Setanta - Developing an Irish programming language, learning environment, and an original parser generator for TypeScript

Eoin Davey

Final Year Project - 2019/2020 - 5 credits

B.Sc. Computational Thinking

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare
Ireland

Contents

List of Figures

Chapter 1:   Abstract

The English language dominates the world of programming languages, it is well established as the lingua franca of the programming world. Effectively all programming languages that are used today are designed to be used in English. Studies have shown that language affects the thoughts of the speaker, so how does it affect how we design our programming languages? This project is an exploration of the design and implementation of a programming language (*Setanta*) from the ground up, to be written in a non-English language, namely Irish, and the effects that the "host" language has on its syntax and semantics. In recent times Irish is often thought of as an academic, historical language, however, Irish is a language used by 73,000 people daily. There is a large contingent of fluent Irish speakers who, if they want to learn how to program, have no choice but to learn in English. In this project, we create *Setanta*, a modern, powerful Irish programming language. In the process of designing *Setanta*, we discover syntactic constructs in traditional programming languages that are tied to English and introduce new features that are motivated by the Irish language. We develop and launch an online learning platform (try-setanta.ie). When implementing an interpreter for *Setanta*, we find and fill gaps in the tooling available for creating a programming language to be executed in the browser. Specifically by creating an innovative parser generator (*tsPEG*) for the language TypeScript.

Chapter 2:   Introduction

For easier distinction between programming languages and "human" languages, from this point I will refer to programming languages as **PLs**, and traditional languages as just **languages**.

2.1   Motivation

English is the language of choice for the programming world, even PLs developed in non-English speaking countries are designed to be written in English, e.g., *Lua* (developed in the Netherlands), *Ruby* (developed in Japan). This focus on one single language must have some impact on the way we design our PLs. Many PLs have been written for other languages, but if you go to use one you will almost certainly find that it is a *translation* of a PL originally written for English speakers[3]. If we design a PL from the ground up around a non-English language, what changes do we see between it and the industry-standard English PLs? By finding deep links between the English language and the PLs used in industry, I hope to reveal subtle ways in which diversity of thought is being constrained by forcing all programming to be done through the lens of English.

Irish was chosen as the language to build the new PL around for many reasons, the obvious being that it is the native language of Ireland, so it is of interest to an Irish audience, but this is not the only reason. Irish has historically faced significant hostility, and today finds itself a minority language in its own country, however, it is still spoken by over 73,000 people daily[1]. If any Irish speaking person wishes to learn about programming, they have no choice but to do it through the medium of English. By creating an Irish PL and an online learning environment around it I hope to enable people to learn to program in the way that they want to.

2.2   Problem Statement

This project involves the design and implementation of a new, modern, innovative PL, named *Setanta*. *Setanta* is to be developed entirely in Irish. It will not be a translation or a modification of a previously existing PL, this is to allow it to be influenced by the Irish language at every stage of the design and implementation process.

   Setanta will be designed with education in mind. It will be built to run in the browser to enable high ease of access to as many people as possible. By running the code in the browser, no installations are required to use the PL, just a web browser.

   Setanta must be a modern PL with all industry-standard features, this is to ensure that by learning *Setanta*, you learn the most fundamental programming concepts. Additionally, it should be built to overcome the limitations of the browser environment. Executing code in the browser limits you to a single thread and no blocking operations, by building a language on top of this we can abstract out these limitations and enable users to use concurrent functions and blocking operations such as IO.

   An online learning environment will be created where the user can write and execute *Setanta* in the browser. It should be accessible and easy to use. To assist in the learning process the environment will take inspiration from popular educational tools like *Scratch* and *Logo* and have a graphical interface where the user can draw and interact with shapes and animations. Research has shown that visual engagement is a strong pedagogical tool. The use of visual elements in educational approaches improves the learning experience[4].

   To implement an interpreter for a PL a parser is needed, usually a parser generator is used to do this. However, as not many languages are built to be browser-first, the parser generator choices available for TypeScript (my PL of choice for this project) were not quite suitable. This leads to the additional part of this project to create a novel parser generator for TypeScript. The parser generator must be powerful enough to support *Setanta*, as well as to be capable of bootstrapping its own parser. It should be built on the latest innovations in parsing technology, providing accurate syntax error detection and ASTs to the user. The ASTs generated by the parser should be strongly typed, to enable maximum utility of the TypeScript type system.

2.3   Approach

The approach to completing this project involves a few main steps. First, a prototype of the learning environment is made as an experimentation sandbox and proof of concept. JavaScript will be initially used as a stand-in for *Setanta* as it can be already executed on the browser. Using JavaScript and the environment prototype we explore options for abstracting out the single threading and non-blocking operations restrictions described above. We also experiment with different options for creating a graphical display that the user can manipulate and draw on.

   After experimenting with the sandbox we move to the design process of *Setanta*. We must decide on several important features of the language, in terms of syntax and semantics. The design process will involve creating several documents outlining the decision process in real-time. *These documents can be found in the supporting content of the project submission.* The linguistic properties of Irish will be contrasted with those of English and used to influence the syntax and semantics of the language.

   After a design is settled on we move to create the parser generator *tsPEG*. This is a key component and will be needed to create the parser for *Setanta*. *tsPEG* is worked on as a largely independent project, in fact, my supervisor has stated that he thinks that *tsPEG* is of sufficient independent interest to be a

final year project of its own. The creation process for *tsPEG* involves reading up on the existing state of the art parser generators and techniques and then creating a new generator with those ideas in mind, as well as the requirements of *Setanta*. *tsPEG* is created by using a bootstrapping process whereby a simple parser is made by hand for a very basic grammar, then this is used to self bootstrap further and further powerful features until we have created an expressive, state of the art parser generator.

tsPEG is then used to create the parser for *Setanta*, and work on implementing *Setanta* can begin. Like *tsPEG*, *Setanta* is created by first implementing the basic features of a PL, variables, loops, function calls. We then move on to implementing more and more complex features including classes, inheritance, closures, first-order functions and concurrency support. As we create *Setanta* we replace the JavaScript in the learning environment with *Setanta*, thus creating the full end-to-end product.

After-creation of *Setanta*, we return to the learning environment, adding a backend that allows users to save their code and send it to others. We improve the UI and UX of the website and optimise the concurrency abstractions as we prepare it for release.

2.4   Evaluation Metrics

The main strategy to evaluate this project is to assess the degree of completion of the main goals. We will evaluate how successful the project was by examining the level of success that was attained in completing the goals. The goals for *Setanta* include creating a new, modern programming language in Irish, evaluating how it differs from languages designed in English, how beginner-friendly it is, and how much interest there is in the language from the community at large. The domain of *Setanta* is education, and as such, performance is not our biggest concern. However, we should ensure that the language is fast enough to not be noticeably a significant problem.

For *tsPEG*, the goals are to create a powerful parser generator for TypeScript. *tsPEG* should be powerful enough to build a real programming language with. It should be capable of self-hosting its own input parser. *tsPEG* should be based on novel ideas in parser technology, and be highly configurable to different projects. *tsPEG* aims to have powerful syntax error reporting and recovery and to be highly strongly typed. We would also like *tsPEG* to be of interest to the parser community, hopefully finding use in other projects.

The learning environment has many goals. It must be an accessible website where users can write and execute *Setanta* code. The website should allow users to experiment with *Setanta*, both through classical text IO programming and by allowing them access to an API to draw and manipulate shapes and objects on a graphics display. It should be a beginner-friendly website so that it draws the interest of people who have not yet learned to program.

After the creation of these three components, we will return to these original goals and assess how well they were completed.

2.5   Project Achievements

In the completion of this project, several significant accomplishments were realized.

- **Successful creation and launch of *tsPEG***

  The TypeScript parser generator *tsPEG* was created successfully and released on the NPM (Node Package Manager) package repository. *tsPEG* was announced on some TypeScript forums and garnered some attention, peaking at 242 average weekly downloads for a period after its release.

tsPEG was used in the creation of *Setanta* to great success, and has found use in other projects in the community.

- **Implementation of *Setanta*.**

  As the main goal of the project, *Setanta* was developed almost exactly to the original design. It's a modern, powerful, expressive language with all the bells and whistles that we have come to expect, allowing for an imperative programming style, as well as a functional style. *Setanta* is a strong but dynamically typed language that can be executed both in the browser and locally on the command line through the NodeJS runtime. To test the expressive power of *Setanta* I wrote solutions to several programming problems from the *Advent of Code* problem set in *Setanta*. These solutions went on to be part of the test suite.

- **try-setanta.ie - The *Setanta* learning environment.**

  try-setanta.ie was created and hosted on the Google App Engine and satisfies all the requirements that were expected of it. Visiting try-setanta.ie the user can write and execute *Setanta* code. They can save their code and send it to their friends by the use of unique URLs assigned to each saved piece of code. try-setanta.ie also features the planned graphics display where the programmer graphically manipulates different shapes and objects.

- **Allowing Irish to affect the design of *Setanta*.**

  Part of the goals of designing *Setanta* was to allow the Irish language to influence its design, and by doing so to see what parts of industry-standard languages are subtly influenced by their design in English. *Setanta* has some syntactic constructs that were affected by the use of Irish as its "host" language. These effects highlight some of the Anglo-centric designs in today's languages, including in features that we consider to be almost universal. A significant example can be found in Section 5.2.3.

- **Abstraction out over the browsers concurrency issues.**

  Anyone who's written code for the browser will know the pain of the single-threaded, non-blocking world of JavaScript. Programs executed on the browser cannot launch multiple threads, or wait for operations to finish. Unless you use *Setanta*! The *Setanta* runtime abstracts out the JavaScript internal task queue and callback systems and allows the user to write code that waits for blocking operations such as user input, and to execute things concurrently such as running infinite rendering loops but still allowing keyboard inputs to be processed. How this was achieved is outlined in Section 5.2.5.

Chapter 3:   Background

3.1   Executing Code on the Browser

The only PL that standard web browsers (*Google Chrome, Mozilla Firefox, Safari, Edge, Internet Explorer, etc.*) can execute is JavaScript. There is a movement to bring other PLs to the browser via a technology called WebAssembly, but this has not fully come to fruition yet. In addition to this restriction on PLs, each browser instance limits executing code to a single, non-blocking thread.

```
1  // Would work if blocking allowed     1  // The real way to achieve this
2  function sleepFor100ms() {            2  function sleepFor100ms() {
3      print('Starting sleep')           3      print('Starting sleep')
4      sleep(100)                        4      setTimeout(() => print('After
5      print('After sleep')                      sleep'), 100)
6  }                                     5  }
```

Figure 3.1: Comparison of blocking and non-blocking code

3.1.1 Task Queue APIs

To overcome the non-blocking limitation of the execution thread the JavaScript environment exposes an API to a simple "task queue". The task queue is exactly what it sounds like, it's a queue of operations that the JavaScript environment maintains and performs in "first in first out" order (roughly). This task queue contains lots of important tasks, such as mouse and keyboard event processing, scrolling, rendering, button animations etc.

The JavaScript runtime allows users to add their own tasks to the queue by passing in **callbacks** to certain built-in functions. Callbacks are added to the task queue and will be executed at some later stage. When the JavaScript runtime completes a task, it moves to the next task in the queue.

By design JavaScript does not allow any blocking operations, e.g., `sleep`, or synchronous network calls or IO reads. The task queue API is therefore used to work around this.

Figure 3.1 shows a comparison of two JavaScript functions, one shows a function if blocking was allowed, and the other shows using the task queue APIs to get around it. Ideally these functions would print "Starting Sleep", wait 100ms, then print "After Sleep". However, sleeping would be a blocking action on the thread, so this is not allowed. In JavaScript, we must use the `setTimeout` function, which takes a callback function, and a time to wait before executing it.

Similarly, JavaScript uses an API to overcome the single-thread limitation. The developer is given access to certain functions that can use an extra thread for specific tasks, e.g., Fetching web content from a different website. These APIs also use callback functions to specify what should be done with the result of the functions, they perform their requested action and then place the callback function on the task queue.

This reliance on callback functions leads to something that the JavaScript community refers to as **Callback Hell**[14].

3.1.2 async / await and Promises

In an attempt to overcome callback hell, recent versions of JavaScript have introduced a concept known as "async / await".[7] This feature introduces what is effectively a syntactic sugar over a concept introduced previously, known as **Promises**.

Promises are a reference to a value that has not yet been computed, but will at some point be computed (resolved). These references are objects that can be manipulated just like any other object.

Mozilla describes Promises as

> A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. [8]

```
1  // Async addition using Promises      1  // Async addition using async / await
2  function doAsyncAddition() {              syntactic sugar.
3      return getA().then(a => {        2  async function doAsyncAddition() {
4          return getB().then(b => {    3      const a = await getA();
5              return a + b;             4      const b = await getB();
6          });                          5      return a + b;
7      });                              6  }
8  }
```

Figure 3.2: Promises vs async / await

Promises allow us to use an almost monadic programming style to write our asynchronous programs. However, in a similar way to Haskell's "do" notation, the async / await syntax allows us to write our asynchronous programs in a more imperative style. Figure 3.2 shows an example of using Promises directly vs using the async / await syntactic sugar to write the same function.

