

# **LexiTrain: A Formal Language for Creating Endurance Sports Training Plans**

Dylan Barratt

2118928

LexiTrain



**Swansea University**  
**Prifysgol Abertawe**

Department of Computer Science

Adran Gyfrifiadureg

1st May 2024

# Declaration

## Statement 1

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed      Dylan Barratt (2118928)

Date        01/05/2024

## Statement 2

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by citations giving explicit references. A bibliography is appended.

Signed      Dylan Barratt (2118928)

Date        01/05/2024

## Statement 3

The University's ethical procedures have been followed and, where appropriate, ethical approval has been granted.

Signed      Dylan Barratt (2118928)

Date        01/05/2024

# **Abstract**

Training plans play a pivotal role in enhancing athletes' performance in endurance sports. Numerous methodologies for creating training plans have emerged, each offering a unique approach.

This dissertation details the creation of the formal language (LexiTrain) and its compilers. This includes creating the formal language design; the formal language, using a parser generator such as ANTLR4; a web-based calendar compiler, using web technologies such as Svelte and TypeScript; compilers for other file formats; and extensive documentation for the language.

By creating a well-specified and clearly defined formal language, LexiTrain is intended to make writing training plans more straightforward with the ability to combine different methodologies into a single framework. This helps coaches or athletes that write training using multiple methodologies or approaches.

In addition to the formal language, the provided compilers allow LexiTrain to create practical tools for use in the real world such as the web calendar, similar to how other popular training platforms display training.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project Aims . . . . .	1
1.2	Changes to my Approach . . . . .	2
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Endurance Sports . . . . .	2
2.2	My Involvement in Endurance Sports . . . . .	3
2.3	Training for Endurance Sports . . . . .	3
2.4	The Importance of Planning Training . . . . .	3
2.5	Types of Planned Training . . . . .	4
2.6	Writing Sessions . . . . .	5
2.7	Writing Training . . . . .	8
2.8	LexiTrain and a Connected Services Ecosystem . . . . .	9
2.9	L <sup>A</sup> T <sub>E</sub> X Inspiration . . . . .	12
2.10	Formal Languages . . . . .	12
2.11	Backus-Naur Form . . . . .	14
2.12	Making and Using a Formal Language . . . . .	14
2.13	Language Parsers . . . . .	15
2.14	ANTLR4 . . . . .	15
2.15	Web Development Tools . . . . .	17
<b>3</b>	<b>General structure of LexiTrain</b>	<b>18</b>
3.1	Language Design . . . . .	18
3.2	Web Calendar Compiler . . . . .	19
3.3	Portable Document Format Compiler . . . . .	19
3.4	Language Documentation . . . . .	19
<b>4</b>	<b>Details of Implementation</b>	<b>20</b>
4.1	Project Design . . . . .	20
4.2	Language Implementation . . . . .	23
4.3	Compilation . . . . .	26
4.4	Documentation . . . . .	30
<b>5</b>	<b>Testing and Results</b>	<b>31</b>
5.1	Test One - Metadata . . . . .	31
5.2	Test Two - Imports and Session Files . . . . .	32
5.3	Test Three - Meso Three Period Test . . . . .	32
5.4	Test Four - Full Period Test . . . . .	32
5.5	Test Five - Function Testing . . . . .	33
5.6	Test Six - Meso Three Calendar . . . . .	33
5.7	Test Seven - Meso Three PDF . . . . .	33
<b>6</b>	<b>Conclusion</b>	<b>34</b>
6.1	Project Development Lifecycle . . . . .	34

6.2	Critical Analysis . . . . .	35
6.3	Contribution to Computer Science . . . . .	36
<b>A</b>	<b>Appendix</b>	<b>40</b>
A.1	BNF Rules . . . . .	45
A.2	Grammar File Rule Examples . . . . .	50
A.3	TypeScript Examples . . . . .	52
A.4	Test Results . . . . .	56

