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 Title

The term “Pac-Man-complete” in the title refers to a somewhat humorous description of the Idris programming language[1] attributed to the language’s author, Edwin Brady[2]. In the context of this thesis 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.

0.2 Scope

This research explores various topics in the field of programming language design. Particularly, the possibilities of integrating and combining multiple representations of the same programming language in a dynamic way. I try to find ways of combining features of visual languages with text-based languages by creating a development environment which lets the programmer work with both representations in parallel or intertwine them in any way.

The created language is used to implement a Pac-Man clone. This provides a demonstration of Dual’s capabilities and is a reference for assessing the performance of the implementation.

I discuss further possible improvements in the language’s design and performance, as well as set down directions for any future research, which I intend to take on. The exploratory nature of this work lets me cover a fairly broad area, connecting programming language design, computer game development as well as web technologies and web application development.

[1] `http://www.idris-lang.org/`
[2] `https://twitter.com/folone/status/494017847585415168`

0.3 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.

0.4 Related work

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[3] game engine, which in the latest version introduced a visual programming language[4] as its primary scripting language. In fact this is the only scripting language that the engine supports, having dropped the UnrealScript language[5] included in the previous versions.

With this research, I intend to further the growth of visual programming languages, propose an even more accessible solution and explore possible improvements over comparable technologies.

0.5 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 explore and establish directions where innovation is possible in programming language design and implementation.
- To design and implement a programming language, whose *complete* textual representation must be directly and dynamically mappable to a visual representation and vice versa.
- To create a prototype of the development environment for the language on top of the web platform.
- To evaluate the usability and performance of the system by implementing a clone of Pac-Man and examining the process as well as the results.
- To explore and discuss further refinements and possible future design directions.

[3]`https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games`, `http://www.guinnessworldrecords.com/world-records/most-successful-game-engine`

[4]`https://docs.unrealengine.com/latest/INT/Engine/Blueprints/`

[5]`http://www.gamasutra.com/view/news/213647/Epics_Tim_Sweeney_lays_out_the_case_for_Unreal_Engine_4.php`

- All throughout this, the designs and implementations shall be compared and contrased with existing solutions.

0.6 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 Dual language: its syntax, semantics and basic design.

Chapter 3 talks about the architecture, design and serves as a practical documentation of the language's development environment. A comparison to existing visual programming languages is included.

Chapter 4 describes a more-than-trivial application developed with Dual: a Pac-Man clone. Performance of the implementation is assessed and possible adjustments and improvements are discussed.

Chapter 5 elaborates on programming language design both in context of Dual and in general.

Chapter 6 summarizes and concludes.

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. I may further use the terms “Dual system” or simply “system” to refer to the language and the environment as a whole.

1.1 Web technologies

One of the main design goals of the system is accessibility. This is accomplished in practice by building it on top of the most accessible and ubiquitous platform – the web platform[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 and CodeMirror[2], a JavaScript library which provides basic facilities for the text-based code editor part of the system. This part is modeled after modern web-oriented code editors with similar design philosophy[3], such as Visual Studio Code[4], Brackets[5], Atom[6] and many others.

The design of Dual’s visual representation draws from many visual programming languages. Analyzing these, we can observe many distinct approaches of which two particular designs are the most widespread and successful. These can be described as “line-connected block-based” and “snap-together block-based” visual languages. The former family is exemplified by the Blueprints Visual Scripting system of Unreal Engine 4[7] and the latter by MIT Scratch[8].

[1] `https://platform.html5.org/`

[2] `http://codemirror.net/`

[3] `https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_source_code_editors`

[4] `https://code.visualstudio.com/`

[5] `http://brackets.io/`

[6] `https://atom.io/`

[7] `https://docs.unrealengine.com/latest/INT/Engine/Blueprints/`

[8] `https://scratch.mit.edu/`

1.1.1 JavaScript

From a programming language design perspective, JavaScript has many great features, borrowed from other excellent languages and packaged in a familiar syntax. For this reason and because it is distributed with a ubiquitous environment, which is the web browser, it became one of the most[9], if not the most[10] popular programming lanugages in the world.

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[11], 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[12] 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.

This model makes reasoning about the program execution very straightforward, but is problematic when a single message takes long to execute. This problem is observed in practice when web applications cause browsers to hang or display a dialog asking the user if he 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 more messages.

The mechanism 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.

[9] `http://www.tiobe.com/tiobe_index`, `http://pypl.github.io/PYPL.html`

[10] `http://stackoverflow.com/research/developer-survey-2016#most-popular-technologies-per-occupation`, `http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/`

[11] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`

[12] At least conceptually. In practice it is much more complicated and there are exceptions to these rules. But this explanation is sufficient for further discussion.

1.2 Programming language design and implementation

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 or Scheme.

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’s also no notion of operator precedence.

The two core components of a Lisp interpreter are the `apply` and `eval` functions[2][13]. 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

[13]`http://c2.com/cgi/wiki?EvalApply`

  – *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

For detailed explanation please refer to [2].

1.2.1 Syntax

The term "Abstract Syntax Tree" refers to a tree data structure that is often built by parsers of programming languages to represent syntactic structure of source code in an abstract and 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 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 a 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.

Constructing programs with this 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 (autoindentation)
- Highlighting different syntactic constructs with different colors
- Context-based autocompletion
- Autoclosing of bracketing characters
- 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
- Etc.

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:

- 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 diff and keep track of changes in text files

1.2.2 Visual programming languages

An alternative representation is the one employed by visual programming languages. 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, he 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 briefly describe the results obtained from this survey[14].

I classified each of nearly 160 languages listed in [3], according to type of their visual representation, to one of three categories:

1. Line-connected blocks: about 66%
2. Snap-together blocks
3. Other

[14]A formal study of visual programming languages, proper classification in terms of statistics and methodical examination are not the focus of this thesis.

Additionally, I associated each language with a number $s \in [0, 3]$, which descirbes its "structure factor". This tries to quantify my subjective assessment of the readability of the representation compared to familiar text representation ($s = 3$)[15].

Below I present the results of this classification. The elements are structured as follows: name of category – percentage of languages that fall into the category – the average "structure factor" s for the category This yielded the following results:

1. Arrow/line connected blocks – 66% – 0.61

2. Snap-together blocks – 11% – 2.4

3. Other representations – 23% – 1.39, notably:

   (a) Lists – 2.5% – 2

   (b) GUI – 2.5% – 1

   (c) Nested – 2.5% – 2

   (d) Enhanced text – 2.5% – 2.75

   (e) Timeline – 2% – 1.17

   (f) *The remaining 11% are various other representations: in-game VPLs, hybrid, specialized, esoteric, etc.*

Taking into account the above and looking at the most popular visual programming languages[16]

From this and a I drew the conclusion that there are basically two main representations. A flowchart-like representation, exemplified by the Blueprints, with blocks connected by lines or arrows, which usually leaves the layout of the program source completely to the user, providing no automatic structuring. Another representation is exemplified by MIT Scratch. There, the code is represented and manipulated in terms of snap-together blocks, similarly to a jigsaw-puzzle. This representation is self-structuring and is designed to resemble a faimiliar text-based, indent-structured representations.

The advantages of the first representation is that it clearly separates

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

To design a visual representation that can be an improvement over the text-based representation all the advantages of the latter need to be kept.

