# *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

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Computational Thinking.

Supervisor: Dr. Barak Pearlmutter.

# Contents

# Chapter 1

# Abstract

The world of programming languages is one dominated by the English language, practically all programming languages that are used today are designed to be used in English. English is established as the lingua franca of the programming world. Studies have shown that language affects the thoughts of the speaker[2]. How does it affect how we design our programming languages? In recent times Irish is often thought of as an academic, historical language. However, Irish is a language used by 73,000 people on a daily basis[1]. 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. 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 it's syntax and semantics. We aim to create a novel, expressive language, and an online environment where the language can be used and learned. In this project we create a modern, powerful Irish programming language *Setanta*. In the process of designing *Setanta* we discover syntactic constructs 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 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.

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. Ireland is a language that historically has 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 they 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 the PL design to be influenced by the Irish language at every stage.

*Setanta* will be designed with education in mind. It will be built to run in the browser, in order 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 it, 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 *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 shapes and interact with visual elements. Research has shown that 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 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. 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 creating 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 it's own. The creation process for *tsPEG* involves reading up on 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 visual look of the website, and improve the concurrency abstractions as we prepare it for release.

## 2.4 Evaluation Metrics

The main strategy to evaluate this project is by evaluating 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.

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 it's 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.

## 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). *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.

- **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. *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 are able to save their code and send it to their friends by the user of unique URLs assigned to each saved piece of code. try-setanta.ie also features the planned graphics display where the programmer can draw and manipulate different shapes and objects in a graphical way.

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

  Part of the goals of designing *Setanta* was to allow the Irish language to influence it's 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.

- **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.

# Chapter 3

# Background

## 3.1 Executing Code on the Browser

The only PL that standard web browsers (*Google Chrome, Mozilla Firefox, Safari, Edge, Internet Explore 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.

### 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, allowing users to pass in **callbacks**, which are functions that will be executed at some later time. Figure 3.1 contains some simple code that would work if the browser thread allowed blocking.

Ideally this 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. Figure 3.2 shows code that uses the `setTimeout` API to add a callback to the task

```
1  function sleepFor100ms() {
2      print('Starting sleep')
3      sleep(100)
4      print('After sleep')
5  }
```

**Figure 3.1:** Code if thread can be blocked.

```
1  function sleepFor100ms() {
2      print('Starting sleep')
3      setTimeout(() => print('After sleep'), 100)
4  }
```

**Figure 3.2:** Using `setTimeout`

queue.

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.

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

### 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 describe 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]

## 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 sets of rules for re-writing strings, first formally researched and defined by Noam Chomsky, 1956[9]. In recent times they find frequent use in specifying the syntactic grammar of various PLs.

The formal grammar family that has found the most usage is that of CFGs, Context Free Grammars. PEG Grammars are a new type of grammar that has come into prevalent usage recently, they are very similar to CFG grammars but are defined in such a way that they cannot be ambiguous. It is conjectured that PEGs are more powerful than CFGs[10]

## 3.4 Parser Generators

A parser generator is a program that, given a description of a grammar in some formal syntax, 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* 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 designed to meet the following requirements:

- *tsPEG* must be a fully functional TypeScript 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 outputted 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.

## 4.2 *Setanta*

The technical requirements of *Setanta* are:

- *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.

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. try-setanta.ie exists as a portal to using and learning *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 be clearly laid out and 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 the TypeScript PL. *tsPEG* runs on 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 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 own 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.