## List of Figures

1	Dr. Seiler's three zone model [35] . . . . .	7
2	Screenshot showing <a href="https://intervals.icu">https://intervals.icu</a> 's data for a week . . . . .	9
3	Screenshot showing <a href="https://trainingpeaks.com">https://trainingpeaks.com</a> 's workout builder . . . . .	10
4	Planned training mesocycles written in a notepad . . . . .	11
5	Tokenization and Parse Tree for the input: '10 + 10 * 1000 + (10 * 10/(5 + 1))' . . . . .	17
6	Design made in Figma for a web calendar displaying compiled training . . . . .	21
7	Design made in Figma for a documentation page . . . . .	22
8	A session import file requested . . . . .	27
9	Sport icons shown for different days in the calendar . . . . .	29
10	MkDocs generated web page for period file page markdown file (appendix listing 12) (one page split into three images) . . . . .	54
11	Mesocycle three written in Intervals.ICU . . . . .	55
12	Test one parse results . . . . .	56
13	Test one data object in a web browser (where the compilation is run) . . . . .	56
16	Test two import parse tree . . . . .	57
17	Only the first set of tokens (this) and the final set of tokens (figure 18) (up to the EOF) have been provided due to the length of the token list. The final screenshot of tokens shows that no string was not tokenized as it reaches the EOF. . . . .	58
18	Test three period tokens part 5 (last set of tokens up to the EOF) . . . . .	59
19	Test three period full parse tree, split version at bottom of appendix for image clarity . . . . .	60
20	Test three session tokens . . . . .	60
21	Test three session parse tree . . . . .	61
22	Test four parse tree . . . . .	61
24	TS-Jest test results . . . . .	63
25	First month of the compiled training plan calendar (note the displayed version is a shrunk version of the final webpage) . . . . .	64
26	Second month of the compiled training plan calendar (note the displayed version is a shrunk version of the final webpage) . . . . .	64
27	The details of a single day in the calendar . . . . .	65
28	The compiled PDF . . . . .	66

## List of Tables

1	Norwegian Olympic Federation five zone heart rate model [35] . . . . .	6
2	Dr. Coggan's seven power zones [2] . . . . .	7

# 1 Introduction

LexiTrain is a formal language used to describe training plans for endurance athletes, enabling the compilation of detailed and attractive training plans.

To complete an endurance event, an athlete must train for a period of time. Throughout this preparation period, athletes plan their training to make best use of their time. This training may be written by the athlete or their coach. Presently, this process can vary greatly across different platforms and planning methods, resulting in a time-consuming and often bewildering experience as users have to familiarise themselves with each new platform they try. LexiTrain offers athletes and coaches a straightforward solution to this problem.

## 1.1 Project Aims

In my initial document I proposed the concept of LexiTrain, a formal language that would allow users to describe their training in a concise and straightforward way. I have now created this language and an application that can demonstrate the language's use. The original aims for the project were:

1. To create a language that can succinctly and simply define training plans writeable by users at all levels of computer literacy.
2. To allow the language to be compiled into formats that are compatible with other training platforms and can be statically viewed by users (such as a .pdf file).
3. To create a web-based training calendar that LexiTrain can be viewed on.
4. To provide a well-structured set of documentation for the new language.

### 1.1.1 Aim One

The primary aim was the creation of the language itself. This stage included the conceptualization, design, and implementation of the language. This involved thorough research into existing analogous languages and the formulation of practical use case scenarios, with inspiration from training periods I had written for actual training.

To complete this stage, I wanted to create a full lexer and parser for the language based on the aforementioned designs and use case examples.

### 1.1.2 Aim Two

The second aim was to provide compilers for the language parser and lexer. This would allow the language to be used in applications by users. As the project developed, the approach to this aim changed significantly with the introduction of a parser generating tool called ANTL4 (Another Tool For Language Recognition) [32]. The use of this tool greatly sped up and improved the development for LexiTrain.