3.2 TypeScript

TypeScript is an open-source PL being developed by Microsoft, TypeScript is the primary PL that is used in the implementation of this project. Microsoft describes it as

> TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.[6]

TypeScript allows developers to write JavaScript programs with all the benefits of compile-time static type checking. As TypeScript is compiled to JavaScript it allows the user to use features of JavaScript that might not be widely supported yet, as it will compile them down to simpler features. Unfortunately, as TypeScript is still compiled to JavaScript, we are left with the same limitations as mentioned before.

3.3 PEG and CFG Grammars

Formal grammars are a model for specifying a certain set of strings. Noam Chomsky first formally researched these grammars in 1956[9] and shaped how PL syntax would be specified for years to come. PL designers use formal grammars to specify the syntax structure for their PL.

Context-free grammars (CFGs) are the family of grammars that have seen the most use in PL specification. However, recently there has been an uptick in the use of a similar grammar specification known as Parsing Expression Grammars (PEGs). These grammars are very similar to CFGs but are defined in such a way that they cannot be ambiguous. For the purposes of this project, PEGs provide a powerful way of formalising the syntax of a language. In fact, it is conjectured that PEGs are more powerful than CFGs, but this remains unproven[10].

3.4 Parser Generators

A parser generator is a program that, given a description of some formal grammar, outputs a parser for that grammar. Parser generators have existed since the earliest days of computer programming. One of the earliest major parser generators, YACC, was released in 1975 [5], and was based on LR parsing.

Several parser generators are available for JavaScript, including the excellent *pegjs* parser generator. However, no such parser generator was available for TypeScript, this is what prompted the creation of *tsPEG*.

3.5   VSO vs SVO languages

In linguistics, there is a concept known as word order, which is studying the order of how the different categories of words appear in sentences. One order that is studied is called the "constituent word order". The constituent word order is the order that a verb (V), object (O), and subject (S) appear in a sentence[11]. English is an SVO language, meaning that the subject comes first, then the verb, then the object. For example

> The man drove his car

In this sentence "the man" is the subject, "drove" is the verb, and "his car" is the object, and they appear in the order SVO.

Irish is a VSO language, meaning the verb comes first, then the subject, then the object. The same above sentence in Irish would be

> Thiomáin an fear a charr

In this case, "Thiomáin" (drove) is the verb, "an fear" (the man) is the subject, and "a charr" (his car) is the object.

Chapter 4:   Technical Problem Description

4.1   *tsPEG*

- *tsPEG* must be a fully functional parser generator. It should allow the user to specify a PEG grammar in some input format, and output a fully correct parser for that grammar.

- *tsPEG* should also be written in TypeScript, to allow it to self-host, and bootstrap. Hence *tsPEG* will run on the NodeJS JavaScript runtime.

- The generated parser must correctly match precisely the input grammar, and report any syntax errors encountered along the way.

- The parser must utilise the TypeScript type system to the fullest extent, making use of discriminated union types and enum types to create a strongly typed AST for the input grammar.

- The *tsPEG* input format should be expressive enough to allow users to specify complex and powerful grammars easily. It should support the standard set of PEG operators.

- *tsPEG* should allow the user to construct dynamic syntax trees through the use of computed properties.

4.2   *Setanta*

- *Setanta* should be fully usable with no knowledge of English, only Irish can be relied on.

- *Setanta* is required to be executable in the browser as well as locally.

- The syntax and semantics of *Setanta* must be directly influenced by the linguistic and cultural properties of the Irish language, a simple re-skin of an existing language with Irish keywords is not sufficient.

- Standard features of modern PLs must be present. The domain of *Setanta* is education, it must not be esoteric, learning *Setanta* should teach fundamental programming concepts.

- When the opportunity presents itself *Setanta* should make programming in it as accessible as possible. This includes using simplified vocabulary compared to mainstream languages, as well as not requiring the use of fadas (diacritics like áéíóú), as not all users can type these characters.

- *Setanta* should have performance as a low priority goal. It's not crucial for the language to be performant, but it will improve the experience.

Outside of these technical requirements, the design of *Setanta* is largely free for me to decide.

4.3   try-setanta.ie

The learning environment at try-setanta.ie also has some technical requirements. The purpose of try-setanta.ie is to act as a portal to using, learning and sharing *Setanta*. Therefore it must satisfy the following conditions:

- try-setanta.ie must enable visitors to the site to write and execute *Setanta* code.

- try-setanta.ie must have a clear layout and be simple to use.

- Some method of saving code and sharing it with friends needs to be included.

- As with *Setanta*, try-setanta.ie must not require any knowledge of English, only Irish can be used.

- Some sort of graphical display must be included and should have an API that is exposed to the *Setanta* runtime.

Chapter 5:   The Solution

I will discuss the design and implementation of each of the main components of the project separately.

5.1   *tsPEG*

5.1.1   High Level Design

tsPEG was built using TypeScript and the NodeJS JavaScript runtime. The NodeJS runtime allows us to execute JavaScript (and hence TypeScript) code locally.

The source code is hosted online at github.com/EoinDavey/tsPEG, and the *tsPEG* package is available on the NPM package repository at npmjs.com/package/tspeg.

tsPEG is self-hosting, meaning that the input parser for *tsPEG* was generated by *tsPEG*. The software architecture of *tsPEG* is given in Figure 5.1. The usage flow of using *tsPEG* is as follows. First, the user creates a grammar file, specifying the input grammar they want to generate a parser for. The syntax for
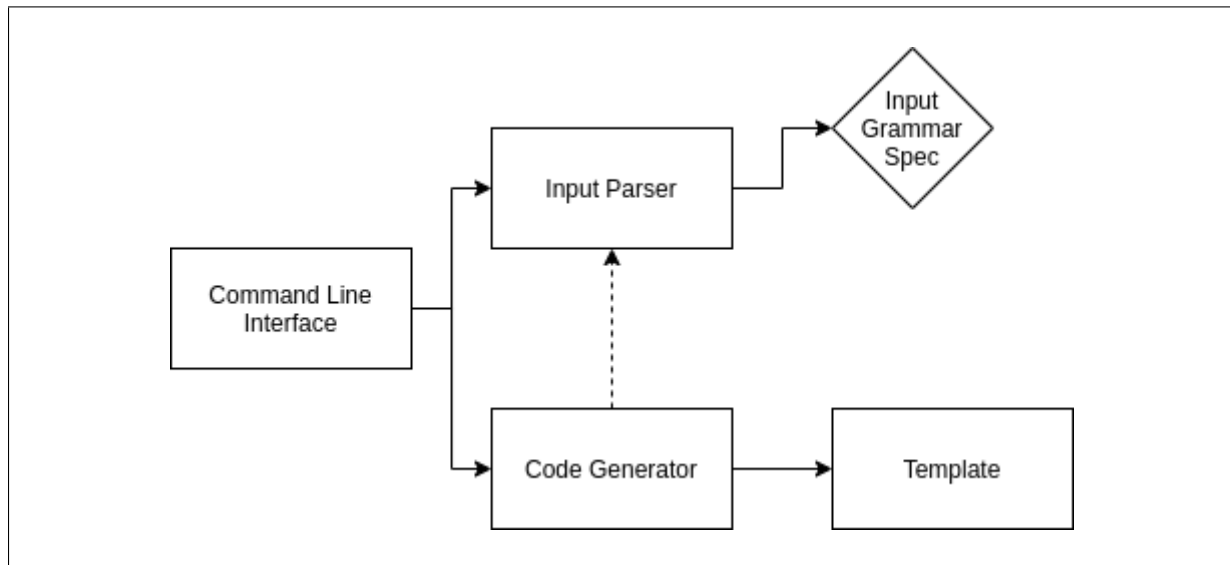
Figure 5.1: *tsPEG* High Level Design

```
1  SUM   :=  head=FAC  tail={ op='\+|-' sm=FAC }*
2  FAC   :=  head=ATOM tail={ op='\*|/' sm=ATOM }*
3  ATOM  :=  val=INT
4         |  '\(' val=SUM '\)'
5  INT   :=  val='[0-9]+'
```

Figure 5.2: *tsPEG* input grammar example

the grammar file follows a similar pattern to the familiar EBNF syntax for grammar specification. In this file, they specify the syntax rules, as well as names of AST fields, and computed properties. The *tsPEG* binary is then called and it is passed in the grammar file.

tsPEG's parser consumes the grammar file and generates an internal description of the grammar. The code generator then takes this grammar specification and generates parsing functions for each rule, as well as TypeScript type declarations and classes for each of the AST nodes. These functions and types are then packaged up into a template file and then written to disk.

5.1.2   Grammar specification

tsPEG uses a custom syntax to define grammars. Figure 5.2 contains an example of a grammar specification for simple arithmetic expressions like "1+2*3".

Grammars are defined by a sequence of grammar rules, for example,

$$\texttt{match} := \texttt{rule1} \,|\, \texttt{rule2} \,|\, \texttt{'a+'}$$

defines a new rule called `match`. *tsPEG*'s parser will first try to match `rule1`. If this match was not successful then it will try to match `rule2`. Finally, if both of those matches failed, then it will try to match the regex expression `a+`. If it can't match `a+` then `match` is considered to have failed. The *tsPEG* grammar also allows specification of computed properties, for example, Figure 5.3 defines a rule to match integer literals that stores the value of the integer as a computed property.

Please see Appendix A for full documentation of *tsPEG*'s grammar, operators and features.

```
1  INT := literal='[0-9]+'
2        .value = number { return parseInt(this.literal) }
```

Figure 5.3: *tsPEG* computed properties example

```
1  GRAM        := header=HDR? rules=RULEDEF+
2  HDR         := '---' content='((?!---)(.|\n))*' '---'
3  RULEDEF     := _ name=NAME _ ':=' _ rule=RULE _
4  RULE        := head=ALT tail={_ '\|' _ alt=ALT }*
5              .list = ALT[] { return [this.head, ...this.tail.map((x) => x.alt)]; }
6  ALT         := matches=MATCHSPEC+ attrs=ATTR*
7  MATCHSPEC   := _ named={name=NAME _ '=' _}? rule=POSTOP _
8  POSTOP      := pre=PREOP op='\+|\*|\?'?
9              .optional = boolean { return this.op !== null && this.op === '?'}
10 PREOP       := op='\&|!'? at=ATOM
11 ATOM        := name=NAME !'\s*:='
12             | match=STRLIT
13             | '{' _ sub=RULE _ '}'
14 ATTR        := _ '.' name=NAME _ '=' _ type='[^\s\{]+' _ '\{'
15    action='([^\{\}\\]|(\\.))*'
16 '\}'
17 NAME        := '[a-zA-Z_]+'
18 STRLIT      := '\'' val='([^\'\\]|(\\.))*' '\''
19 _           := '\s*'
```

Figure 5.4: *tsPEG* meta-grammar definition

5.1.3   Bootstrapping

When developing *tsPEG*, first a simple input parser was written by hand, supporting only the most basic syntax. A code generator was written that could take in the AST from this simple grammar and output a parser for it. This first version acts as the foundation from which *tsPEG* was bootstrapped. It had no extra operators, only the very foundational capabilities to define simple PEG grammars.

The meta-grammar for the input grammar syntax was subsequently written in this basic syntax. The parser generator was then ran on this grammar, this replaced the handwritten parser with a new generated one, self-hosting itself and opening itself up to bootstrapping. An interesting property of this process is that the original hand-written parser was destroyed, yet the project only functions because it did exist at some point in the past.

This new self-hosting generator was then used to add more and more features and operators to itself, eventually resulting in the full self-describing meta-grammar for the *tsPEG* input syntax in Figure 5.4.