[15]This is based solely on the screen shots from the editors. 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 automatic structuring by the editor, s will be low.

[16]I was not able to find and am not aware of any official or even unofficial ranking of popularity of visual programming languages, but analyzing the top hits when google searching the phrase "visual programming language" in combination with `https://en.wikipedia.org/wiki/Visual_programming_language` and my personal experience suggest that we can find these among the most popular: MIT Scratch Unreal Engine 4's Blueprints

1.3 Programming discipline

The prototype was implemented largely in the spirit of exploratory programming: “the kind where you decide what to write by writing it.”[17].

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.

[17] `http://arclanguage.org/`

Chapter 2

Dual programming language

2.1 Introduction

In order to make a visual language a viable choice for textual-oriented programmer it's important that a language has a solid textual representation.

This chapter describes the textual representation of Dual, which is the basis for the executable representation for the interpreter (the syntax tree) and for any other editable representation – including the block-based visual representation implemented in the prototype.

While implementing this project I learned that programming language design is a tremendous task, especially if the language being designed is intended to be of 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 and this is an excellent opportunity.

The character of this research project is exploratory, although I intend to further develop ideas described here and continue my research, which will, as I hope, eventually result in creation of an innovative and useful language ready for real-world use.

That aside, I believe that at least some of the ideas described here are – in a varying degree – innovative and worth exploring further.

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. This might show in descriptions, where along with talking about the prototype implementation I propose alternatives and refinements.

In chapter 5 I discuss features and ideas that reached only the design stage and were not implemented, although some of them will be further researched outside of scope of this thesis.

Most of the features of the language and editor in the prototype are implemented as a proof-of-concept, although some are more refined than others in order to fulfill the major goals of this thesis, one of which was to implement a working non-trivial application in the language. I cover a lot of design and implementation surface, only delving deep into some features that are relevant to core ideas that I wanted to convey in this thesis. The other features are implemented necessarily, as steps along the path to practical application of those ideas.

For these reasons the created environment is by no means complete and ready for use in developing complex applications. The degree of completeness of the project is reflected in:

- The core language, which is sufficiently expressive to implement any algorithm, i.e. Turing-complete in the practical sense[1]. A simple Brainfuck[2] interpreter is included as an example program [[resources]] to demonstrate this
- Implementation of several interesting additional features of the core language, which are described in this chapter
- The ablility of the language to implement also non-trivial applications. This is demonstrated by implementing a clone of Pac-Man, described in Chapter 4
- The direct correspondence of the visual and text representations, with the possibility of parallel dynamic editing; albeit the visual editing part of the editor includes only basic features and is not optimized in terms of performance; details are described in Chapter 3
- Implementation of the prototype of the language's development environment on top of the web platform, which includes a server-side and a client-side part. It runs locally on the user's machine, but is designed to be easily deployed as an online web application
- The editor has several useful features, such as basic support for text editing built on top of the CodeMirror JavaScript component with custom syntax highlighting and integration with the editor. The text editing component is integrated with the visual editing component, so that navigation or changes to each representation are tracked and visible to the user [[|]]

[ ]

The language was not originally intended as a Lisp-like language of clone thereof, but throughout the research I ended up reinventing some language constructs characteristic of Lisp and learning a lot of and about the language.

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

[2]Which is easily demostrable to be Turing-complete: `http://www.iwriteiam.nl/Ha_bf_Turing.html` and thus

An somewhat philosophical interpretation of this would be that Lisp is built on some fundamental principles that are (re)discoverable rather than invented.

Even though the language presented in this thesis is complete in the sense of being able to implement any algorithm and non-trivial applications, as exemplified by the Pac-Man clone, it is by no means a complete design. It should be viewed as a snapshot from a continuous design process that is intended to progress in the future.

Provided brief specification and description primarily describes the implementation provided with this thesis. But it is also annotated with suggestions for improvements or, in other words, descriptions of the more refined, future design, which itself is subject to change.

2.2 Grammar and syntactic features

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. A language with one of the most (if not *the* most) minimal syntax is Lisp[1].

An interpreter for List is also trivial to implement, so this is a good starting point.

There are many approaches to implementing interpreters for LISP in JavaScript[3], but the general principles are the same.

Although I opted for a minimal syntax, I did not want it to be exactly Lisp-like, as I thought the syntax of this language could be considerably improved – in terms of ease of use and parsing by a human – with a few simple adjustments, without significantly increasing the complexity of the interpreter.

There primary criticisms of Lisp's syntax are:

- Almost absolute uniformity of syntax makes the source code difficult to read by a human and thus [[]]
- In general it is hard to teach[6], because complex code gets easily confusing
- 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[4]

2.2.1 Basic syntax

Before the primary notation of Lisp, namely S-expressions[5], was established – a slightly different and, in my opinion, slightly more readable notation was used in

[3]`http://ceaude.twoticketsplease.de/js-lisps.html`
[4]`http://c2.com/cgi/wiki?LostInaSeaofParentheses`
[5]`http://www-formal.stanford.edu/jmc/recursive/node3.html`

the early theoretical publications about the language, called meta-expressions or M-expressions[4]. I first simplified this notation in the following way:

- I dropped the semicolon `;` as a separator for arguments, as it is entirely superfluous and there is no need for the programmer to type it or the parser to be concerned with it; this means that the only separating characters are the whitespace characters, exactly as in pure S-expressions
- 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]`
- I decided not to include any other syntax in the first prototype, as the one described so far is entirely sufficient to represent any Lisp-like expressions – it maps directly to S-expressions[6]

This gives a notation that is somewhat in between S-expressions and M-expressions. This was the basic syntax of the first prototype of the language. It can be defined[7], using pure, left-recursion-free BNF notation with the addition of a regular expression (between `/` delimiters) in the definition of `<word>`, as follows:

```
<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> | ""
```

The regular expression can be read as "any character which is not whitespace, `[` or `]`". This means that aside from whitespace, which acts as expression separator there are only two special characters – the square brackets – that the parser have to worry about. For this reason it is very trivial to implement.

The above – somewhat verbose – definition is obviously somewhat similar to a Lisp BNF description[8].

[6]Any additions can be considered syntax sugar. Such syntax extensions were introduced in later prototypes and designs and some of them are included in the final prototype included with this thesis; see [][]

[7]This definition is included here only for the sake of formality. I believe that for such a simple grammar BNF seems to introduce more noise and is unnecessarily more complex than a textual description in terms of regular expressions or simply verbatim parser source code. For these reasons any extensions to this basic grammar will later on be described in these ways.

[8]`http://www.cs.cmu.edu/Groups//AI/util/lang/lisp/doc/notes/lisp_bnf.txt`, `http://cui.unige.ch/db-research/Enseignement/analyseinfo/LISP/BNFlisp.html`

An example of a valid expression in light of this definition would be:

```
do [
    bind [a 3]
    bind [b 5]
    bind [is-a-greater if [>[a b] true false]]
    is-a-greater
]
```

Where[9]:

- `do` is a language primitive that serves the role of a single block of code, much like blocks delimited by `{` and `}` in C-like languages.
- `bind` is a basic construct for defining variables, like `var` or `define` in other languages.
- `if` serves as a basic conditional evaluation construct. Its semantics are like those of the analogous construct in Lisp.