This aim specified that multiple compilers for different formats should be created to support the LexiTrain language. Using scripts written alongside ANTL4, a data object can be generated

to create the compiled content. A similar method of generating the data object can be reused within different compilers, speeding up the development time of each compiler. This data object was also used to compile the web-based training calendar provided with the project.

### **1.1.3 Aim Three**

The third aim was the creation of a web-based training calendar for the end user that displays a compiled version of LexiTrain. This provides a platform to demonstrate use of the language. This aim included the design, coding, and testing of the web application. The completion of this objective was a pivotal part of the project as it allowed proper testing of the language in a real world environment. Completing the web application helped to identify some flaws in the design of the original language and made further refinements of the language's rules possible.

### **1.1.4 Aim Four**

The fourth aim was to provide a set of documentation to accompany the language. This documentation is intended to aid users in using the LexiTrain language and was written in a way that could be understood even if the user lacks a background in computer science. This includes the use of plain language, examples, and additional definitions where complex terms are used. The examples written in the design stage of the language were used in the documentation to provide users with realistic use cases for the language with demonstrations of common features.

## **1.2 Changes to my Approach**

After adopting ANTLR4, the method for completing aims one and two changed slightly. Using ANTLR4, a grammar is created that can then be run through the ANTLR4 tools to generate parsers and parse tree listeners in many different coding languages. A grammar is the combination of both a lexer and parser. The generated parse tree listeners make up a part of the compilation process and compilation depends on these listeners. The process is similar to the original aim of writing an individual lexer and parser, therefore not increasing the workload or changing the project's outcome. Both aims were still able to be completed.

## **2 Background**

### **2.1 Endurance Sports**

Endurance sports are a popular hobby and even a profession for some. They are athletic activities that primarily rely on an athlete's ability to sustain prolonged physical effort. These activities include training and racing in a chosen sport. One popular endurance sport is triathlon, which consists of a swim event, a bicycle event, and a run event. A common triathlon distance is the Olympic distance: 'The Olympic distance triathlon involves successive swimming, cycling and running sessions; it begins with a swimming segment of 1500 m, followed by a 40-km cycling leg and concludes with a 10-km running leg.' [19]



## **2.2 My Involvement in Endurance Sports**

In my personal life I am a keen triathlete and the race captain for the university's triathlon club. This year I am coaching myself to complete a full distance triathlon. This requires lots of written planned training. Through this process I have found the current way of writing training laborious and not at all straightforward. My own knowledge and experience of writing training plans has greatly influenced my work in creating LexiTrain and has also helped to inform my decisions about how language files are written.

## **2.3 Training for Endurance Sports**

When completing an endurance event, an athlete goes through a preparation period. This includes completing training activities that can simulate race conditions or encourage performance adaptations to improve race performance. This period can be planned and structured to provide the athlete with the most efficient way to use their (often limited) training time.

Training also comes with risks as the athlete must often complete strenuous activities. 'Health professionals who care for elite athletes are concerned that poorly managed training loads combined with the increasingly saturated competition calendar may damage the health of athletes.'[37]. It is therefore essential that an athlete completes the correct type of training as well as the correct amount of training, both of which come from realistic, well written training plans.

## **2.4 The Importance of Planning Training**

To ensure that an athlete is able to achieve peak performance, a period where an athlete reaches an elevated level of fitness, coaches and athletes must 'design a well-controlled training program'[15]. A peak performance is intended to allow an athlete to complete a race to a high standard, often aiming to beat a previous personal best or the other competitors.

A way of quantifying the amount of training an athlete completes is with training load. Training load is defined as "internal" or "external" workloads based on what data is available. Two examples of this data are: rate of perceived exertion (RPE) or heart rate (HR). RPE is self-assessed by an athlete based on a pre-defined scale, typically from 1 to 10. HR is measured using a heart rate monitor (HRM) to count the heart rate in real time, typically quantified in beats per minute (bpm). This training load can be predicted when writing training for an athlete indicating how the training stimuli will affect the athlete's body [14].

By using both time and load management an athlete or coach can create a more informed training plan. Athletes are constrained by the time they have available during the day to train. Efficient time management is crucial when writing a training plan.