5.1.4   Syntax Error reporting

The parsers generated by *tsPEG* must be capable of accurately reporting syntax errors in input strings. To achieve this an error reporting technique known as "farthest failure" is used. This approach is based on the approach outlined in the paper "Error handling in PEG Parsers"[13]. The idea is that the PEG parser should descend as far as possible into the input string while maintaining a valid AST, and report a syntax error at the farthest point reached in the string.

tsPEG uses this approach to generated and return `SyntaxError` objects. These objects contain the exact position (line and character) of the error, as well as a full list of possible symbols that could have been in that position that would have allowed the parser to continue.
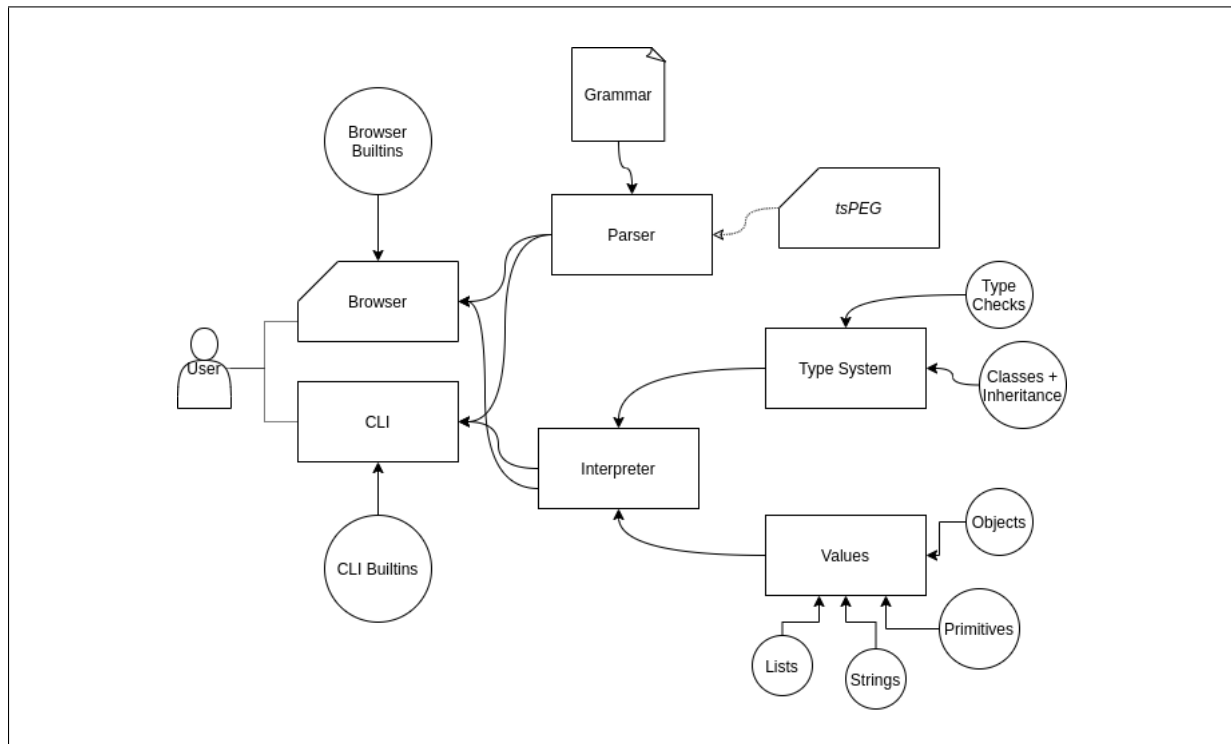
Figure 5.5: *Setanta* High Level Design

5.2 *Setanta*

5.2.1 High Level Design

Setanta is also written in TypeScript. The code structure of *Setanta* allows it to be run as a local REPL through the NodeJS engine, but also in the browser. A high-level architecture diagram is given in Figure 5.5. The high level structure of the *Setanta* interpreter was inspired by the high level structure of the Java tree walk interpreter of the "Lox" PL outlined in the book "Crafting Interpreters"[16], adapted to the needs of this project.

The *Setanta* runtime uses the *tsPEG* parser generator to generate its parser. Both the browser and CLI execution environments import the same interpreter and parser but provide their own set of built-in functions. Each runtime environment passes some input text to the parser, the parser returns an AST or a syntax error object. This AST can then be passed to the interpreter with the relevant builtins. The interpreter can then be stopped and started asynchronously from the main executing thread.

5.2.2 Syntax

The syntax of *Setanta* is new but should feel familiar to most people. It has been designed to be simple and approachable.

Setanta programs, like most imperative languages, consist of a sequence of statements. Some important *Setanta* features are outlined below:

- **Variable declarations**

    In *Setanta*, variables are declared using the := operator and can be re-assigned using the classic = operator. The distinction is to provide a clear lexical difference between variable declaration and reassignment.

- **Loops + Conditionals**

  Setanta supports the classic conditional execution structure of "if, else if, else". This is mostly a direct translation into Irish. However, it should be noted that no bracketing is required around the expression.

  Listing 5.1: Setanta conditionals

```
1  má x == 0
2      scríobh('Tá x cothrom le 0')
3  nó má x == 1
4      scríobh('Tá x cothrom le 1')
5  nó
6      scríobh('Tá x níos mo ná 1')
```

  Setanta supports two main types of loops, "le idir" loops that allow the user to specify the start and end of the loop, and "nuair-a" loops, which are the familiar while loops.

  Listing 5.2: Setanta loops

```
1  i := 0
2  le i idir (0, 10)
3      i = i + 1
4  x := 0
5   nuair-a  x < 10
6       x = x + 1
```

- **Classes + Functions**

  Setanta supports declaring new classes, with methods, as well as functions on their own.

  Listing 5.3: Setanta classes

```
1  creatlach Person ó Animal {
2      gníomh nua(name) {
3          name@seo = name
4      }
5      gníomh speak() {
6          scríobh('Hi, My name is ' + name@seo)
7      }
8  }
```

- **Literals**

  Setanta supports literals for integers, booleans, null, strings and lists.

  Listing 5.4: Setanta literals

```
1  a := 500
2  b := 'Dia duit domhan'
3  c := [1,2,3,4, fíor ]
4  d := fíor != breag
5  c := neamhní
```

Please see Appendix B for a full description of the syntax and semantics of *Setanta*. The 20 page limit on the report means the full description cannot be directly included.

5.2.3 The @ operator

One of the most important features of *Setanta*'s syntax is the @ operator. The @ operator is where the influence of the Irish language on the design of *Setanta* is felt most clearly. In fact, the @ operator is a simple innovation that addresses 2 distinct differences between Irish and English.

The @ operator is the lookup operator, it is used to reference member fields and methods of objects. It is functionally equivalent to the classic dot ('.') operator in C, C++, Java, Python, JavaScript etc. The subtle difference is the order, both the @ operator and the '.' operator are binary operators, but the @ operator flips its arguments compared to the '.' operator.

In the standard usage of the dot operator, the expression `a.b` refers to the member "b" of the object "a". In *Setanta* `a@b` refers to the member "a" of the object "b", the order has been reversed.

This change seems unimportant, however, it breaks 2 deep and subtle links between English and the standard way we look at OOP programming languages.

Firstly, as discussed in the technical background, English is an SVO language, meaning that in English sentences, the subject comes first, then the verb, then the object. This property of English is directly reflected in the design of the classic dot operator.

The usage of the dot operator for member lookup gives rise to expressions like `man.goTo(shop)`, or `dbClient.query(q)`, these expressions implicitly put the subject first, then the verb, then the object, directly mimicking the SVO structure of the host language.

By using the @ operator in *Setanta* we instead get expressions like `goTo@man(shop)` or `query@dbClient(q)`, these expressions put the verb first, then the subject, then the object, reflecting the VSO structure of Irish. This subtle connection between a simple operator like '.' and the linguistic properties of English is not apparent at first glance.

There is a second connection between English and the lookup operation that is broken by the @ operator. In English when talking about possession relationships ("has-a" relationships in database terms), we list the entities in decreasing order of possession, e.g. *"The man's car door window pane"*. The order of possession here is

$$man \rightarrow car \rightarrow door \rightarrow window \rightarrow pane$$

In direct contrast to this, in Irish we list possession in increasing order, meaning we start at the bottom of the possession relationship and work our way up. We would instead list

$$pane \leftarrow window \leftarrow door \leftarrow car \leftarrow man$$

This linguistic difference between Irish and English is again addressed by the @ operator. The standard '.' operator reflects the English language structure, `man.car.door.window.pane`, but in *Setanta*, we use the Irish ordering `pane@window@door@car@man`. This is another connection between English and the standard design of PLs that is easily overlooked.

5.2.4 Semantics

The semantics of *Setanta* will be familiar to most users, it's an imperative, strongly dynamically typed language. *Setanta* supports a gauntlet of modern features including first-class functions, inheritance, event-based concurrency, and automatic memory management. Although *Setanta* is an imperative language, it does support a more functional style of programming as well.

Please see Appendix B for a full description of the syntax and semantics of *Setanta*.

5.2.5   Blocking operations & Concurrency

As discussed in the technical background section, the JavaScript runtime, whether it be NodeJS or a browser, only allows usage of a single thread and no blocking operations. However, *Setanta* allows the user to write and use blocking functions. Figure 5.6 shows a side by side comparison of equivalent programs in JavaScript and Setanta, it's clear that the Setanta program is much simpler due to allowing the program to block.

```
            Listing 5.5: JavaScript                          Listing 5.6: Setanta
1  setTimeout(() => {                          1  coladh(100)
2      console.log('sleep1');                  2  scríobh('sleep1')
3      setTimeout(() => {                      3  coladh(100)
4          console.log('sleep2');              4  scríobh('sleep2')
5          setTimeout(() => {                  5  coladh(100)
6              console.log('sleep3')           6  scríobh('sleep3')
7          }, 100);
8      }, 100);
9  }, 100);
```

Figure 5.6: Equivalent code in JavaScript + Setanta

If you open the JavaScript console on any browser and enter **while**(**true**){} the browser tab will cease to be functional, i.e no buttons or text-boxes or animations will work anymore. This is because the JavaScript engine is single-threaded, the thread is busy computing **while**(**true**){} and cannot process any other events.

In contrast, if you write **nuair-a** fíor {} in *Setanta*, the browser tab will remain responsive. This is due to the implicit concurrency of all *Setanta* functions. *Setanta* manipulates the JavaScript engine task queue to ensure that execution time is shared between the main browser thread actions and executing the *Setanta* code.

How this was achieved

As discussed in the technical background, Recent versions of JavaScript include a technology known as "Promises". These values are stand-ins for future values that are yet unknown. The `Interpreter` class in *Setanta* makes heavy use of Promises, to allow the execution of the program to be suspended at any moment. By allowing suspension of the interpreter, we allow the code to appear to block, while under the hood, the thread has not technically blocked.

This idea works in theory, but in practice, an interesting problem was observed. As outlined in the technical background, JavaScript engines use task queues in the runtime to manage callbacks. Promises use this task queue to enqueue the operations that should be performed on the unknown values that they are a proxy for. However, unlike other task queue APIs exposed to the JavaScript runtime, Promises use something called the "microtask queue". The normal task queue is referred to then as the "macrotask queue". This distinction is little known, as it is usually of very little importance. However, the difference is crucial to *Setanta*'s implementation.

When the JavaScript runtime is deciding what task to execute next, it always chooses a task from the microtask queue over any macrotask, as they are usually short. However, when Promises have been used to construct a Turing complete language like *Setanta*, we encounter the problem that while the program is running, it keeps the microtask queue full, never allowing a macrotask to be run.
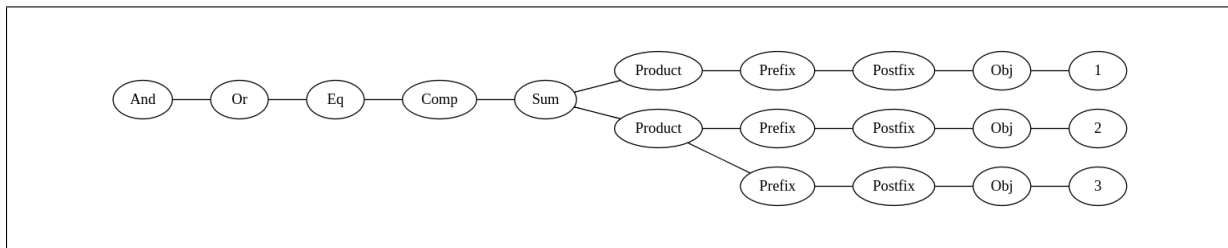
Figure 5.7: Uncompressed tree for "1 + 2 * 3"

Macrotasks are important to the JavaScript runtime, and the browser especially, things like animations, scrolling, rendering, keyboard events, button presses etc. are all macrotasks. Therefore by blocking the macrotask queue, we are preventing all these important operations from happening.

I experimented with forcing all the Promises to use the macrotask queue, rather than the microtask queue. But it turns out that the microtask queue is much faster than the macrotask queue, so this resulted in massive slowdowns. A happy medium had to be found. My eventual solution was to allow some finite amount of microtasks to be used before a macrotask should be used instead. After experimentation on a few platforms, I found a nice number to be around 5000. Meaning that after every 5000 operations performed by the *Setanta* runtime, it enqueues the next operation on the macrotask queue, allowing other macrotasks to be processed before the execution continues.

5.2.6   Tree Compression for performance

The arithmetic expressions in *Setanta* have a certain operator precedence, this precedence matches the usual order of operations that we are used to. For example in the expression 1 + 2 * 3, the multiplication is performed before the addition, giving the correct answer of 7.

The grammar for *Setanta* is carefully designed to ensure that all precedences are correct. It's not just multiplication and addition, there are 10 different levels of precedence in the grammar for *Setanta*. You can see the *Setanta* grammar spec (that *tsPEG* uses to generate the parser) in Appendix C.

A side effect of this many-layered grammar specification means that simple expressions like 1 + 2 * 3 generate very tall, but sparse syntax trees. In Figure 5.7 you can see the full syntax tree generated for 1+2*3. Most of these AST nodes contain no information. But in evaluating this expression directly we must traverse the entire tree. If the expression includes variables e.g., "a + b * c" we must do this every single time we evaluate it, we can't cache the result.

The solution that I came up with is to take advantage of *tsPEG*'s computed properties support. Notably, this tree compression problem is exactly the problem I added computed properties to *tsPEG* to allow me to solve.

The idea is that, at parse time, each node computes a "shortcut" attribute. If the node detects that it is not required, it computes a pointer to the next "important" node below it and returns that, shortcutting the node entirely, while maintaining the strong typing of the AST. This is all computed once when the AST is constructed. Following only the shortcut links for the expression 1 + 2 * 3, we instead get a compressed syntax tree seen in Figure 5.8. These shortcut attributes also compute if an operation is "quick", i.e. if it can be computed in a reasonably short amount of time, allowing us to skip the overhead of the Promises API for the evaluation.