Advantages of this M-expression-based notation over S-expressions are:

- 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[10]
- 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 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 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 any minimal syntax, because such syntaxes have the property of being very homogenous. In the next subsections and later throughout this thesis I gradually introduce extensions, which make the syntax 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.

[9]Note: I will introduce brief definitions of language constructs as they appear in the presented listings. For a comprehensive list of primitives and functions see Section 2.3 of this chapter.

[10]See `http://www.tiobe.com/tiobe_index`; 11 out of the top 20 languages as of June 2016 have C-based syntax (by this classification: `https://en.wikipedia.org/wiki/List_of_C-family_programming_languages`). If we extend the syntax family to Algol-like, its virtually 20 out of 20 – `http://www.martinrinehart.com/pages/genealogy-programming-languages.html`. There are no languages with Lisp-based syntax among the most popular ones.

2.2.2 Comments, whitespace and the syntax tree

The parser was further extended with support for comment syntax similar to the ones 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]

--[
    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]
    ]
]
```

In principle multiline 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[11] 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]
```

[11]Inspired by `https://en.wikipedia.org/wiki/JSDoc`)

```
        --]]

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

Nevertheless the implementation in the prototype treats the comments as a stream of characters, taking into account nesting and balancing of brackets, but doesn't keep them as a tree-like structure.

Whitespace characters and comments have no semantic significance, unless serving as separators could be considered one. After building the syntax tree, bracketing characters also serve no purpose and can be safely be discarded, without influencing the interpretation of the program.

Despite this, all of these are included in the syntax tree generated by the parser. Storing these characters in the syntax tree means that the entirety of the textual representation, in structural form, is accessible by any other representation we might devise. This allows for implementation of variety of interesting "smart" language and editor features.

A thing to note at this point is that such syntax tree can't be described as "abstract"[12], as it includes all of the "concrete" syntax, with its runtime-insignificant elements. The syntax tree that is built for the Dual language is also later extended with references to other representations of code. For these reasons, I will later on use the acronym Enhanced Syntax Tree to refer to the representation used internally by the Dual language interpreter.

This representation greatly simplifies the implementation of and integrates with the language the following features:

- Automatic indentation

- Documentation comments. Comments can easily be associated with corresponding code blocks (syntax tree 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

- This also means that any expression can be stringified on-the-fly and this string can be used as a value in the program. This feature allowed me to completely omit definition of strings at the parser level, although this is not a very efficient solution. Nevertheless keeping strings in such structural form – as syntax trees – in combination with pattern matching enables language-

[12]According to these definitions: `https://en.wikipedia.org/wiki/Abstract_syntax_tree`, `http://c2.com/cgi/wiki?AbstractSyntaxTree`

native structural manipulation of strings[13][14]. For example we could write:

```
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 single words and binds these words to identifiers provided as its arguments and `characters` performs an analogous operation on the single character-level. The notation `{rest}` matches zero or more arguments (see Section 2.2.6 for details). `log` outputs the values of its arguments to the JavaScript console.

Such representation was implemented in order to fulfill the design goal of direct and complete mapping of the textual representation into any other representation.

2.2.3 Numbers

Numbers in the language are represented as JavaScript numbers. This means that there's only one number type – 64-bit floating point[15]. 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.

[13]See: `https://reference.wolfram.com/language/tutorial/WorkingWithStringPatterns.html` and `https://en.wikipedia.org/wiki/Pattern_matching#Pattern_matching_and_strings` for similar concepts.

[14]I chose the structural representation as the only representation, because the performance penalty is acceptable in the prototype implementation. An obvious and very simple optimization would be to keep the raw form of the string as a value in the corresponding syntax tree node. Having these two forms alongside each other would enable the programmer to use the familiar string manipulation methods as well as structural manipulation.

[15]Defined by the IEEE 754 standard: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469`, `http://www.2ality.com/2012/04/number-encoding.html`

- 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.

This shows a possible way of optimizing the representation of strings, which I intend to introduce in a future version of the language.

2.2.4 Zero and single argument expressions

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 (`|`). This character is used for single-argument expressions, as in:

```
-- compute factorial of 32:
factorial|32 -- equivalent to 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]]
```

The above example shows that if a function is invoked with one argument, we can omit the closing brace and replace the opening brace with `|`. The parser produces equivalent syntax tree.

Another special character (`!`) was introduced for analogous use for 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]!]
forty-two -- evaluates to 42
```

In combination with macros

A unique macro system, described in Chapter 5[16] in combination with these two (| and !) special characters help reduce the amount of bracketing characters even further.

For example, if we define a `match*` and `of*` macros as described, 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]
```

which is somewhat similar syntactically to ML-style[17] languages, could 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
]
```

[16]This macro system is not included in the final version of the prototype, although a proof-of-concept of it that I implemented in earlier prototypes is the basis for this description.

[17]`https://en.wikipedia.org/wiki/Standard_ML#Algebraic_datatypes_and_pattern_matching`

where `apply` would be defined analogously to Lisp's `apply`. `of` would be defined as a function primitive with arity 2..*, which treats its penultimate argument as the function's body and all the preceding arguments as patterns for the function's arguments. The last argument is used when such a function is called and the values supplied as arguments don't match the patterns. If the argument is a function, it will be called with the same values as arguments and if it's a value it will be returned.

I used such a solution for pattern matching in the early prototypes, but replaced it with a native `match` construct (described in Section 2.3) for performance reasons. Nevertheless this shows that a few simple, but general syntax rules and a powerful macro system, can be a very flexible tool for extending syntax.

The pattern matching mechanism is explained in the next subsection.

2.2.5 Pattern matching

A simple, yet powerful pattern-matching facility was added to the language.

Pattern matching works with bindings, functions (although the primary function-producing expression in the prototype doesn't use it by default), `match` primitive and macros (not available in the prototype).

The pattern matching works in a way similar to most other languages that support this feature (e.g. ML family). The general rules are[18]:

- 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 doesn't bind:

```
-- returns its third argument,
-- discards the rest:
```

[18]For brevity I assume here that 'of' is a primitive that works like described in 2.2.4, where the alternative argument is optional. This is how it was implemented in an early prototype of the language. If no viable alternative was present, an error was thrown.

```
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).

- 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[19]:

    ```
    -- 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.

The above examples show pattern matching used for destructuring values and binding their components to identifiers and for function definitions. There's also

[19]`https://en.wikibooks.org/wiki/F_Sharp_Programming/Pattern_Matching_Basics#Using_Guards_within_Patterns`

a `match` primitive, which can serve the role of a `switch` statement from C-like languages. Although pattern matching makes it much more powerful than that, as any values supported by the pattern matching system can be matched, including lists, which allow us to switch on multiple values and in any combinations.

The `match` primitive's first argument is a value to match and all subsequent arguments are two-element lists, where the firs 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 successful match, which is the first one that matches. The subsequent matches are not evaluated.

```
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]]
]
```

•

With an arsenal of these few simple pattern matching tools we can use a lot of useful features, which further add expressivity to the language. We can also imagine many possible extensions and generalizations, as briefly discussed in Chapter 5.