Athletes are also constrained by the amount of training load they can cope with. 'The Constrained Model of Total Energy Expenditure (CMEE) holds that energy expended in increased physical activity will not add to total energy expenditure, but instead to induce compensatory energetic savings elsewhere'[12]. Therefore, an athlete must be careful when planning training load to ensure that the energy expenditure from the planned endurance training does not become excessive and lead to harm.

The importance of planning was evident at the 1964 Tokyo Olympics ('All roads lead to the 1964 Tokyo Olympics' [21]). In the 1964 5000m race, the leading athletes finished within seconds of each other [22]. The leading athletes also all had 'extremely different training' [21]. For these athletes, the actual training that each undertook was less important. The similarity between these athletes was meticulous planning and execution of the training.

Training periodization is a branch of training theory that outlines the plan an athlete has for an annual cycle, a period where an athlete prepares for and then completes races. This plan aims to maximize training adaptations and prepare the athlete for peak race performance.

The first proper proposed periodized training was in the former USSR where training was divided into 'separate periods of general and more specialized training'. This method of planning training is still used today to plan an athlete's annual cycle with a few adapted versions such as block periodization, periodization charts, and linear and non-linear periodization [23].

## 2.5 Types of Planned Training

A recent review identified what training methodology was used by highly trained and elite distance runners to plan their training. These training methodologies fit into the aforementioned periodization models and specify the quantity and type of session an athlete has to complete. The three models that the review looked for were: pyramidal, polarized, and threshold. These three models are characterized by the intensity distribution across sessions in a period.

**Pyramidal** - 'The pyramidal model is characterized by a decreasing training volume from z1 to z2, and z3, respectively'

**Polarized** - 'The polarized model is characterized by covering approximately 80% of the volume at z1 with most of the remaining 20% conducted at z3'

**Threshold** - 'The threshold model features a higher proportion of overall volume conducted in z2 (ie, >35%) [with] the majority of the training volume (ie, 60%–62%) in z1' [7]

Although these 'delimitations have not yet reached a full consensus in the current literature' they do provide a good idea of what is required to write the training for these athletes [7]. Coaches and athletes must be able to plan training such that the intensity distribution conforms to their chosen model and adheres to the general training strategy applied by the author.

Achieving diverse training methods with similar outcomes requires a flexible and comprehensive way to express training. This presents a challenge that most modern training platforms must solve by providing users with the most prevalent methods of training expression.

When planning a period using periodization, there is a standard set of terminology that training authors use. This terminology allows authors to express training in a standard way, allowing the athlete to understand the training if they have used the terms before. A cycle is a period of time in a training plan. Within periodization terminology, these cycles are defined as:

**macrocycle** - 'a single competitive season' this can be anywhere from a year to over 4 year cycles (including both competition preparation and competition). A macrocycle is made up of a number of mesocycles.[30]

**mesocycle** - ‘Medium duration training cycles that typically contains more than two to six interrelated microcycles’. [30]

**microcycle** - ‘a number of training sessions appropriately interrelated in order to reach one or more specific objectives’ [30]

By these definitions, a training plan firstly begins with a macrocycle. This is a period of time including both preparation and race where an athlete has a specific overarching goal to achieve. This could be to win a race in the race period or to become stronger by the end of the macrocycle. A macrocycle is then made up of many mesocycles.

These mesocycles contain a training objective that, on completion, brings the athlete closer to a main goal in the macrocycle. For example, one mesocycle may aim to increase the duration of session an athlete can handle. A mesocycle comprises of multiple microcycles and normally lasts ‘between 4 and 6 weeks’ [30].

A microcycle allows an athlete to plan all the sessions in a week (‘the most common length being 7 days’ [30]) that need to be completed to achieve the current mesocycle’s aim. Microcycles allow the athlete to plan all the actual sessions in a week that need to be completed as a part of the goal set out in the macrocycle.