This change caused a measurable speedup of around 19% when measured against the test programs. The test set of programs before this change would run in about 1.6s, but after would take about 1.3s.
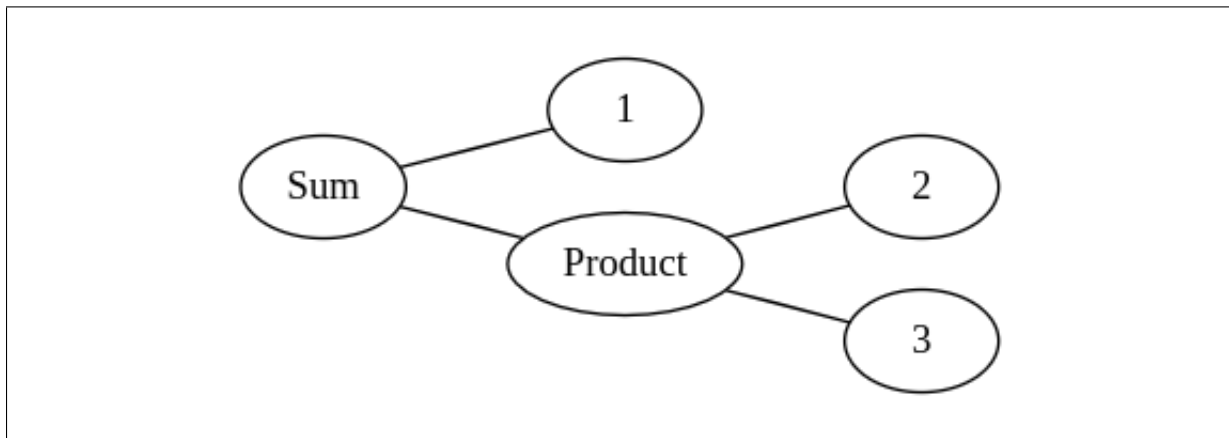
Figure 5.8: Compressed tree for "1 + 2 * 3"

5.2.7   Async / await slowness

As mentioned in the technical background, recent versions of JavaScript have introduced a new syntactic construct known as async/await. This feature is intended to get around the problem of writing blocking and asynchronous code in JavaScript. In theory, async / await is just syntactic sugar for the Promises API, and is recommended to use to improve readability and maintainability of code.

When writing the interpreter for *Setanta* to allow blocking operations, I took advantage of this new async / await feature, and it did lead to simple easy to read code. However, when I timed my tests, I found that the interpreter had been reduced to running at an absolute crawl. My test suite now took over 13 seconds to execute all programs.

As an experiment, I re-wrote the whole interpreter directly with the Promises API, manually resolving the async / await syntactic sugar, and then I executed my tests again. The results were shocking, the execution time for my tests was reduced down to almost 1 second, a 13x speedup!

This was highly shocking to me, as I had never heard anything about async / await being slower than the raw Promises API before, in fact, a cursory Google search will bring up countless results of people talking about how the speed of async / await is comparable or even faster than Promises.

5.2.8   Documentation

I have written documentation and tutorials for *Setanta*. They are not yet 100% complete, but are available in full in both English and Irish. The documentation is hosted at `https://docs.try-setanta.ie`.

The documentation requires syntax highlighted code, rather than hand colour the code snippets myself, I wrote a markdown pre-processor that would take the markdown documentation files, extract the *Setanta* code, parse it, and then replace the snippet with a highlighted version automatically.

5.3   The learning environment - try-setanta.ie

5.3.1   High Level Design

try-setanta.ie, like the other components, is written mainly in TypeScript. The website is built on Google's LitElement library. LitElement is a library to write custom re-usable web components, it's a descendant of the Polymer framework, which has since been deprecated. LitElement is used to achieve a Material Design aesthetic. The text editor component uses the open-source library CodeMirror.
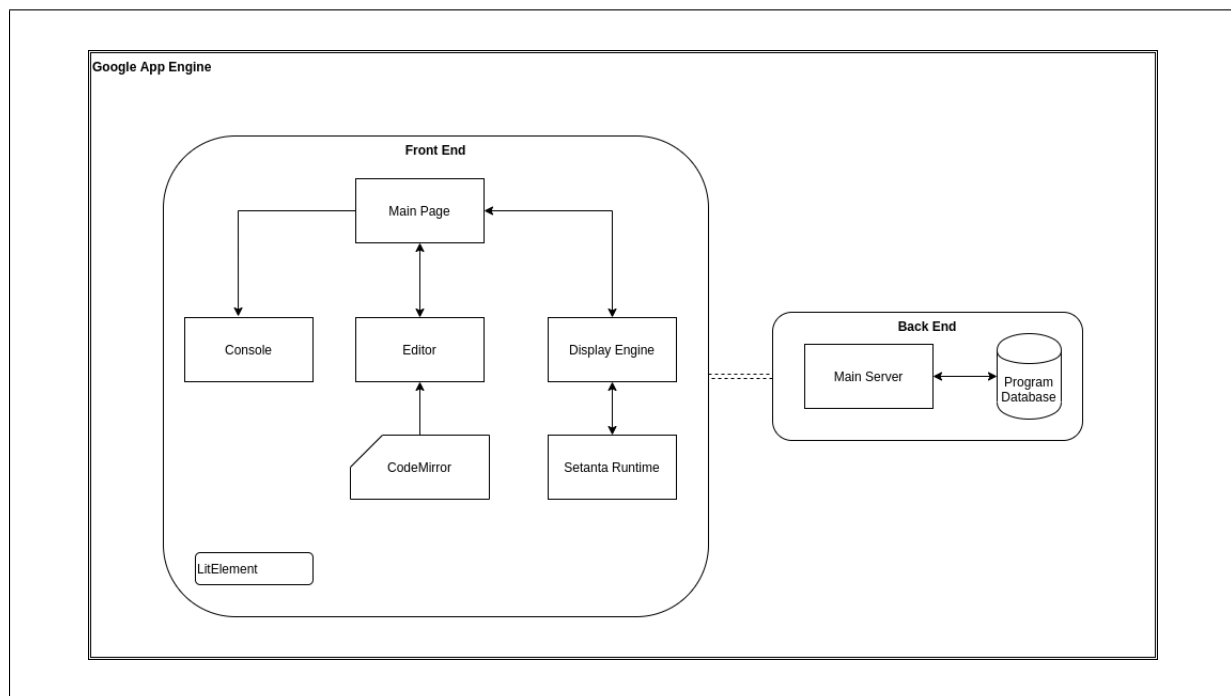
Figure 5.9: try-setanta.ie High Level Design

The backend for try-setanta.ie is written in Go and runs on the Google App Engine (GAE) cloud infrastructure platform. The backend for the site supports the ability to save and retrieve *Setanta* programs to the cloud. A high-level architecture outline for try-setanta.ie can be seen in Figure 5.9

try-setanta.ie is known to work with the Google Chrome, Mozilla Firefox, Safari and Microsoft Edge browsers, however, it does not work with Internet Explorer. This is due to Internet Explorer not supporting the Promises API directly. It should be possible to compile the code down to an earlier version of JavaScript that would work with IE, but this is of very low priority.

5.3.2   Graphics API

To better enable the educational experience on try-setanta.ie, the *Setanta* runtime has access to a graphics API. The graphics API allows the user to draw simple graphics primitives and animations. This API is exposed to the *Setanta* program as a global object. The user can access various fields and methods on this object to draw and manipulate shapes on the display. Appendix D contains screenshots of programs rendering the Sierpinski triangle, and a game of snake (Figure D.6, Figure D.3 respectively).

The graphics API also exposes methods to consume keyboard inputs, so the user can program simple games like *Snake* with ease. It also features a full-screen mode. You can find the *Snake* game demo at https://try-setanta.ie/EhEKBlNjcmlwdBCAgICgxt6JCg.

5.3.3   Saving Code

try-setanta.ie supports the ability to save a program to a specific URL. Clicking "Faigh nasc" (meaning "get link") redirects the user to a unique URL with which they can access their code. For example, the Sierpinski triangle example is available at: try-setanta.ie/EhEKBlNjcmlwdBCAgIDgycODCg.

This feature is handled by the Go backend on the Google App Engine. The programs are stored on the cloud Datastore, each program is identified by a unique integer. This unique integer is encoded into a string key, and this key is the URL where the code can be retrieved.

5.3.4   UI/UX Design

The UI/UX design for the try-setanta.ie website follows the Material Design[15] design philosophy. When try-setanta.ie is loaded, you are first brought to a simple landing page that directs the user to either visit the documentation, or open the editor, in English and in Irish.

The learning environment site consists of 3 main components: A console, a graphics canvas and a text editor. They are distributed according the material design method, placed on individual card like components. Contextually relevant actions such as "run code", "stop code" etc are displayed in floating action buttons. Care has been taken to ensure that only the correct buttons are displayed to the user (e.g. Only show the run button when code is not running, then switch to a stop button while it is running).

The positioning and layout of the buttons has been chosen to provide implicit context to their functionality. For example: The button to "Run in fullscreen mode" is not displayed by default, but when the user hovers over the "run" button, the run in fullscreen mode button appears above it. This tells the user that the fullscreen mode is an alternative to the usual method of running the code, without explaining it in words.

Screenshots of try-setanta.ie are included in Appendix D.

Chapter 6:   Evaluation

6.1   *tsPEG*

The *tsPEG* parser generator was implemented almost precisely to the original design. The correctness and effectiveness of *tsPEG* can be seen in a few ways.

- *tsPEG* is self-hosting, meaning that the input parser for *tsPEG* is generated by *tsPEG* itself. This would not be possible if *tsPEG*s parsers were not correct. The input grammar makes use of all the main features of *tsPEG*, so this acts as a rigorous self-referential test, whereby the whole project fails to even compile if it is not correct.

- The correctness of *tsPEG* is also guaranteed by a suite of unit tests and end to end tests.

- *tsPEG* generates the parser for the *Setanta* interpreter. This proves that *tsPEG* is capable of generating parsers for fully-featured programming languages, with complex and intricate grammars. The use of *tsPEG* in the *Setanta* build process serves as a full stress test of all the features available to it.

tsPEG has attracted attention from the TypeScript community, showing that it serves a useful purpose as a tool for the community. *tsPEG* has over 30 stars on Github and hit a peak of over 250 downloads in one week after it's initial release. *tsPEG* has found use in projects outside of *Setanta*, In fact on the submission deadline day (13th April) I received an email from someone thanking me for creating *tsPEG* and telling me they have put it to use to solve a complex problem. However, *tsPEG* still has many features that it would benefit hugely from, the main ones being improving the syntax error reporting capabilities and efficiency.

6.2   *Setanta*

As outlined in the solution section of *Setanta* (5.2), the implementation of *Setanta* was a success, it meets every goal that was set out originally for it. *Setanta* is a modern, novel, innovative language, that runs in the browser and is built from the bottom up in Irish. Lot's of programs have been written in *Setanta*, some by me, some by others.

The correctness of *Setanta* is ensured by a large suite of tests, including unit tests, and full end to end tests. *Setanta* contains over 1300 lines of test code alone. These tests ensure that every edge of language behaviour is correct, theses tests are run automatically by the continuous integration system with each commit. The current state of *Setanta*'s builds can be seen at travis-ci.com/EoinDavey/Setanta.

Setanta's main issue is its speed. As it is built on top of the JavaScript environment but requires blocking operations and concurrency, it means that *Setanta* has taken on an extremely large overhead. The result of this is that *Setanta* is notably slow at some tasks. However, the domain of *Setanta* is education, not high performance or production computing, so this is not a direct detractor from the quality of the project. I have taken several steps to keep the efficiency of *Setanta* at a high enough level that it's not noticeably slow. These steps are outlined in sections 5.2.6 and 5.2.7.

The fact that the same *Setanta* interpreter can be used both in the browser and in a local CLI shows the code is well abstracted and isolated, the main Interpreter class is very portable and can be loaded in a wide variety of environments.

Setanta has received a notable amount of attention online. Before I had even announced the project I had received requests to use the project from people who had stumbled upon the repository. I gave a talk on *Setanta* at SISTEM 2020 (A tech conference held in UCD) and it received high praise from many of the attendees.

6.3   try-setanta.ie

The try-setanta.ie learning environment has also been largely successful and meets most of the original design goals for it. try-setanta.ie is a simple to use website where the user can write and execute *Setanta* code, it has a good editor with syntax highlighting and a powerful graphics API. The implementation of try-setanta.ie is quite modern, taking advantage of the latest features of HTML5 and CSS3. try-setanta.ie is built on the web components technology, this means that the main components of the try-setanta.ie website are fully isolated, abstracted and re-usable.

try-setanta.ie doesn't meet the original design plan of having an extremely high-level graphics API. Instead, there is a lower-level graphics API. This decision was made as I found in the development process of this API that it was easier to understand a simple API that let you call a function to draw circles, squares etc. than a very high-level API where you would need to use classes and inheritance and other features to interact with the API.