- Destructuring assignments or, more precisely, destructuring definitions.[20]. An example of such definition would be:

[20]Destructuring could easily be extended to mutation as well, although I have found it sufficient to be usable only in definitions, while implementing the prototype.

```
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]
```

2.2.6 Rest parameters and spread operator

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

- Rest parameters mechanism known from Lisp[21], recently also adopted in JavaScript (as of the ECMAScript2015 standard[22]). That is, 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 enables the user to easily define variadic functions, which can be called with a variable number of arguments. This works in any place, where pattern matching works:

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

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

  thus enabling non-exact matching.

- Spread operator (also inspired by the analogous feature from ECMAScript2015):

  ```
  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]}]
  ```

[21] https://www.gnu.org/software/emacs/manual/html_node/elisp/Argument-List.html
[22] https://developer.mozilla.org/pl/docs/Web/JavaScript/Reference/Functions/rest_parameters

This provides a much nicer and more powerful alternative to `apply`, Lisp's fundamental function, which applies a function to a list of arguments. It's a way to flatten any list onto a list of arguments. This works for multiple values and lists as well:

```
-- alternative way to achieve the same
-- result as in the previous listing
-- logs `9 8 7 6 5 4`:
f[{9 args $[5 4]}]
```

- String interpolation notation:

```
bind [name '|Bill]

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

As we can see this gives us a very convenient notation for string interpolation, similar to e.g. template literals in JavaScript[23]. In order to escape curly braces, they should be doubled:

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

I also added a special type of string – an HTML string, where interpolation notation is the other way around – 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.

- Unquote notation for macros[24] – see Chapter 5.

2.3 Basic primitives and functions

Below is a description of the primitives and functions supported by the Dual language. Each item is structured as follows:

[23]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

[24]Analogous to unquote or , in Lisp: https://docs.racket-lang.org/reference/reader.html#%28part._parse-quote%29

- `<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).

  <description> is a brief description of the function/primitive.

2.3.1 Language primitives

The Dual language supports the following primitives:

- `do [0..*]`

  Evaluates its arguments in order and returns the value of the last argument.

- `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[26], as definitions are processed in order in which they appear in code.

- `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.

[25]`https://developer.mozilla.org/pl/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment`

[26]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var#var_hoisting`

- `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. [[first-class environments, Bla paper?]]

- assign
- code
- macro
- of
- of-p
- procedure
- match
- cons
- invoke*
- .
- :
- @
- dict*
- async*

Basic functions and values:

- `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[[link]].
- `typeof` wraps JavaScript's `typeof` operator.
- `or` and `and` are the basic logical operators – analogous to `||` and `&&` in JavaScript.
- `any` and `all` are like the above, but accept variable number of arguments. These return either `true` or `false`.
- `not` is the negation operator (`!`)

Chapter 3

Development environment

The goal is to build an online Integrated Development Environment IDE, similar to Codeanywhere[1] or Cloud9[2], works offline as well.

3.1 Overview

A folder is considered a project, similarly to modern code editors, such as Brackets or Visual Studio Code.

The current version of the development environment is intended to be used offline, on user's machine. Nevertheless it is designed so that it could be easily transformed into an online system.

I decided to implement the system with minimal dependencies, so it can be easily installed and so I can achieve a greater level of integration by having more control over every part.

The only required dependency for the basic functionality of the prototype to work (which is the editor) is a web browser and the CodeMirror library. An additional dependency is the Node.js environment.

The language's development environment is implemented as a web application. It consists of three parts:

- The server part, implemented in JavaScript on top of Node.js. This part's functions is mainly to enable access to user's file system, so any local folder can be opened as a project – modern web browsers restrict access to the local file system, because of security reasons. The server part also handles persisting changes to files and configuration.

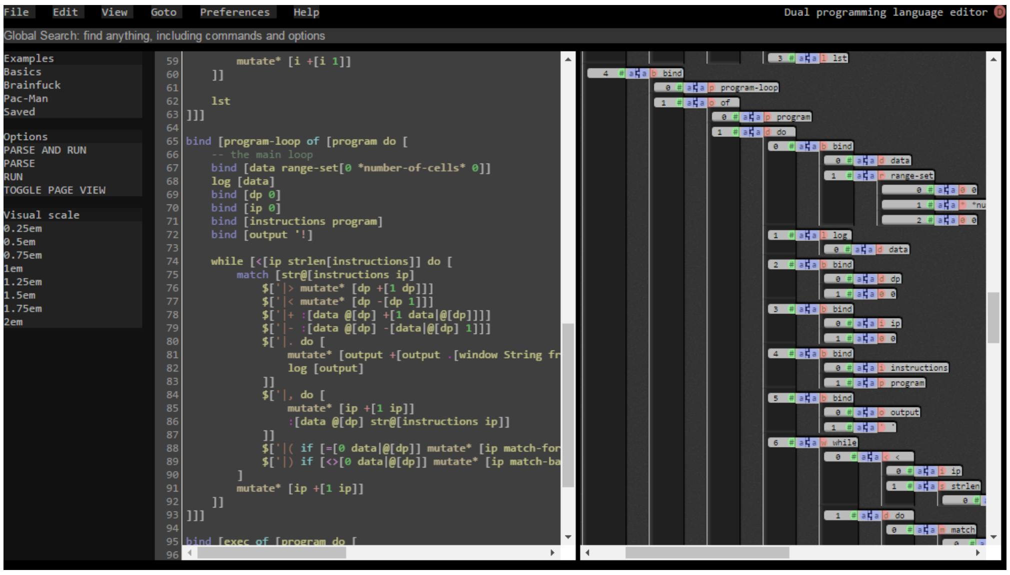
- The project manager part, which communicates directly with the server part. The connection is maintained over a WebSocket[3]. This part provides access to user's file system via a custom folder selection interface. Basic configuration

[1]`https://codeanywhere.com/`
[2]`https://c9.io/`
[3]`https://developer.mozilla.org/pl/docs/WebSockets`

of server communication, such as changing the address and ports is also possible. Once a project is selected, the user may open it in the editor part.

- 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[4].

The project manager and the editor, which can be considered the front-end parts of the system are designed to be a Single-Page Application[5]. 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[6].



```
        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 3.1: The editor

Figure 3.1 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 a global search input (not implemented in the prototype). 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 following options are implemented in the prototype:

- Available from the menu bar:

[4]https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[5]https://en.wikipedia.org/wiki/Single-page_application
[6]http://eloquentjavascript.net/13_dom.html

  - 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.

3.2 Text editor

The text editor is built on top of the CodeMirror framework[7]. It was integrated with the editor in the following way:

- A custom syntax highlighting mode for Dual was defined.
- 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. This 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 every representation that is associated with the EST.

[7]`https://codemirror.net/`

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 doesn't have to be reestablished every time a modification is made.

We can distinguish three representations used by the system:

1. The EST, which is the master representation of the program.

2. 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.

3. The visual representation, which is implemented in terms of pure HTML tables fully styled with CSS. This allows for easy and complete customization of the representation.

