meanings (described below)

- Names of the primitives are *full* English words, although as short as possible. `let` introduces a variable definition – similarly to `bind`. `fit <name> <args> <body>` is a shorthand for `let <name> (of <args> <body>)`, where `of` produces a function value. This translates to `bind [<name> of [<args> <body>]]`.

- `{}` delimit a string; inside a string words are separated by `;`. Strings are stored in raw as well as structural (syntax tree) form. They are a way of quoting code. This provides an explicit laziness mechanism. One-word strings are denoted with `"` at the end of the word, which resembles the mathematical double prime notation.

- `[]` delimit list literals; inside list literals, elements are separated by `,`. Lists are a basic data structure. They are actually objects, somewhat like in JavaScript. If a list contains at least one `:` character (not shown in the example), it will be validated as key-value container; if it doesn't, it will be treated as array with integer-based indices

- `()` are used in function invocations; `f(a, b, c)` translates to `f[a b c]`; `,` separates function arguments; `f x` is a shorthand notation for `f(x)`. This, in combination with currying primitives into appropriate macros allows for elimination of excessive brackets and separators. Invocations of primitives resemble use of keywords from other lanugages.

- But at the same time primitives are defined as regular functions – they are no longer treated exceptionally by the interpreter. When they are invoked, all of their arguments are first evaluated. This works, because now it is required that the programmer quote any words that shouldn't be evaluated, such as identifier names when using `let`. So primitives are just regular functions operating on code, thanks to the explicit laziness provided by strings.

- `(())` introduce an infix expression, which respects basic operator precedence: `((a + b * 2))` would translate to `+[a *[b 2]]`. This could be implemented with a separate parser based on the shunting-yard[2] or similar algorithm that is triggered by the `((` sequence. It would translate these infix expressions to prefix form and return them back to the original parser.