6.4   Overall project

Overall I think this project has been a resounding success, especially for a project that is worth just 5 credits. The project works seamlessly from end to end, covering a wide range of technical skills, including compiler design, language design, back-end servers & databases, front end web-development, technical documentation writing, UI/UX design, cloud infrastructure/deployment, testing, build systems, bootstrapping, etc. Each part of this project presented its own set of interesting problems to be solved,

and each part comes together to create a complete final package. The project as a whole now contains over 15k lines of code, and between the 3 main components racked up over 450 commits.

Chapter 7:   Conclusion

7.1   Goals and project reception

The main goals for the project were to create an original Irish programming language, a learning environment, and a parser generator for TypeScript. In reference to these goals, the project has been fully successful.

At the time of writing, try-setanta.ie has been visited **over 1400 times by over 200 unique users** and the *Setanta* REPL has been downloaded over 100 times, these parts of the project haven't even been properly announced yet, only to people I directly have spoken to. *tsPEG* has been downloaded well over 700 times and has been maintaining an average of over 20 downloads a week.

The reception of the project has been fantastic, there is a lot of interest in Irish language programming out there and I have received highly positive feedback from many people. I gave a lightning talk about the project at SISTEM 2020, which was hosted in UCD and the support and interest I received was very high.

7.2   The value of non-English PLs

In creating *Setanta*, we intended for the design process to expose some links between standard PLs and English. As discussed in the section on the @ operator (5.2.3), even simple, extremely common features such as the dot (.) operator have subtle links to the English language. Although the links we found were not groundbreaking by any means, the existence of these links is important. Diversity of thought is hampered when all programming is passed through the lens of the English language. In this project, we've seen that it's not just the keywords that form this lens, but the deeper design of the PL too.

Setanta wasn't created to be a huge technical breakthrough, in fact, as an educational language, we were aiming for familiarity. *Setanta* was created to break the pattern of English PLs, and reveal the limitations that had been placed upon us by them.

While I was working on this project, a student at Carnegie Mellon had a similar idea and created the worlds first classical Chinese PL[12]. This shows that there is a general interest in non-English PLs in the world.

7.3   The future of the project

This project has been a passion project for me, I have put in far beyond 5 credits of work into this project because of that, and I fully intend to continue its development after the project has been submitted. I plan on a full release of version 1.0 of *Setanta* soon, after I complete the full documentation. I hope that *Setanta* finds use in CS education, there are many schools and institutions around the country that I hope will have some interest in it. I have met many people who have shown great enthusiasm for the project, I plan on opening the project up to the open-source community, hopefully including someone with much better Irish than me.

Bibliography

[1] CSO, *The Irish language*, 2017, https://cso.ie/en/media/csoie/releasespublications/documents/population/2017/7._The_Irish_language.pdf

[2] *Language regions of brain are operative in color perception*, Wai Ting Siok, Paul Kay, William S. Y. Wang, Alice H. D. Chan, Lin Chen, Kang-Kwong Luke, Li Hai Tan, Proceedings of the National Academy of Sciences May 2009, 106 (20) 8140-8145; DOI: 10.1073/pnas.0903627106

[3] Non-English based programming languages, Wikipedia [https://en.wikipedia.org/wiki/Non-English-based_programming_languages, Accessed on 2020/02/10]

[4] Graphic Organisers: A review of Scientifically Based Research, The Insitute for the Advancement of Research in Education at AEL [https://www.inspiration.com/sites/default/files/documents/Detailed-Summary.pdf, Accessed on 2020/02/10]

[5] Johnson, Stephen C. (1975). Yacc: Yet Another Compiler-Compiler

[6] TypeScript, Microsoft, [https://www.typescriptlang.org/, Accessed on 2020/02/13]

[7] ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017), [https://www.ecma-international.org/ecma-262/8.0/index.html, Accessed 2020/02/10]

[8] Mozzila Developer Network Documentation, Promises, [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, Accessed on 2020/04/13]

[9] Chomsky, Noam (Sep 1956). "Three models for the description of language". IRE Transactions on Information Theory. 2 (3): 113–124. doi:10.1109/TIT.1956.1056813.

[10] Ford, Bryan p (2004). "Parsing Expression Grammars: A Recognition Based Syntactic Foundation". Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. doi:10.1145/964001.964011. ISBN 1-58113-729-X.

[11] Song, Jae Jung (2012), Word Order. Cambridge: Cambridge University Press. ISBN 978-0-521-87214-0 & ISBN 978-0-521-69312-

[12] World's First Classical Chinese Programming Language, Charles Q. Choi, IEEE Spectrum, [https://spectrum.ieee.org/tech-talk/computing/software/classical-chinese, Accessed on 2020/03/02]

[13] Rüfenacht, Michael, Oscar Nierstrasz, and Jan Kurš. "Error handling in peg parsers." (2016).

[14] Callback Hell, [http://callbackhell.com/, Accessed on 2020/03/12]

[15] Material Design Introduction, [https://material.io/design/, Accessed on 2020/04/09]

[16] Crafting Interpreters, Bob Nystrom, [http://www.craftinginterpreters.com/, Accessed on 2020/04/09]

Appendices

Appendix A: *tsPEG* documentation

tsPEG is a PEG Parser generator for TypeScript. *tsPEG* takes in an intuitive description of a grammar and outputs a fully featured parser that takes full advantage of the TypeScript type system.

A.1   Installation

tspeg can be installed by running

```
npm install -g tspeg
```

A.2   Features

- Fully featured PEG support, more powerful than CFGs.

- Infinite lookahead parsing, no restrictions.

- Regex based lexing, implicit tokenisation in your grammar specification.

- Tight typing, generates classes for all production rules, differentiable using discriminated unions.

A.3   CLI Usage

The CLI invocation syntax is as follows

```
tspeg <grammar-file> <output-file>
```

This generates a TypeScript ES6 module that exports a parser class, as well as classes that represent your AST.

A.4   Parser Usage

The parser exports a parse function that accepts an input string, and returns ParseResult object like this

```
class ParseResult {
    ast: START | null;
    err: SyntaxErr | null;
}
```

If the err field is non-null, then a syntax error was found, otherwise the AST is stored in the ast field.

A.5   Grammar Syntax

tsPEG grammars are specified with a simple syntax, similar to the classic EBNF syntax.

Grammars are composed of a sequence of rules, each rule defines what text that it should match, using regex literals, names of rules, and powerful operators like | for choice.

Each rule is defined by a name, followed by a := sign and then a "rule description". Rule descriptions are easiest seen by example here:

```
hello := 'Hello '
helloWorld := hello 'World'
helloChoice := hello 'Mars' | helloWorld
```

- The first line defines a rule `hello` which matches the string 'Hello' directly.

- The next line defines the rule `helloWorld`, this rule first matches our first rule `hello`, then matches the string 'World', i.e it matches the string "Hello World".

- The third line uses the | operator to make a choice between two options.

   1. First this rule will try to match the left hand side of the operator: `hello 'Mars'`, which as before will first match the rule `hello`, then the regex literal `Mars`.
   2. If this left hand side fails, we move on to trying the right hand side, which is just a reference to the `helloWorld` rule. In practice this means it either matches "Hello World" or "Hello Mars"

tsPEG **starts parsing with the first rule in the grammar** so to match the `helloChoice` rule we should either move it to the start or defined a `start` rule that points to it like this:

```
start := helloChoice
hello := 'Hello '
helloWorld := hello 'World'
helloChoice := hello 'Mars' | helloWorld
```

Putting this grammar into a file called "grammar.peg" and running

```
tspeg grammar.peg parser.ts
```

we get our parser for this simple grammar in the file "parser.ts". We can compile this (target at least es2015) and load it into NodeJS to test it. This can be seen in Figure A.1.

As you can see we get `null` error for 'Hello World', and 'Hello Mars', this means the match was successful. But when we try 'Hello Jupiter' we get an error object, and it lists the location of the error, and the expected matches at that location, namely 'Mars' or 'World'. You can see more about errors in the Syntax Error section.

Notably the `ast` field of the parse result is empty. This is because we haven't told *tsPEG* what elements of the grammar we would like to be returned. When we write a rule definition, we can specify the fields we'd like to save in the AST by assigning them with an = sign. For example, say we want to match a string that's the sum of 2 numbers, e.g. "2+3", "123+456", we'd like to save both side of the sum in our AST, we can do this by writing a grammar like the following. *Note that we had to write '\+' to escape the '+' symbol as the '+' symbol has special meaning in regex.*

```
eoin@tsPEG$ tspeg grammar.peg parser.ts
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> var P = require('./parser');
undefined
> var parser = new P.Parser('Hello World');
undefined
> parser.parse()
ParseResult { ast: { kind: 2 }, err: null }
> parser = new P.Parser('Hello Mars'); parser.parse()
ParseResult { ast: { kind: 3 }, err: null }
> parser = new P.Parser('Hello Jupiter'); parser.parse()
ParseResult {
  ast: null,
  err:
   SyntaxErr {
     pos: PosInfo { overallPos: 6, line: 1, offset: 6 },
     exprules: [ 'Mars' ],
     expmatches: [ 'Mars', 'World' ] } }
>
```

Figure A.1: Matching our simple grammar

```
sum := left=num '\+' right=num
num := '[0-9]+'
```

Running *tsPEG* on this input and testing the parser in node we can see that the parser result has saved the left and right numbers of our sum. Note that we didn't have to assign the result of the `[0-9]+` rule in the `num` rule. This is because rules that are just references to other rules are saved implicitly.

```
eoin@tsPEG$ tspeg grammar.peg parser.ts
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> var P = require('./parser');
undefined
> new P.Parser('2+3').parse()
ParseResult { ast: { kind: 0, left: '2', right: '3' }, err: null }
>
```

Figure A.2: Sum example

Now that we have saved the numbers of our sum, it's easy to write a program to compute the result of adding the numbers. Each rule that assigns results to the AST is exported as a class or interface with the same name as the rule, so we can just import the `sum` interface from the parser to write our function. For this grammar we get an interface like:

```
interface sum {
    left: string;
    right: string;
}
```
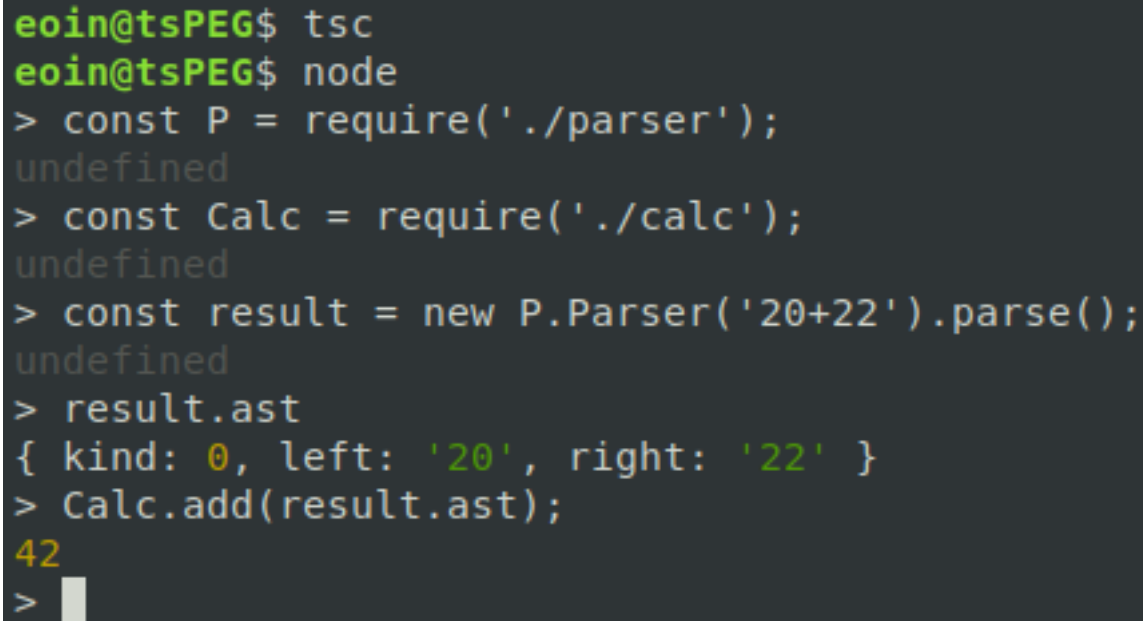
Allowing us to write our calculator function like:

```
import { sum } from "./parser";

export function add(ast: sum): number {
    return parseInt(ast.left) + parseInt(ast.right);
}
```

Calling this file "calc.ts" we can use it to calculate our sums:

Figure A.3: Calc screenshot

We can also use computed properties to calculate the result of the sum during the parsing process.

A.6 Operators

tsPEG supports a set of powerful operators to build complex grammars. The only operator we have seen so far is the | operator, which allows us to make choices between two rule expressions, however there are many more.

- The ? operator is used to make a match optional. For example in the rule

```
rule := 'I ' 'really '? 'love tsPEG'
```

The match for the 'really' string has been marked optional, so this rule can match either "I really love tsPEG", or "I love tsPEG".