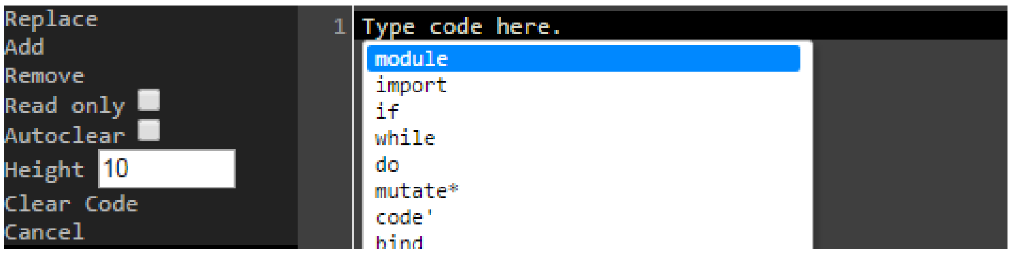


Figure 3.2: Visual editor's context menu

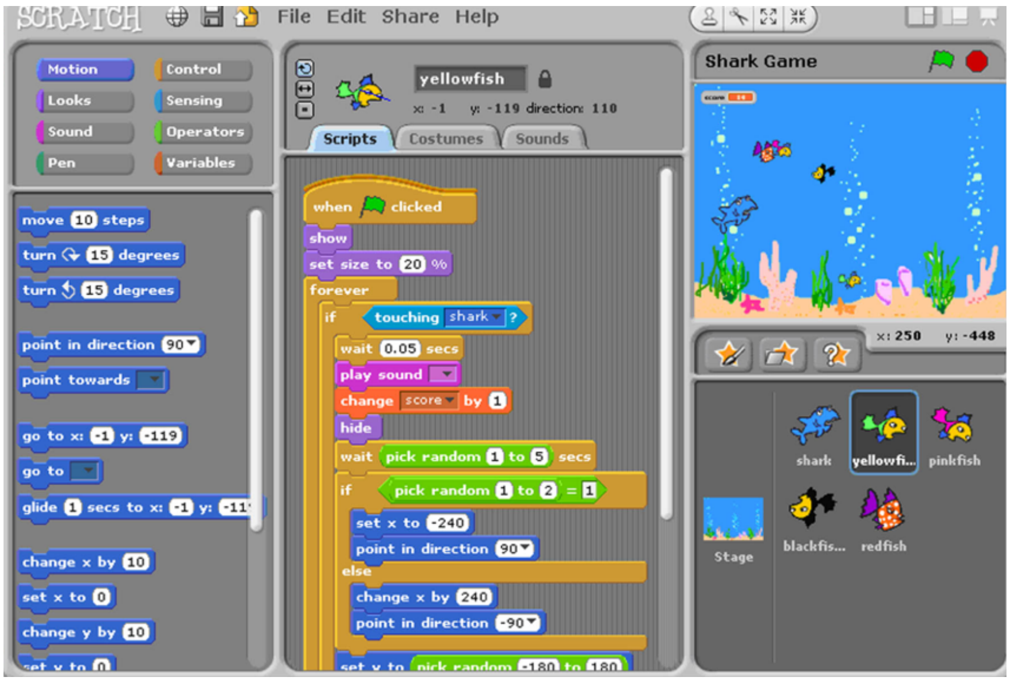
In case of the visual representation this is implemented in a rather straightforward way: every EST node has a corresponding set of DOM nodes. Thanks to this, we can track any actions performed on the DOM through the standard browser-implemented interface. This is solved in the prototype by attaching `click` event handlers to relevant nodes. Such an event triggers the following:

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

- A context menu appears similar to that depicted in 3.2. This menu has basic options for editing the visual nodes: Replace, Add and Remove. These perform their corresponding action on the currently focused node and propagate it to all representations. The Remove option simply removes the selected node and its subtree from the DOM, the EST, as well as the associated text fragment. Add and Replace make use of the small text-editor area next to the context menu. It contains a list of possible names of nodes to be inserted in case the user wants to a node. 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 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 autocomplete functionality on top of CodeMirror. Every item in the autocomplete 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. Autocompletion should also be made context-sensitive, similarly to modern code editors.