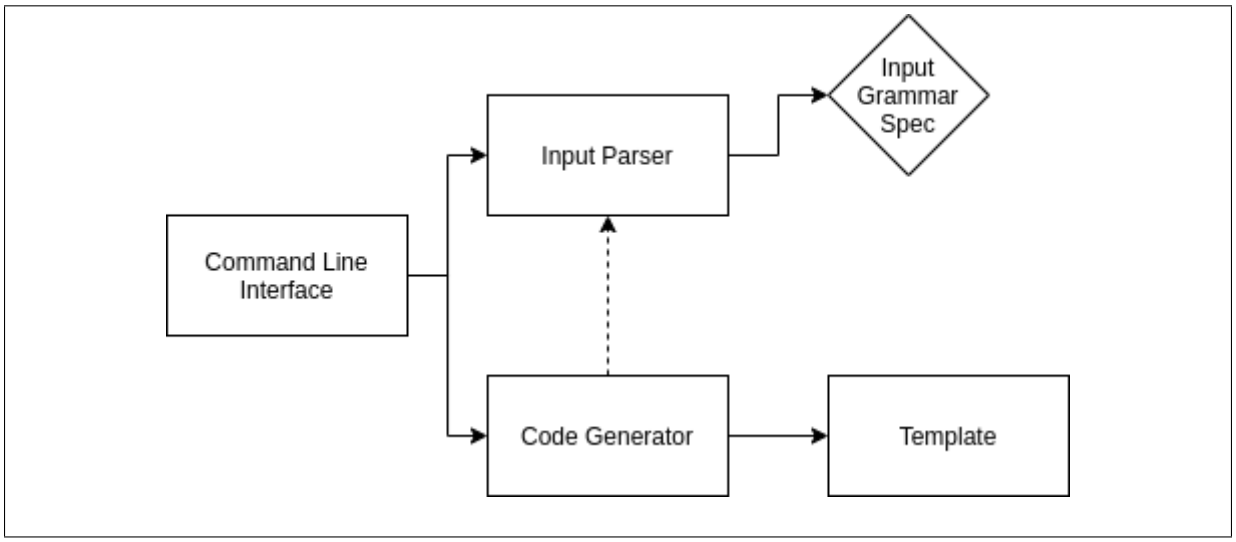


**Figure 5.1:** *tsPEG* High Level Design

```
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

### 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

$$\text{match} := \text{rule1} \mid \text{rule2} \mid \text{'a+'}$$

defines a new rule `match` that tries first `rule1` then `rule2`, then tries to match the regex expression `a+`. The full list of operators is included in Appendix A.

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.

```
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. Then the meta-grammar for the input grammar syntax was written and the generator was run. This replaced the hand written parser with a new generated one, self hosting itself and opening itself up to bootstrapping.

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.

## 5.2 *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.

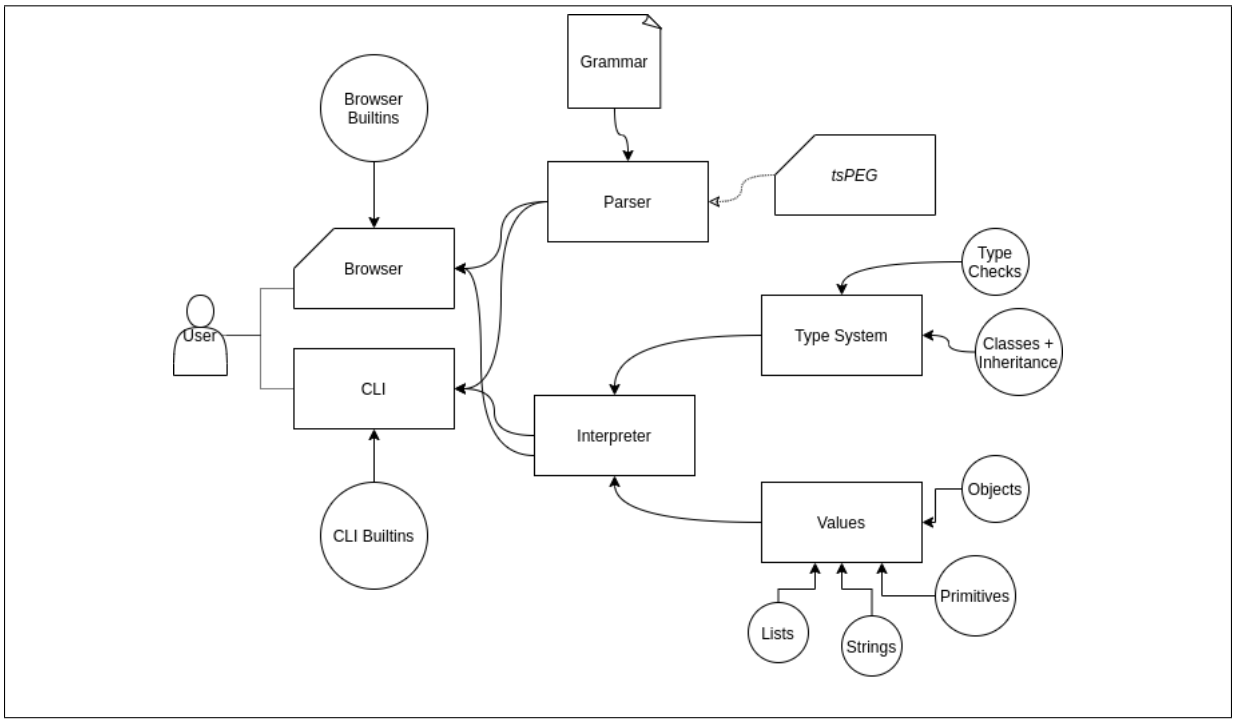


**Figure 5.5:** *Setanta* High Level Design

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 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* support 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 start and ends to 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
```

### 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 '.'.

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 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 expression 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.

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 Allowing blocking operations

As discussed in the technical background section, the JavaScript runtime, whether it be NodeJS or a browser, only allow usage of a single thread, and no blocking operations. However, writing *Setanta* allows the user to 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

```
setTimeout(() => {
    console.log('sleep1');
    setTimeout(() => {
        console.log('sleep2');
        setTimeout(() => {
            console.log('sleep3')
        }, 100);
    }, 100);
}, 100);
```

Listing 5.6: Setanta

```
sleep(100)
scríobh('sleep1')
sleep(100)
scríobh('sleep2')
sleep(100)
scríobh('sleep3')
```

**Figure 5.6:** Equivalent code in JavaScript + Setanta

# Bibliography


[1] CSO, *The Irish language*, 2017, 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 https://en.wikipedia.org/wiki/Non-English-based_programming_languages.

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

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

[6] https://www.typescriptlang.org/

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

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

[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-