- The + operator matches 1 or more copies of the match it's applied to, for example:

```
rule := 'It\'s a ' 'long '+ 'way away'
```

This rule matches "It's a long way away", "It's a long long way away", etc. for any amount of "long"s that's not 0. When this operator is used, a list of results is attached to the AST.

- The * operator is the same as the + operator but it allows zero matches.

- The ! is called the "*negative lookahead*" operator, and it does exactly what it says on the tin. This operator inverts the result of the match, meaning you can specify a rule by what it should not match. For example the rule `rule := 'The banned word is ' !'Macbeth' '[a-zA-Z]+'` This will match the phrase "The banned word is X" for any value of X, except when X is 'Macbeth'

- The & operator is called the "*postitive lookahead*" operator, this operator will change a match so that it will test for the match, and fail if it doesn't work, but it will not consume the input. This allows you to lookahead at what comes next in the string, but not to consume it.

A.7   Sub-rules

Inline sub-rules can be specified in *tsPEG*. These use the { and } brackets for grouping, and allow you to write smaller rules inline in a larger rule. For example

```
rule := 'start' {some optional part}? 'finish'
```

We want the part in the middle to be optional, so we wrap it in { and } and apply the ? operator. The {} brackets create a sub-rule. Note that if you want to assign the subrules values to the tree you do need to assign a name to it e.g. `rule := 'start' middle={some optional part}? 'finish'`.

A.8   Syntax Errors

A `SyntaxErr` object is composed of two fields, a `pos` field with the position of the error, and `expmatches` which contains a list of expected matches.

```
class SyntaxErr {
    pos: PosInfo;
    expmatches: string[];
}
class PosInfo {
    overallPos: number;
    line: number;
    offset: number;
}
```

A.9   Computed Properties

As well as assigning parsing results to variables and storing them on the AST, *tsPEG* also allows you to create **computed properties**, which are fields on the AST that are computed when the parser is run.

Computed properties are added to a rule by appending a new expression after the rule description like

```
.<propertyname> = <type> { <code to calculate property> }
```

Returning to our sum example from earlier we had a grammar to match strings like "2+3", "40+200" etc.

```
sum := left=num '\+' right=num
num := '[0-9]+'
```

We can add computed properties to this to compute the value of this sum at parse time, instead of writing our own function to do it after. First we add a computed property to `num` to store the value of the number:
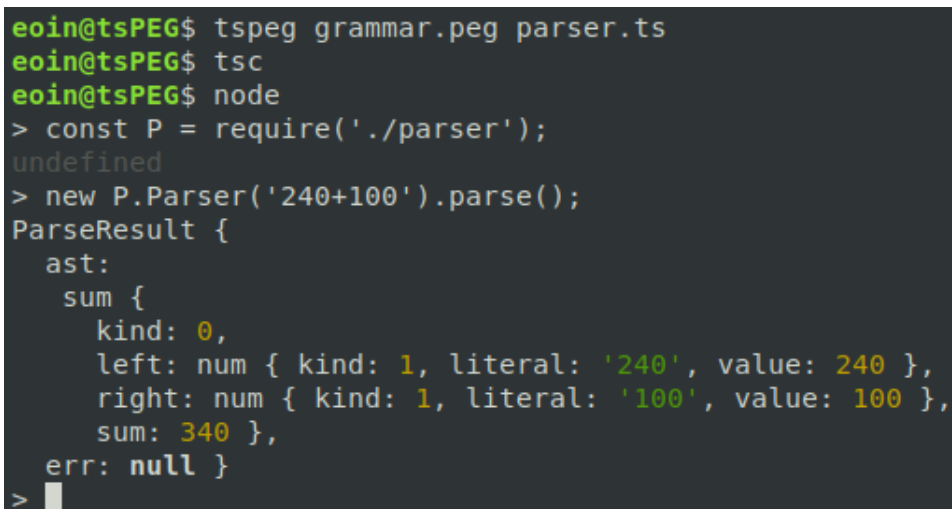
```
sum := left=num '\+' right=num
num := literal='[0-9]+'
        .value = number { return parseInt(this.literal); }
```

As you can see we've assigned the text match to the field `literal`, and added a computed property called `value` which is the `literal` field parsed as an integer.

Now we can add a computed property to `sum` to do the arithmetic, this is very simple

```
sum := left=num '\+' right=num
        .sum = number { return this.left.value + this.right.value }
num := literal='[0-9]+'
        .value = number { return parseInt(this.literal); }
```

We use the computed property of the `num` rule to compute our `sum` property. Let's see this in action! Let's try and parse the string "240+100".


```
eoin@tsPEG$ tspeg grammar.peg parser.ts
eoin@tsPEG$ tsc
eoin@tsPEG$ node
> const P = require('./parser');
undefined
> new P.Parser('240+100').parse();
ParseResult {
  ast:
   sum {
      kind: 0,
      left: num { kind: 1, literal: '240', value: 240 },
      right: num { kind: 1, literal: '100', value: 100 },
      sum: 340 },
  err: null }
>
```

Figure A.4: Computed property

As you can see the AST has a field called `sum` with the correct value of 340 in it. The `left` and `right` also have their computed `value` properties of 240 and 100.

A.10   Header

The introduction of computed properties means that now you might want to import some types or functions into your parser. Luckily you can specify a header in the file that will be inserted directly into the generated parser. Anything at the top of the grammar file between two lines of three dashes `---` will be inserted straight into the generated parser, allowing you to write grammars like

```
---
import { myFunc, myType } from "./mypackage";
---
rule := hello='Hello World'
       .value = myType { return myFunc(this.hello); }
```

A.11 Kind Checking

tsPEG uses discriminated unions to distinguish between types of AST nodes. Each AST node type has a field called `kind` which contains some value from the `ASTKinds` enum. This `kind` field can be used to differentiate between AST results.

Example

```
Choice := Word | Int
Word   := word='[a-z]+'
Int    := val='[0-9]+'
```

When writing a function to process the parse tree for this grammar, you can check if `kind ===` `ASTKinds.Word` or `kind === ASTKinds.Int` to see which rule was matched.

```
import { Choice, Parser, ASTKinds } from  "./parser";

function makeChoice(c: Choice) {
    if(c.kind === ASTKinds.Word) {
        console.log('Matched word ', c.word)
    } else {
        console.log('Matched int ', c.val)
    }
}
```

Kind names

The names of the ASTKinds enum entries vary.

- For simples rules like `Rule := name='regex'` the kind will be `ASTKinds.Rule`.

- For rules with multiple choices such as `Rule := choiceA='regexA' | choiceB='regexB'` the `kind` will either be one of `ASTKinds.Rule_1` or `ASTKinds.Rule_2` depending on which rule was matched. In general they are of the form `ASTKinds.<RuleName>_N` for the Nth choice.

- Rules that directly reference a different rule like `rule := otherrule` don't get their own AST type, so inherit the `kind` from the other rule.

- Sub-rules are given the kind `ASTKinds.<ParentRule>_$N` for the Nth subrule of `ParentRule`. For example in the rule

```
                       Rule := sub={ name='regex' }
```

the `kind` for the sub-rule is `ASTKinds.Rule_$0`.

- If in doubt it's simple to inspect the generated parser file to find what the correct kind name is. The compiler will also be sure to tell you when you're wrong.

A.12 Position tracking

tsPEG supports a special match expression "@" for storing the positions of matches. Using @ you can store the current position of the parser.

Example

Returning to our addition of two numbers from earlier, which matches "12+56", "123+456" etc.

```
sum := left=num '\+' right=num
num := '[0-9]+'
```

If we want to store the location of the "+" operator we can use the @ match like this:

```
sum := left=num pluspos=@ '\+' right=num
num := '[0-9]+'
```

This stores the position of the parser into the AST before the '+' symbol, the generated interface for the AST looks like:

```
interface sum {
    left: string
    pluspos: PosInfo
    right: string
}
```

Appendix B: *Setanta* syntax and semantics

The syntax of *Setanta* is new, but should feel familiar to most people. It has been designed to be simple and approachable. It takes inspiration from C like languages, but has some new ideas of it's own. The grammar has been designed for unambiguity so no semicolons or similar construct are required.

Setanta programs, like most imperative languages, consist of a sequence of statements. Some important *Setanta* features are outlined below:

B.1 Semantics

B.1.1 Paradigm

Setanta is classic imperatively executed language, it's multi-paradigm as it supports objects and inheritance, as well as higher order first class functions.

B.1.2 Type System

The *Setanta* type system is a strongly, but dynamically typed system. In practice this means the variables can hold any type of value, and the type of a variable can change over time, but that functions and operators will not try to automatically cast types to other types. This is in stark contrast to JavaScript which is famed for how weak it's typing system is, leading to lots of unintuitive errors.

The primitive types in *Setanta* include the familiar set of primitives: numbers (floating and integral), strings and booleans.

Setanta also supports functions and objects as first class entities.

B.1.3 Scoping & Closures

Setanta, like most modern programming languages, is lexically scoped and supports closures. By syntactically distinguishing between variable initialisation and variable assignment (Section B.2.1), the lexical scoping is easy to visually understand.

Listing B.1: Closures

```
1   gníomh  adder(x) {
2       gníomh  f(y) {
3           toradh  x + y
4       }
5       toradh  f
6   }
7
8   addfive := adder(5)
9   scríobh(addfive(2))
10  >-- Outputs 7
```

B.1.4   Inheritance

Setanta uses a traditional single dispatch, class based single inheritance system. In practice this means that a class A can inherit from a single class B. A inherits all of B's methods, including constructor. If A redefines any of Bs methods, including the constructor, it can access the methods of B using the tuis keyword.

B.2   Syntax

B.2.1   Variable declarations

In *Setanta* variables are declared using the := operator, and can be re-assigned using the classic = operator. The distinction is to provide a clear lexical difference between variable declaration and reassignment.

Listing B.2: Variable declaration and assignment

```
1  x := 0
2  x = x + 1
```

B.2.2   Literals

Setanta supports literals for integers, booleans, null, strings, lists and null.

Listing B.3: Setanta literals

```
1  a := 500
2  b := 'Dia duit domhan'
3  c := [1,2,3,4, fíor ]
4  d := fíor != breag
5  c := neamhní
```

B.2.3   Expressions

Setanta expressions are built from literals, identifiers, and a large selection of operators.

The familiar arithmetic operators are included, +,-,*,/,//,% meaning addition, subtraction, multiplication, division, integer division and modulus respectively. The operators ==, !=, <, >, <=, >= can be used for comparisons. (and) can be used for grouping.

Functions are called by using appending a bracket enclosed sequence of arguments to the function identifier e.g. function(arg1, arg2).

Lists and strings can be indexed with the traditional square bracket notation, arr[index].

Object access is done with the @ operator. If some object obj has a member or method a, this is expressed as a@obj. Note that this is inverse from the familiar dot (.) operator.

Listing B.4: Setanta expressions

```
1  2 + 3 − (5 ∗ 3)
2  4 >= 2
3  4 % 2 == 0
4  scríobh('hey')
5  array[1 + 2][0] + 'hey'
6  a@b@c@obj != 4
```

B.2.4 Assignment

As well as variables, lists and objects can be assigned to in the way you would expect

Listing B.5: Valid Setanta assignments

```
1  field@obj = 'dia duit'
2  arr[0][0] = 4 + 5
3  field@obj(arg1)[1] = 42
```

B.2.5 Conditionals

Setanta support the classic conditional execution structure of if → else-if → else. This is mostly a direct translation into Irish, as it uses the keyword **má** meaning "if", and the word **nó** meaning "or" However it should be noted that no bracketing is required around the expression.

Listing B.6: Setanta conditionals

```
1  má x == 0
2      scríobh('Tá x cothrom le 0')
3  nó má x == 1
4      scríobh('Tá x cothrom le 1')
5  nó
6      scríobh('Tá x níos mo ná 1')
```

B.2.6 Loops

Setanta supports two main types of loops, "le idir" loops that allow the user to specify start and ends to the loop, and "nuair-a" loops, which are the familiar while loops.

"le idir" take the form **le** i **idir** (a, b, c), which translates as roughly "with i between (a, b, c)". This is similar to Python's **for** i **in range**(a,b,c) loop, meaning the loop starts loops from a to b, with step size c. The current iteration would be available in the variable i inside the loop.

"nuair-a" translates roughly as "when". This loop is equivalent to a while loop from many C-family languages.

Listing B.7: Setanta loops

```
1  i := 0
2  le i idir (0, 10)
3      i = i + 1
4  x := 0
5   nuair-a  x < 10
6      x = x + 1
```

B.2.7 Functions

Functions in *Setanta* are referred to with the term "gníomh" meaning "action". They can constructed with the **gníomh** keyword, followed by a name and argument list. The **toradh** keyword can be used to return values from the function.

Listing B.8: Setanta function example

```
1  gníomh fibonacci(n) {
```

```
2        má  n  <=  1
3            toradh  1
4        toradh  fibonacci(n−1) + fibonacci(n−2)
5    }
```

B.2.8 Classes

Setanta supports declaring new classes, with methods, and a constructor. Classes can inherit from other classes using the keyword *ó*. Methods are declared as functions within the body of the class. A method with the name nua defines the constructor, "nua" translates as "new" in English. The keyword seo can be used to access member fields and methods of the class, "seo" is a direct translation of the familiar "this" keyword.