The templates could also be selected by the user from a visual library of puzzle pieces, like the one in Scratch (Fig. 3.3).



```
when [flag] 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 3.3: MIT Scratch programming language editor[8]

3.3 Visual editor

3.3.1 The visual representation

Basic design principles: make it no harder to use than the textual representation ideally it should add useful capabilities, without taking away these provided by text editors

Existing visual languages are mostly criticized, because they fail to meet these basic criteria.

Flexibility

The fact that the visual representation is composed purely out of HTML and CSS
Fully customisable with CSS

Design

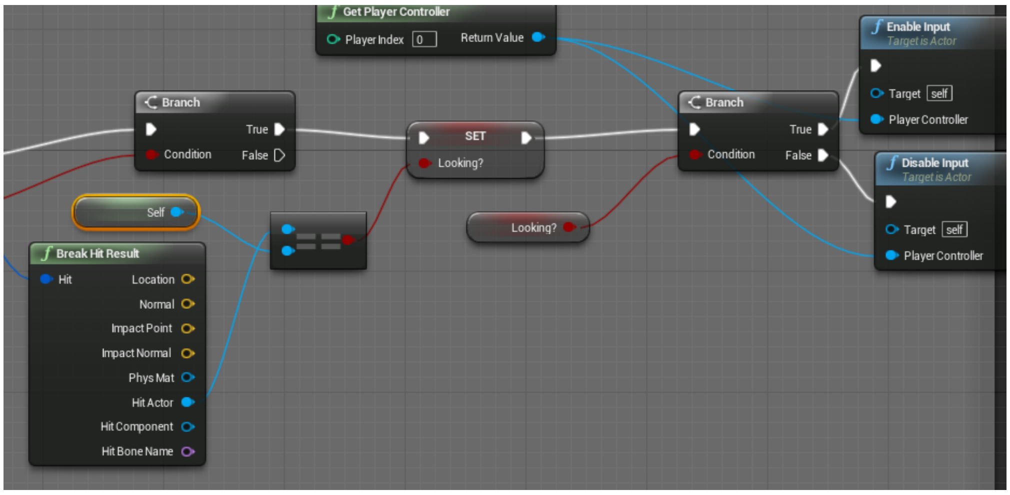


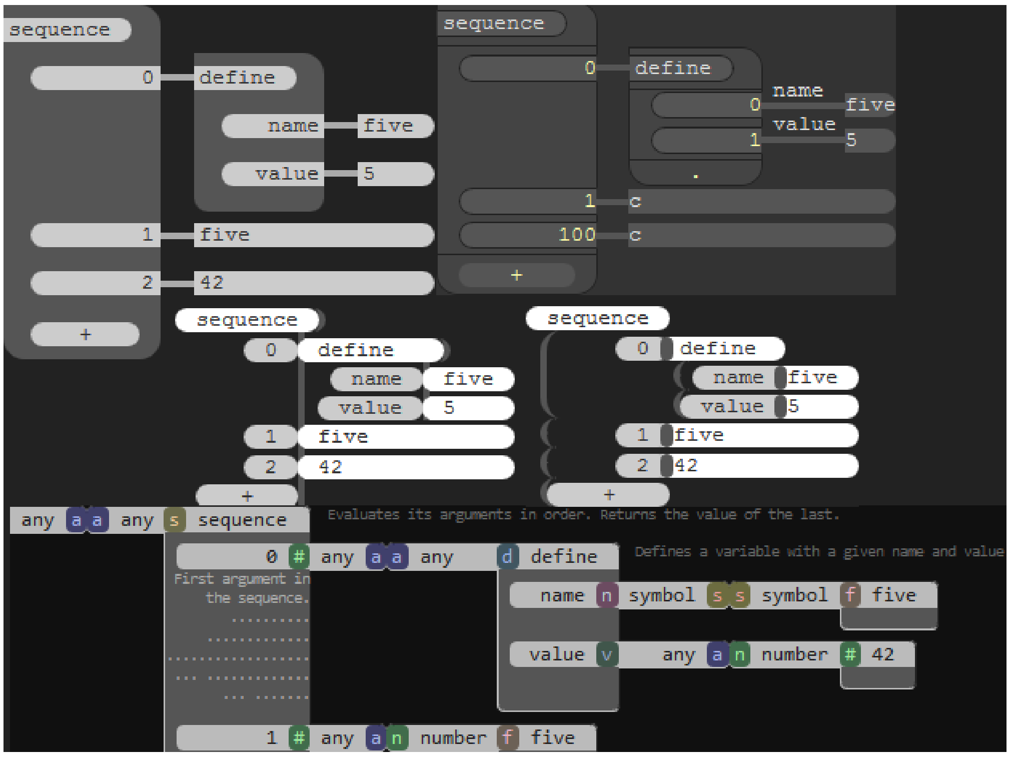
Figure 3.4:

Many iterations:

The visual form allows us to provide much more information about different elements of the program. Spatially relate this information with these elements through blocks and connections. Relate expressions with other expressions through connections, which can also carry additional information.

We can distinguish the following visual elements:

- 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 operator. Next to the header a documentation comment might be displayed.
  - Slots, which are the numbers or names of the arguments followed by an icon; below these documentation comments could be displayed.
  - Possibly additional buttons, which could be used 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 expression return value's type annotation.


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 3.5:

Icons are a way to minimize the use of text to represent different entities

The text representation is very compact, which often is an advantage. This advantage can be maintained by the visual representation by making all the additional information optional. The user should have easy way of configuring whether or not and what informations should be displayed. By folding some textual elements into icons the visual representation could actually be made more compact than the text form.

Structure

The programmer could have the ability to alter the structure of the representation, but he should not be required to constantly shape it. A connected-block representations, such as the one in Unreal Engine 4 has no mechanism that automatically structures the visual code similar to autoindentation and other useful autostructuring features that evolved in code editors over time and experience with using text-based programming languages. This can be considered a regression on the visual editors' part.

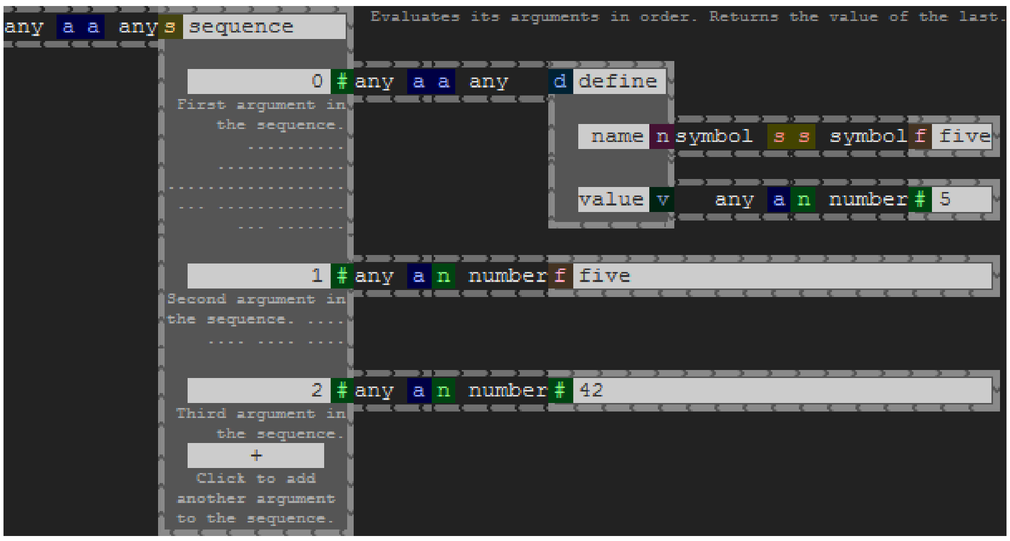
The early designs show the following features that are not implemented in the final prototype:

- Names of the arguments are displayed instead of numbers if sensible.
- Names of types of variables

The colored squares with letters inside are actually placeholders for icons. I imagine a design, where the user is able to click on those icons and fold the blocks into a more compact form, hiding the names and excessive text. This could be done on the level of individual blocks, whole subtrees or the entire program – similar 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 text below the slots could be documentation comments associated with the given argument. Their visibility could be toggleable through clicking on them, on an individual or global basis, similarly to icons.

We can observe that there's a need to manipulate or set visual properties of individual objects, clusters of objects/subtrees as well as the entire program tree.


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

3.4 Performance

It could be optimized similarly to CodeMirror or other web-based text editors or applications. That is, only a visible portion (plus a margin, which allows for fast scrolling) of the code is rendered as DOM nodes at any time. The scrollbar is virtual and controlled by the editor rather than the browser.

Text editors like CodeMirror use similar amount of DOM nodes [[]], but thanks to these optimizations are able to handle megabyte-sized[9] text files and are used in many real-world applications[10], which includes being a built-in editor in developer tools in major web browsers.