## 2.6 Writing Sessions

Writing a session is arguably the most important part of a training plan. A session is the work an athlete will complete. It details what the athlete is doing, how they should do it, and sometimes the effect this should have on the athlete (expressed in terms such as the aforementioned training load).

One example of a session is an interval session. An interval session is ‘a method of training which alternates between exercise periods with high and low intensity’ [39] where the aim of the session is to allow ‘the athlete to maintain a higher intensity and work for longer time at high intensities’ [39].

A typical interval session on a bike could be:

```
1 1 hour easy warm-up (typically low intensity work)
2
3 5 repetitions of:
4   3 minutes at high intensity
5   5 minutes at low intensity
6
7 30 minutes easy cool-down (again, typically low intensity work)
```

Listing 1: An example bike interval session.

To write this session, the author must be able to write each section individually. Each section should be able to include the required workloads. For example, the first section needs to be written with a workload of one hour at a low intensity. The way a coach expresses intensity is complex and many have different measures for it.

### 2.6.1 Intensity and Zone Models

One way of expressing intensity is by using a defined zone model. A zone model categorizes different levels of intensity using physiological markers (for example, heart rate). There are lots of different zone models used by athletes and coaches. In order to balance simplicity and comprehensiveness, three different intensity models have been chosen for LexiTrain. Each model measures a different physiological marker allowing authors a wide range of options for intensity selection. The three models supported in LexiTrain are: five zone heart rate, three zone ventilatory (or Lactate Threshold (LT)), and seven zone power. These three zone models encompass enough information to give training plan authors versatility with a variety of intensities across different endurance sports.

### 2.6.2 Five Zone Model

The five zone heart rate model (table 1) was developed by the Norwegian Olympic Federation. It has been used by them as a guideline for cross-country skiers, rowers, and biathletes. It is now used more widely by the endurance community as heart rate monitors are accessible. The five zones are based on a percentage of the athlete's maximum heart rate. Through testing, an athlete can find their maximum heart rate and then percentages of it to calculate what zone a heart rate is in [35].

Intensity Zone	Heart rate (% max)
1	50-65
2	66-80
3	81-87
4	88-93
5	94-100

Table 1: Norwegian Olympic Federation five zone heart rate model [35]

### 2.6.3 Three Zone Model

The three zone intensity model by Dr. Stephen Seiler (figure 1) is based on ventilatory thresholds. It splits training into low (zone one), moderate (zone two), and high (zone three). The three zones can be separated and measured by blood lactate turn-points. The first cut-point (between zone one and two) is where lactate levels become elevated above resting levels. The second cut-point is where lactate levels accumulate rapidly and can no longer be maintained for an extended period of time [35].

### 2.6.4 Seven Zone Model

The seven zone power model by Dr. Andrew Coggan (table 2) is based on power meter data. A power meter is a device most commonly found on a bicycle that measures the power output of the rider. The amount of power an athlete produces can be measured and equated to a workload. Power meters can also be found in other endurance sports. Power is also roughly equivalent to

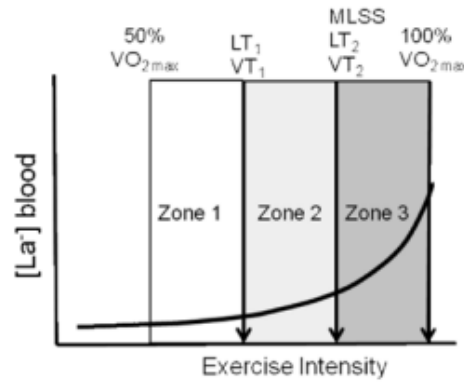


Figure 1: Dr. Seiler's three zone model [35]

a heart rate measurement, allowing athletes to use two data points to ensure zone compliancy within a workout. The zones are based on a lactate threshold (LT). Through testing, an athlete can find their LT and use it to calculate the seven zones. Zone seven does not have a percentage of power to measure because it is the athlete's maximal effort [2].