Listing B.9: Setanta classe example

```
1    creatlach  Person  ó  Animal  {
2        gníomh  nua(name)  {
3            name@seo  =  name
4        }
5        gníomh  speak()  {
6            scríobh('Hi, My name is ' + name@seo)
7        }
8    }
```

Appendix C: Setanta grammar specification

Below is the full specification for the *Setanta* syntax, as used by *tsPEG* to generate its input parser.

```
1  ---
2  import { Environment } from "./env";
3  import { callFunc, idxList, Value } from "./values";
4  import { unescapeChars } from "./litreacha";
5  import * as Asserts from "./asserts";
6  import * as Checks from "./checks";
7  import { orBinOp, orQuickBinOp, andBinOp, andQuickBinOp,
8      binOpEvalFn, binOpQuickEvalFn } from "./binops";
9  import { objLookupsEval, postfixArgsEval, csArgsEval, prefEval, EvalFn } from "./
       evals";
10 import { qEvalToEval } from "./evals";
11 import * as Quick from "./quickevals";
12 ---
13 Program      := stmts=AsgnStmt* _
14 AsgnStmt     := IfStmt
15              | BlockStmt
16              | NuairStmt
17              | LeStmt
18              | CCStmt
19              | BrisStmt
20              | CtlchStmt
21              | GniomhStmt
22              | ToradhStmt
23              | AssgnStmt
24              | DefnStmt
25              | Expr
26 NonAsgnStmt := IfStmt
27              | NuairStmt
28              | LeStmt
29              | CCStmt
30              | BrisStmt
31              | ToradhStmt
32              | BlockStmt
33              | AssgnStmt
34              | Expr
35 IfStmt       := _ 'm[áa]' \&gap expr=Expr \&gap stmt=NonAsgnStmt elsebranch={_ 'n[oó]
       ' \&gap stmt=NonAsgnStmt}?
36 BlockStmt    := _ '{' blk=AsgnStmt* _ '}'
37 NuairStmt    := _ 'nuair-a' expr=Expr \&gap stmt=NonAsgnStmt
38 LeStmt       := _ 'le' \&gap id=ID _ 'idir' _ '\(' strt=Expr _ ',' end=Expr step={_ ',
       ' step=Expr}? _ '\)' stmt=NonAsgnStmt
39 DefnStmt     := _ id=ID _ ':=' _ expr=Expr
40 AssgnStmt    := _ lhs=Postfix _ op=AsgnOp _ expr=Expr
41 GniomhStmt   := _ 'gn[íi]omh' \&gap id=ID _ '\(' args=CSIDs? _ '\)' _ '{'
42     stmts=AsgnStmt*
43 _ '}'
44 CtlchStmt    := _ 'creatlach' \&gap id=ID tuis={_ 'ó' id=ID}? _ '{'
```

35

```
45        gniomhs=GniomhStmt*
46  _  '}'
47  BrisStmt      := _  'bris'
48  CCStmt        := _  'chun-cinn'
49  ToradhStmt    := _  'toradh' \&gap exp=Expr?
50  Expr          := And
51  And           := head=Or tail={_ '\\&' trm=Or}*
52                   .evalfn = EvalFn { return andBinOp(this); }
53                   .qeval = Quick.MaybeEv { return andQuickBinOp(this); }
54  Or            := head=Eq tail={_ '\|' trm=Eq}*
55                   .evalfn = EvalFn { return orBinOp(this) }
56                   .qeval = Quick.MaybeEv { return orQuickBinOp(this); }
57  Eq            := head=Comp tail={_ op='[!=]=' trm=Comp}*
58                   .evalfn = EvalFn { return binOpEvalFn(this) }
59                   .qeval = Quick.MaybeEv { return binOpQuickEvalFn(this); }
60  Comp          := head=Sum tail={_ op=Compare trm=Sum}*
61                   .evalfn = EvalFn { return binOpEvalFn(this) }
62                   .qeval = Quick.MaybeEv { return binOpQuickEvalFn(this); }
63  Sum           := head=Product tail={_ op=PlusMinus trm=Product}*
64                   .evalfn = EvalFn { return binOpEvalFn(this) }
65                   .qeval = Quick.MaybeEv { return binOpQuickEvalFn(this); }
66  Product       := head=Prefix tail={_ op=MulDiv trm=Prefix}*
67                   .evalfn = EvalFn { return binOpEvalFn(this); }
68                   .qeval = Quick.MaybeEv { return binOpQuickEvalFn(this); }
69  Prefix        := _ op='-|!'? pf=Postfix
70                   .evalfn = EvalFn { return prefEval(this); }
71                   .qeval = Quick.MaybeEv { return Quick.qPrefEval(this); }
72  Postfix       := at=ObjLookups ops=PostOp*
73                   .evalfn = EvalFn { return postfixArgsEval(this); }
74                   .qeval = Quick.MaybeEv { return Quick.qPostfixArgsEval(this); }
75  ObjLookups    := _ attrs={id=ID '@' !wspace}* root=Atom
76                   .evalfn = EvalFn { return objLookupsEval(this); }
77                   .qeval = Quick.MaybeEv { return Quick.qObjLookupsEval(this); }
78  PostOp        := '\(' args=CSArgs? _ '\)' | '\[' expr=Expr '\]'
79  Atom          := _ '\(' trm=Expr '\)'
80                   .evalfn = EvalFn { return (env: Environment) => this.trm.evalfn(env);
                         }
81                   .qeval = Quick.MaybeEv {
82                       const childF = this.trm.qeval;
83                       return childF === null ? null : childF.bind(this.trm);
84                   }
85                 | ID
86                 | Litreacha
87                 | Int
88                 | Bool
89                 | Neamhni
90                 | ListLit
91  LSpec         := id=ID ops=PostOp*
92  ListLit       := _ '\[' els=CSArgs? _ '\]'
93                   .evalfn = EvalFn {
94                       return (env: Environment) => this.els ? this.els.evalfn(env) :
                           Promise.resolve([]);
95                   }
96                   .qeval = Quick.MaybeEv { return Quick.qListLitEval(this); }
97  CSArgs        := head=Expr tail={_ ',' exp=Expr}*
```

```
 98  |                     .evalfn = (env:Environment)=>Promise<Value[]> { return csArgsEval(
     |                         this); }
 99  |                     .qeval = ((env:Environment)=>Value[])|null { return Quick.qCSArgsEval
     |                         (this); }
100  | CSIDs          := head=ID  tail={_ ',' id=ID}*
101  | ID             := _ !{Keyword gap} id='[a-zA-Z_áéíóúÁÉÍÓÚ]+'
102  |                     .evalfn = EvalFn { return qEvalToEval(Quick.qIdEval(this.id)); }
103  |                     .qeval = Quick.EvalFn { return Quick.qIdEval(this.id); }
104  | Bool           := _ bool='f[ií]or|br[eé]ag'
105  |                     .evalfn = EvalFn { return qEvalToEval(Quick.qBoolEval(this.bool)); }
106  |                     .qeval = Quick.EvalFn { return Quick.qBoolEval(this.bool); }
107  | Neamhni        := _ 'neamhn[ií]'
108  |                     .evalfn = EvalFn { return () => Promise.resolve(null); }
109  |                     .qeval = Quick.EvalFn { return () => null; }
110  | Int            := _ int='-?[0-9]+(?:\.[0-9]+)?'
111  |                     .evalfn = EvalFn { return qEvalToEval(Quick.qIntEval(this.int)); }
112  |                     .qeval = Quick.EvalFn { return Quick.qIntEval(this.int); }
113  | Litreacha      := _ '\'' val='([^\'\\]|\\.)*' '\''
114  |                     .evalfn = EvalFn { return qEvalToEval(Quick.qLitreachaEval(this.val))
     |                         ; }
115  |                     .qeval = Quick.EvalFn { return Quick.qLitreachaEval(this.val); }
116  | _              := wspace*
117  | wspace         := '(?:\s|>--(?:(?!--<).)*(--<|\n|$))'
118  | gap            := { wspace | '[^a-zA-Z0-9áéíóúÁÉÍÓÚ]' }+ | '$'
119  | PlusMinus      := '\+|-'
120  | AsgnOp         := '=|\+=|\*=|-=|%=|\/='
121  | MulDiv         := '\*|\/\/|%|\/'
122  | Compare        := '<=|>=|<|>'
123  | Keyword        := 'm[áa]' | 'n[oó]' | 'nuair-a' | 'f[ií]or|br[eé]ag'
124  |      | 'gn[ií]omh' | 'chun-cinn' | 'neamhn[ií]' | 'toradh' | 'creatlach'
```

Appendix D: try-setanta.ie Screenshots

D.1 Desktop Screenshots

Below are the following desktop screenshots:

- The landing page for try-setanta.ie: Figure D.1. Exists to direct users to either the documentation/-tutorial or open the learning environment.

- The default editor: Figure D.2. This is the environment that loads by default, the loaded code, if run, draws a version of the main logo.

- A game of snake in progress: Figure D.3.

- The documentation site (Irish section): Figure D.4.

Then follows the following mobile screenshots:

- The landing page for try-setanta.ie: Figure D.5.

- The Sierpinski triangle demo running on mobile: Figure D.6.