[9]http://codemirror.net/doc/internals.html#approach
[10]http://codemirror.net/doc/realworld.html

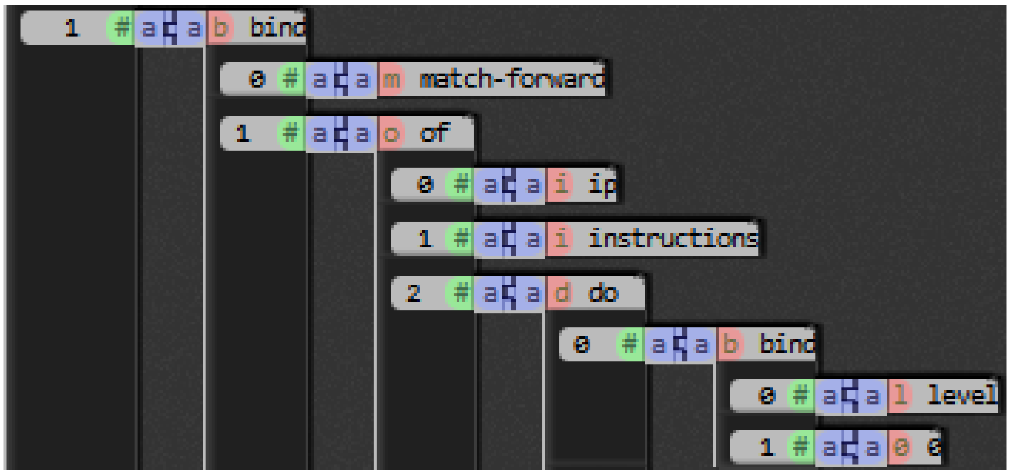


Figure 3.7:

Another source of inefficiency is that parsing is done twice – once by Dual's parser and once by CodeMirror's system, which are incompatibile. A solution to that would be to implement a custom text editor or extend/modify CodeMirror to work with Dual's parser.

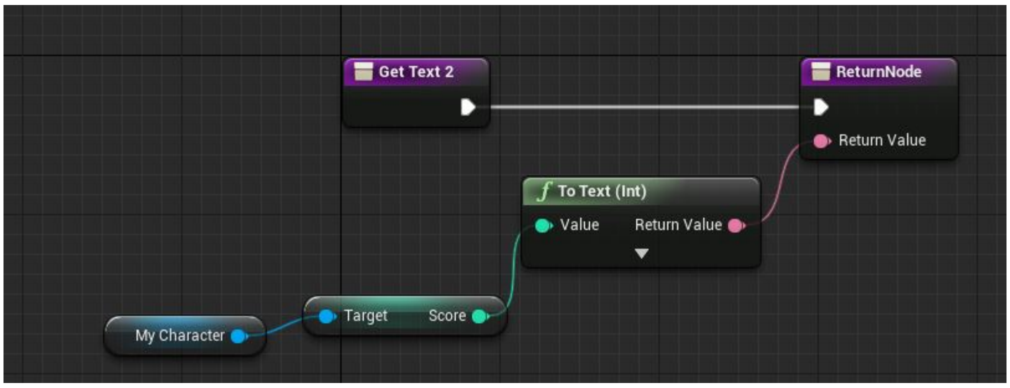


Figure 3.8: Blueprints Visual Scripting[11]

Chapter 4

Case study

Dynamic languages with a garbage collector have the advantage of letting the programmer use the exploratory style of programming, where he doesn't have to worry about memory management or other low level considerations. [][] Doesn't have to design a complex type hierarchy or any similar scaffolding. He can just jump in and start implementing an idea. This is excellent for prototyping.

But when it comes to performance and robustness this approach shows its downsides very quickly. The safety of static types combined with a good development environment catches a lot of bugs and inconsistencies before runtime. Garbage collector mechanisms vary in implementation, each showing a different performance characteristic. In this chapter I will describe the implementation of a clone of Pac-Man in Dual and performance issues that I've encountered. These are very much related to the JavaScript environment, notably the event loop and the garbage collector, which has different implementations across browsers. The latter did shows a significant difference when comparing different web browsers.

Implementation of a non-trivial application allows to test the language design and quickly establish which features are the most useful in practice. I found that I could do away with a lot of the more complex ones

4.1 The game

It is actually a port of my earlier clone of the game, which was written in Links[1], a functional language.

4.1.1 Main loop

A typical game loop in a modern JavaScript game[2] relies on the `requestAnimationFrame` method[3]. This method takes a single argument, which is a function callback. This

[1] `http://groups.inf.ed.ac.uk/links/`

[2] `https://developer.mozilla.org/en-US/docs/Games/Anatomy`

[3] `https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame`

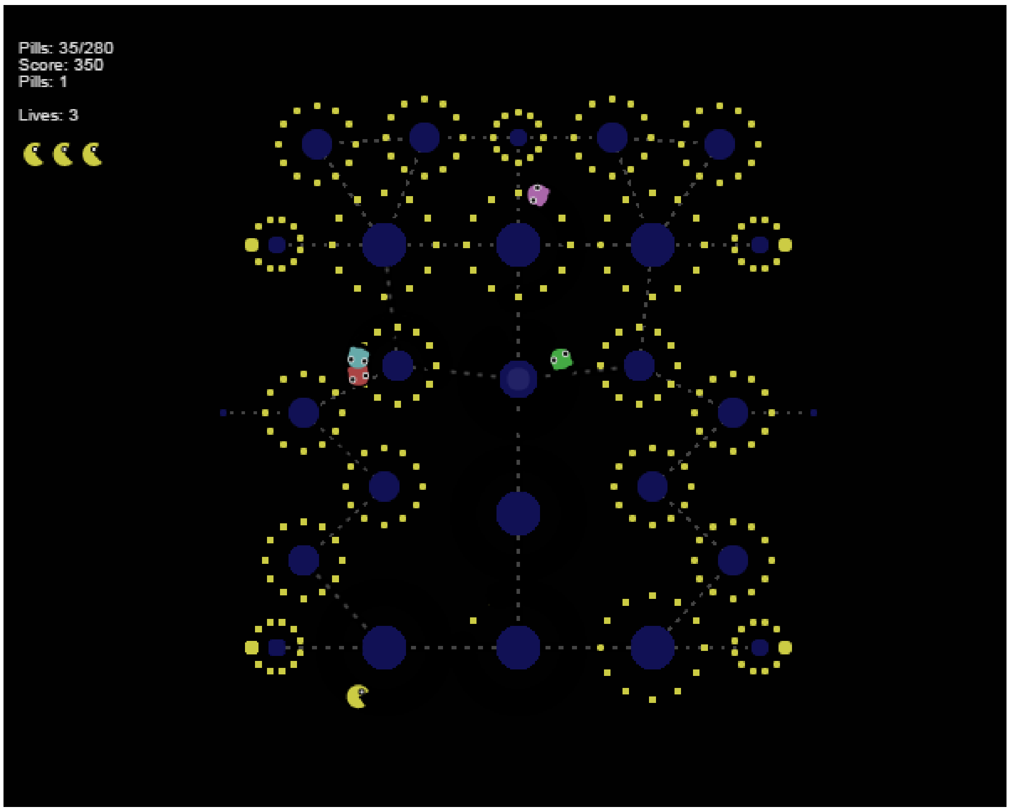


Figure 4.1: A screenshot from the game

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

-- the main loop function:
bind [main-loop
-- arguments:
--     game-state -- current game state
--     last-tick -- time of the last
--        game state update
--     fps-info -- an object used for debugging,
--        which contains information about
--          the frame rate
--      current-time -- the time of  the current
--          invocation of the loop
of [game-state last-tick fps-info 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
                fps-info
                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]
            ]
        ]
    ] false]

    -- 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]]
        -- invoke the function that updates
        --        the game-state:
        mutate [
            game-state
            main-game-logic [game-state input-queue]
        ]
        mutate [i +[i 1]]
    ]]

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

4.2 Performance

Disparity between Firefox and Chrome, reflecting differences in garbage collector implementations.

Issue: interpreter is blocking the event loop. There's a lot of intermediate objects created that have to be garbage collected.

General pattern: Chrome more frequent garbage collections more regular more predictable

Details are a topic for another thesis

4.3 Possible improvements

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

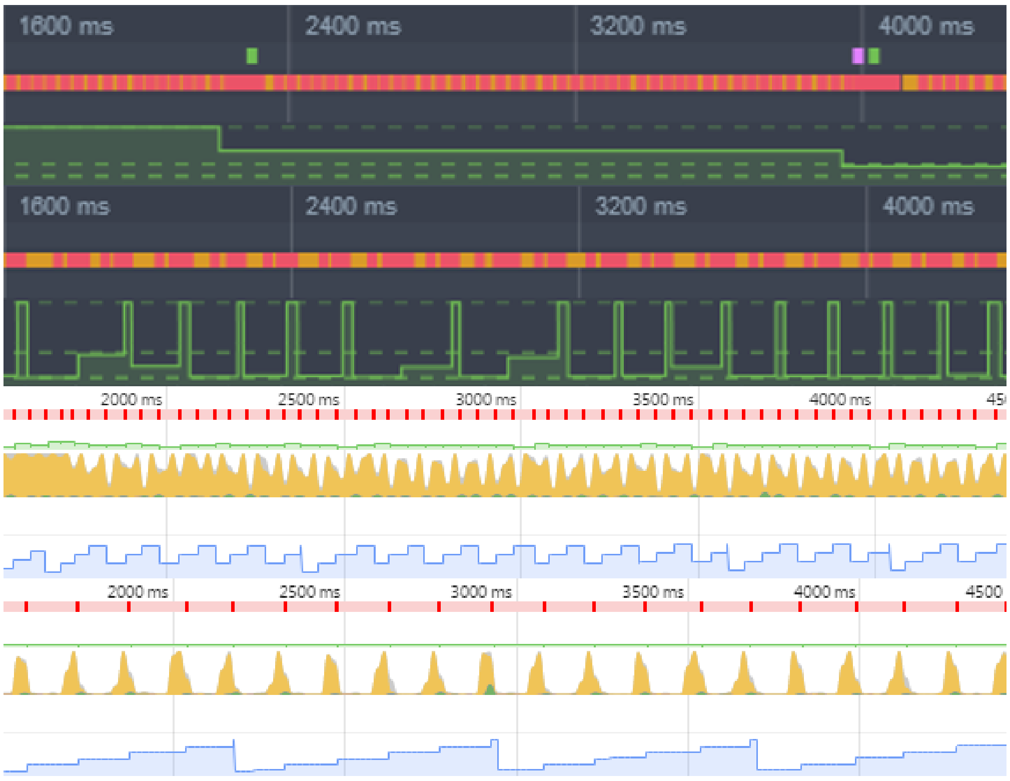


Figure 4.2: Comparison of Firefox' and Chrome's profiler outputs

Chapter 5

Design discussion

5.1 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[1]:

```
[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]
]
```

[1]`https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions`

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[2] 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.

5.2 Structural string manipulation

```
words[_ _ _ fourth _ sixth] -- super fast, out of the box
characters[_ _ _ _ fifth]
```

[2]https://flowtype.org/

5.3 Just-in-time macros

One feature that I experimented with while creating the prototype of Dual is support for "first-class just-in-time expanded macros". By this I mean macros similar to those found in Lisp, but with a few key characteristics, which differentiate them from conventional implementations of macros in Lisp-like languages. These are:

- Macros are expanded upon evaluation; when a macro invocation is encountered by the interpreter, it is expanded into code; the node in the EST containing the macro invocation is permanently replaced by the expanded expression, which is subsequently evaluated and its value is returned as the value of the invocation.
- Macros can return other macros in a straightforward manner; this feature nicely composes with the variation of Lisp syntax found in Dual; I used it extensively to improve it, mostly with the goal of reducing the amount of adjacent closing brackets in the source code.

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

- Primitives are moved into the top-level environment; 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 are 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[5]

The below listing presents an example of a macro named `if*` that wraps the `if` primitive in a slightly different syntax. 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:

```
bind [if* macro [a op b macro [{then} macro [{else}
          code'[if [apply[{op} {a} {b}] do[{then}] do[{else}]]]
]]]
```

```
if* [a < b][
        log ['[a is less than b]]
        a
][
        log ['[b is less than or equal to a]]
        b
]

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

TODO: elaborate

***

A somewhat tangent, but interesting observation here is that a Lisp interpreter could be simplified further by removing all special cases from `apply`. It would only apply a function to arguments, without checking its type. This would cause the following:

- All primitives would cease to be "special".
- If we keep the strict evaluation strategy, a programmer would have to explicitly quote all expressions that should not be immediately evaluated.
- We could also switch to a variation of lazy evaluation, which could be best described as "explicit evaluation", where evaluation *never* happens unless explicitly requested. Under this evaluation strategy, a programmer would have to explicitly evaluate all expressions that should be evaluated.

Assuming the explicit evaluation strategy, the `if` primitive could be defined in the interpreter as follows (in JavaScript):

```
// args is the list of arguments, which would be provided by apply
// this is a greatly simplified implementation, to demonstrate the essence
function if (args, environment) {
        var condition = args[0], consequent = args[1], alternative = args[2
        var conditionValue = evaluate(condition, environment);

        if (conditionValue) {
                return evaluate(consequent, environment);
        }
        return evaluate(alternative, environment);
}
```

Assuming we have `bind` and all other necessary primitives defined to conform to this evaluation strategy and a terse syntax for explicit evaluation (**'**):

```
-- ' causes an expression to be evaluated

-- bind would always evaluate its second argument:
'bind [a 5]

-- +, - and other arithmetic operators would always evaluate all of th
'bind [b +[a 7]]

-- log would evaluate all of its arguments
-- logs '12':
'log [b]

-- of would not evaluate any of its arguments:
'bind [factorial of [n do [
        'bind [n-value n]
        *[n-value factorial[-[n-value 1]]]
]]]
```

TODO: how this could be useful compiling to simplified Lisp

5.4 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 5.1), 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[3] 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.

5.5 Universal visual editor

The structure produced by a visual editor does not have to necessarily be a syntax tree of a particular language. Text editors allow inputing arbitrary sequences of characters, which are transformed into a syntax tree by a specialized parser. Analogously, an universal visual editor could allow insertion, connection and mainpulation of arbitrary generalized blocks, thus forming a truly abstract syntax tree (language-independent). This tree could then be "parsed" to produce a syntax tree for a particular language.

[3]`www.cs.utexas.edu/~EWD/MCReps/MR35.PDF`

Chapter 6

Summary and conclusions

I intend to continue my research with the goal of creating a modern real-world programming language useful for a specific range of tasks

Bibliography

[1] Douglas Crockford. Syntaxation. `gotocon.com/dl/goto-aar-2013/slides/DouglasCrockford_Syntaxation.pdf`, September 2013. Presentation from the 2013 edition of the "goto" conference, `http://gotocon.com/aarhus-2013/`.

[2] Julie Sussman Harold Abelson, Gerald Jay Sussman. The Metacircular Evaluator. In *Structure and Interpretation of Computer Programs*, chapter 4.1. MIT Press, 1996. Also available online: `https://mitpress.mit.edu/sicp/`.

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

[4] John McCarthy. The implementation of LISP. `http://www-formal.stanford.edu/jmc/history/lisp/node3.html`, February 1979. Part of the "History of Lisp" draft, `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`.

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

[6] 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`.

Glossary

Document Object Model [[DOM description]]. 36

Enhanced Syntax Tree [[EST description]]. 20

Acronyms

AST Abstract Syntax Tree. 8

DOM Document Object Model. 36, 38, *Glossary:* Document Object Model

DSL Domain-Specific Language. 19

EST Enhanced Syntax Tree. 20, 37, *Glossary:* Enhanced Syntax Tree

IDE Integrated Development Environment. 35

SPA Single-Page Application. 36

VPL Visual Programming Language. 10

Appendix A

Płyta CD

Zawartość katalogów na płycie:

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 application

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.