Intensity Zone	Power (% of LT Power)	Heart rate (% of LT HR)
1	≤ 55	≤ 68
2	56-75	69-83
3	76-90	84-94
4	91-105	95-105
5	106-120	>106
6	≥ 121	N/A
7	N/A	N/A

Table 2: Dr. Coggan's seven power zones [2]

Using the three chosen zone models, the above bike session (listing 1) can be expressed in a more specific way:

```

1 1 hour < HRZ2
2
3 5 repetitions of:
4   3 minutes HRZ5
5   5 minutes < HRZ2
6
7 30 minutes < HRZ2

```

Listing 2: An example bike session using an intensity model.

In the example, each section has had the broad intensity converted to an equivalent intensity from one of the aforementioned zone models. In this modified example (listing 2), the previously highlighted first section has had low intensity converted to less than (<) heart rate zone two (HRZ2). When given to an athlete, this session could be instantiated to have an actual heart rate

value so that when the athlete completes it, they can measure their heart rate and ensure that it stays within the specified bound (in this case, less than heart rate zone two).

For LexiTrain to be a comprehensive language, it must be able to express periods of time as well as the intensity and workload required to complete sessions. LexiTrain users should have the ability to write the macrocycle as full file, the mesocycle as a period of time like a month, and a microcycle as a sub-period within a mesocycle such as a 7 day week. Furthermore, within these microcycles, LexiTrain needs to enable the detailed description of individual sessions, similar to the example provided above including workloads and intensities.

## 2.7 Writing Training

Currently, a coach or athlete wanting to write a training plan has many different options. It is important when writing training for an athlete to ensure that the method is accessible to the athlete. For example, if a coach writes the session in a notepad the athlete needs a copy in order to know what training to complete.

The coach can attend training sessions to prescribe the workout and ensure it is completed properly. In this format, getting the training plan to the athlete is straightforward.

Another approach is to use an online coach. An online coach uses a training platform to digitally send athletes their sessions. Once a session is completed, the athlete uploads it in some way to the platform so that the coach can see the progress and compliancy to the original session plan.

An athlete can also use a generic, static training plan. A static training plan is broad and usually includes a goal intended to be achievable if most sessions are completed by the end of the plan's duration. An example of this is a 'couch to 5km' running plan that takes athletes from any level to a 5km run [36].

With all three of the above options, an online training platform can be used. An online training platform is a web application that combines the details of a training plan and the actual sessions an athlete has completed. This data is typically presented in a calendar, with future days showing training to be completed and past days showing completed training. All three approaches listed above can be used in a training platform by uploading the athlete's planned training.

There are many online training platforms available to athletes. In the initial document, three were researched and documented: Intervals.icu, TrainingPeaks, and FinalSurge.

### 2.7.1 Intervals.icu (<https://intervals.icu>)

Intervals.icu is a free training platform that seamlessly syncs planned training with completed training from services that an athlete may use to record their sessions such as Garmin and Strava. Intervals.icu also provides the user with plenty of calculated metrics for both completed and planned training. These metrics are things like the previously explained training load and intensity distribution (shown in figure 2). Intervals.icu allows users to drag and drop training onto a calendar in order to plan a period. It also has a session editor that allows users to drag and drop, or write, workloads with specified intensities to express a session.