- The Setanta documentation (English): Figure D.7.
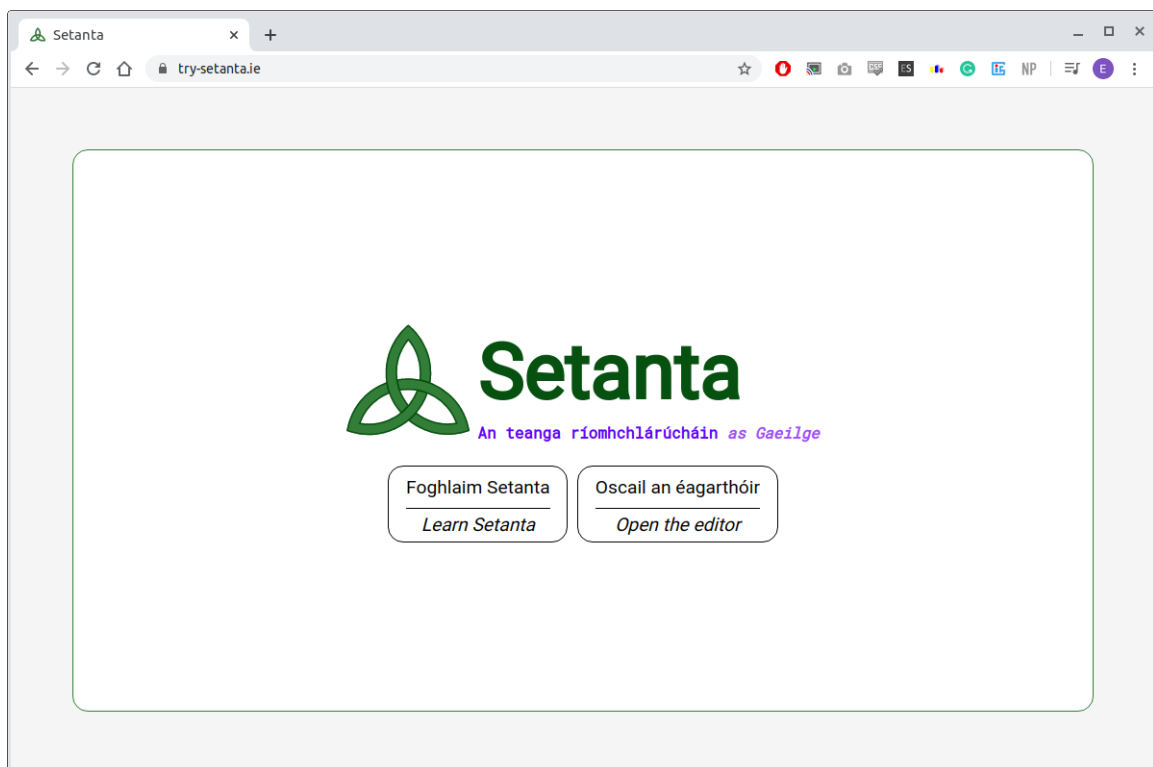
Figure D.1: Desktop landing page
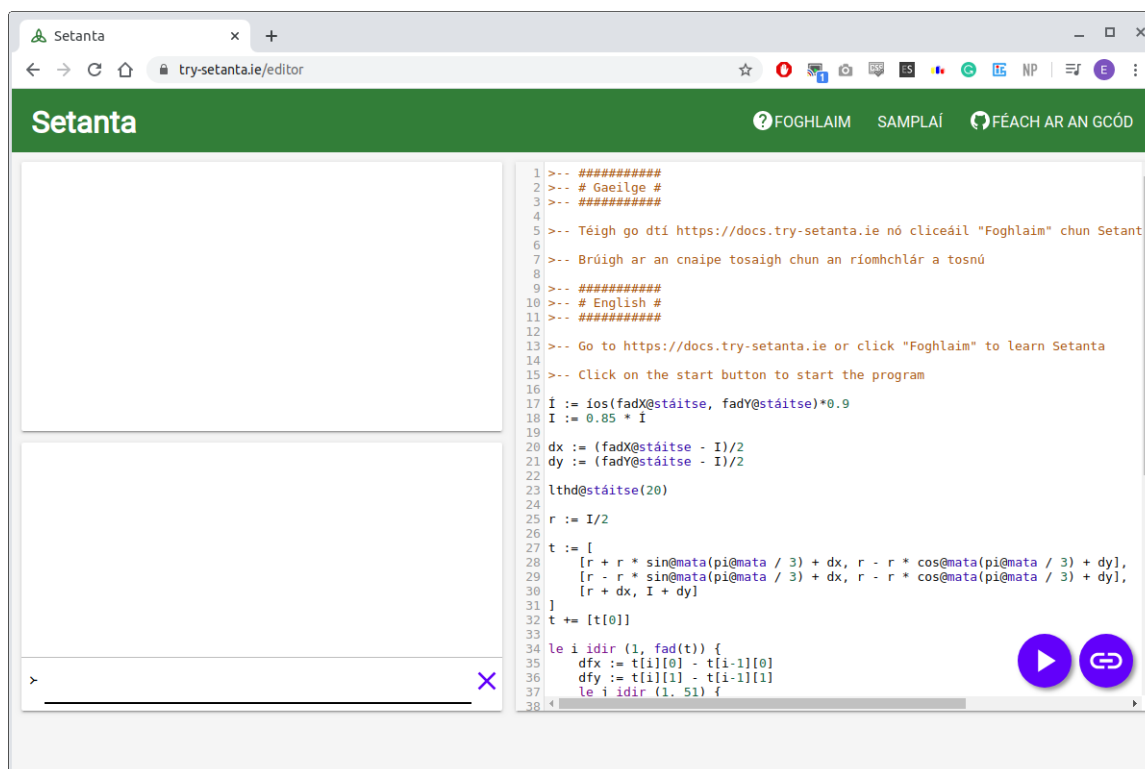
Figure D.2: Default editor page
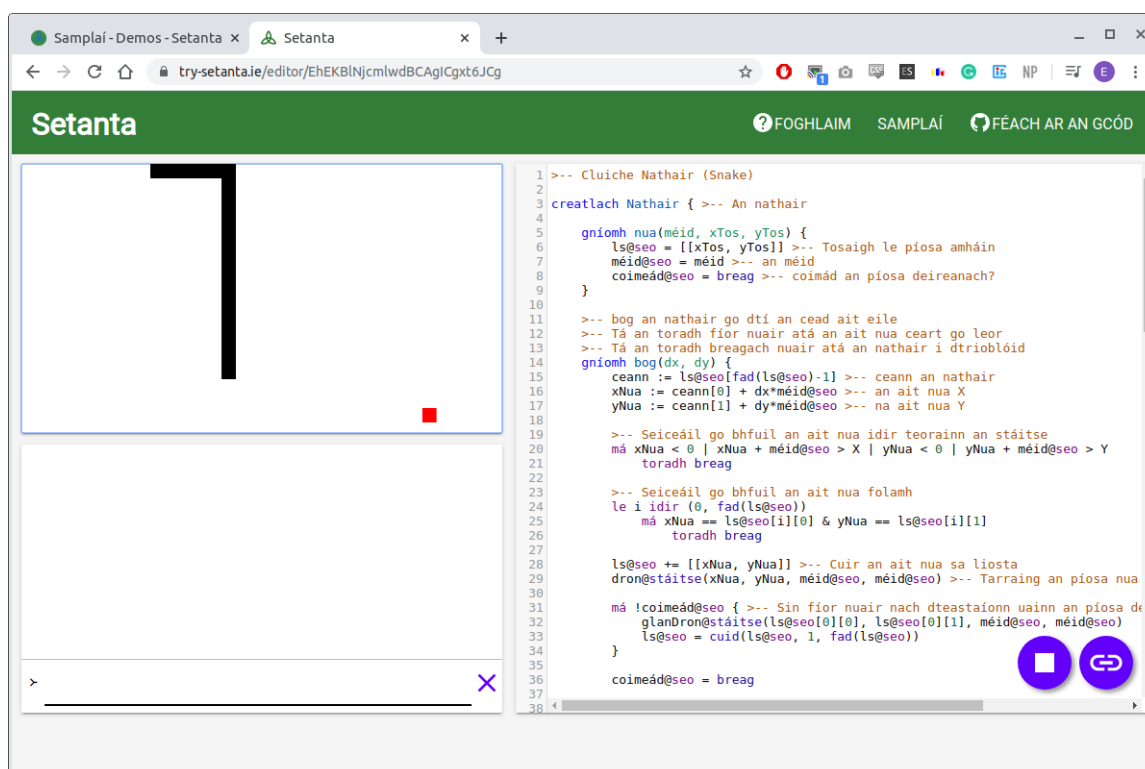
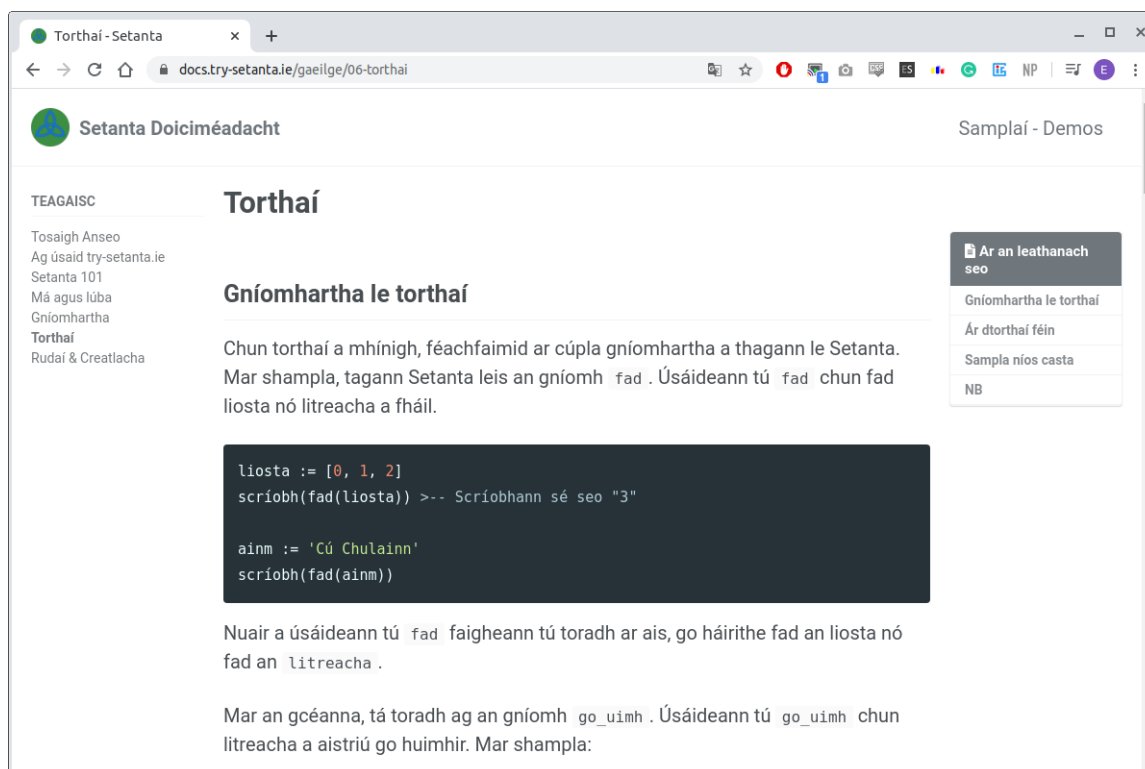Figure D.3: Snake game in progress

Figure D.4: Documentation site (Irish section)
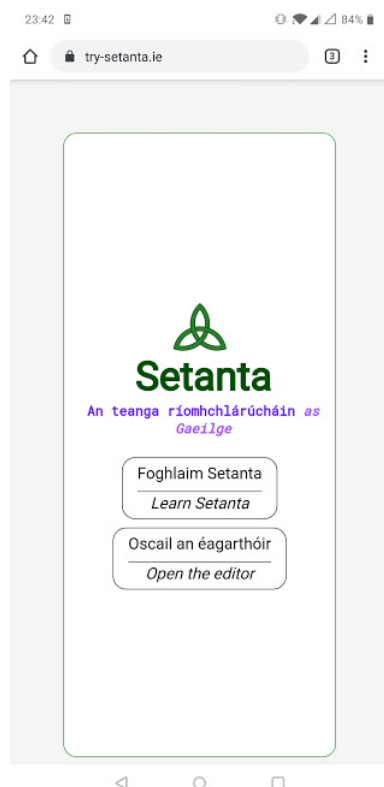
D.2 Mobile Screenshots
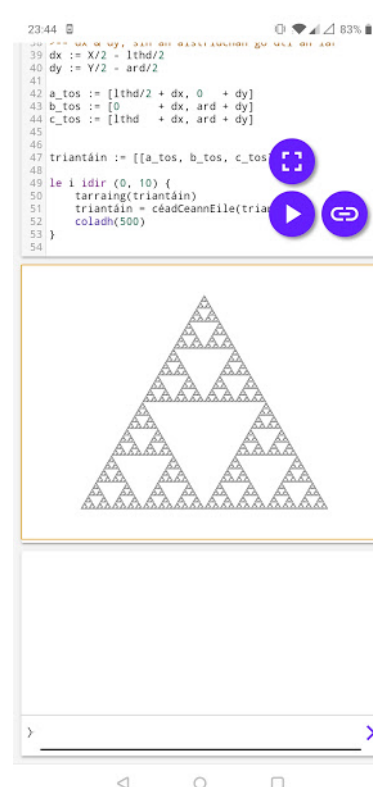

Figure D.5: Landing page on mobile

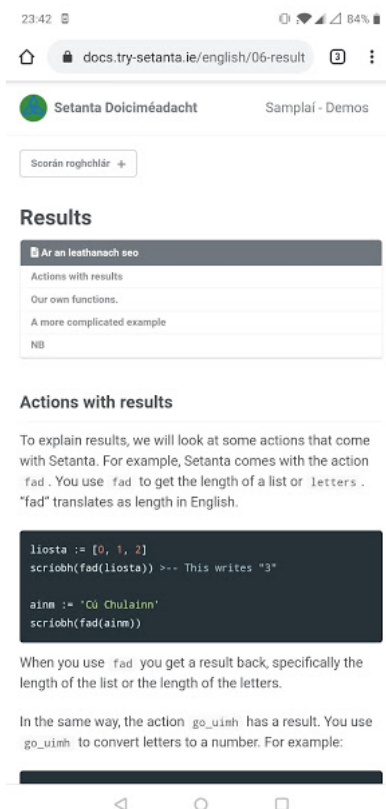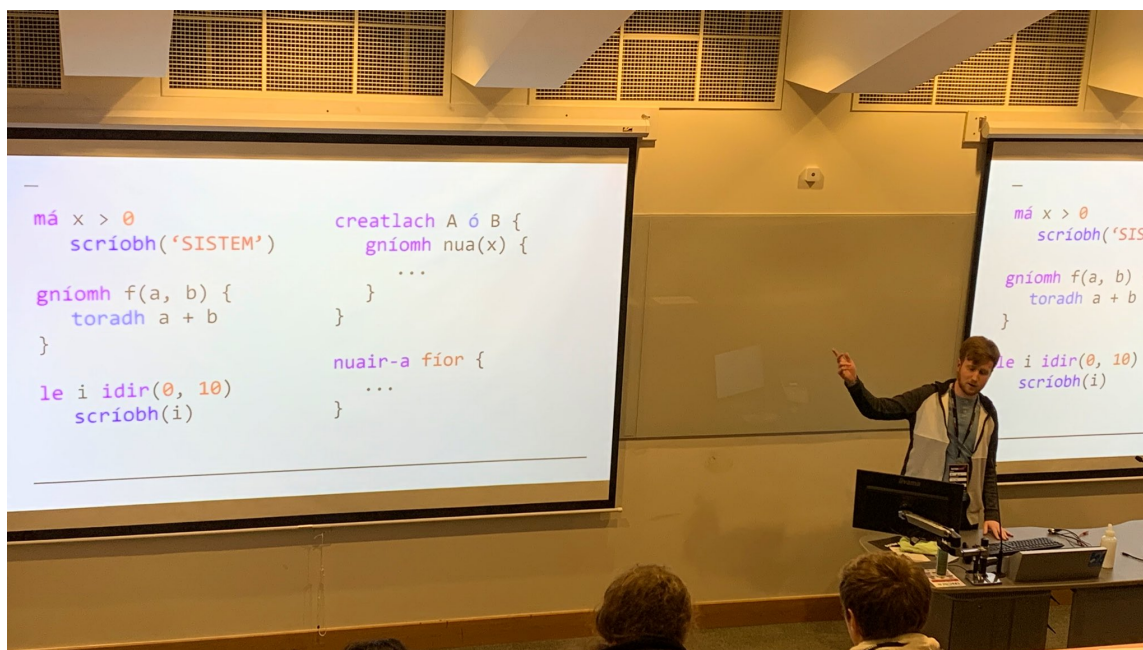Figure D.6: Sierpinski triangle demo running on mobile

Figure D.7: Setanta documentation mobile (English)

D.3   Bonus pictures

Figure D.8: SISTEM 2020 Talk