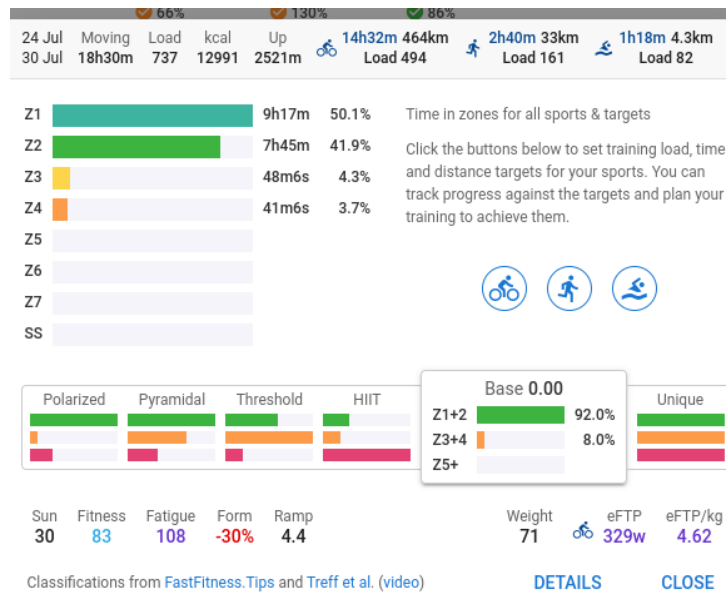


Figure 2: Screenshot showing <https://intervals.icu>'s data for a week

### 2.7.2 TrainingPeaks (<https://trainingpeaks.com>)

TrainingPeaks is a premium training platform that gives users a high level of analysis for both previous and future training. TrainingPeaks has a large platform for online coaches to sell users their training plans. The service ranges from one-to-one coaches, where an athlete has lots of contact and feedback from the coach, to static training plans where an athlete buys a pre-written plan and completes it with no correspondence. Once a plan is bought, it is seamlessly imported into the user's TrainingPeaks calendar. TrainingPeaks has a GUI workout editor that allows users to drag and drop session blocks into a complete session. It also functions by letting users drag and drop sessions into a calendar to plan a period.

### 2.7.3 FinalSurge (<https://www.finalsurge.com/>)

FinalSurge is a web-based training platform that is free for athletes. The data it offers is more simplistic than the other two platforms. However, it still offers similar planning functionality. FinalSurge allows a user to drag and drop sessions into a calendar to plan training. It also has a workout builder that lets the user plan workloads with specified intensities.

Each of these platforms offer the same core functionality. They all enable users to plan individual sessions and training periods. However, they all offer it in different ways. None of the platforms offer a common language for the expression of training planning.

Each platform adopts its own unique system for planning training, which means users switching between platforms have to understand many different methods to express the same training.

## 2.8 LexiTrain and a Connected Services Ecosystem

When an athlete wants to complete a session in a training plan, they first must check what the session is. Using one of the above training platforms, the athlete can refer to the current day's



workout to find out details such as the required intensities and durations.

The athlete then completes the session and can record it on a device like a multi-sport watch or cycling computer. These devices are offered by many different manufacturers.

Each company has a different way of uploading the training to a training platform, with some even offering their own training analysis platform. For example, Wahoo Fitness is a fitness technology company that offers lots of different products to record training. One such product is the Wahoo Element Roam bike computer. The Element Roam allows users to track their workout progress using data such as time and gps. It also supports displaying the user's planned training session during a workout. This can be imported from an online training platform like TrainingPeaks or uploaded manually in the .fit file format [20].

Once a session is completed it is then uploaded to a training platform for a coach or athlete to analyse and check metrics such as workout compliancy. Using the Wahoo Element Roam, the athlete must first upload the session to the Wahoo system which then uploads the session to connected training and social media platforms such as Strava.

This whole process is unnecessarily complicated. It involves lots of different platforms for the user to have to interact with. It also means lots of different data types and platform connections for the developer to integrate.

This complexity poses significant challenges for independent developers to make products for endurance sports systems as a whole. It requires working with different companies to make a product that enables users to author training plans. The process would be simplified greatly if there was a unified method for planning training, and this is the intention behind LexiTrain. This does not mean that training platforms would become obsolete, but rather they will share a common planning process.

By giving users a standard way to express training, developers would only need to implement LexiTrain for this part of their training platform. Training plan authors would also only have to learn one way of writing training regardless of what platform they choose to use. This process of learning is also greatly simplified and improved by the extensive and easily understandable LexiTrain documentation.

The platforms mentioned above all have some way of writing training plans equivalent to LexiTrain. The most common way of writing training is by using the drag and drop workout builder that all three platforms provide. Figure 3 shows TrainingPeaks' implementation of the workout builder.

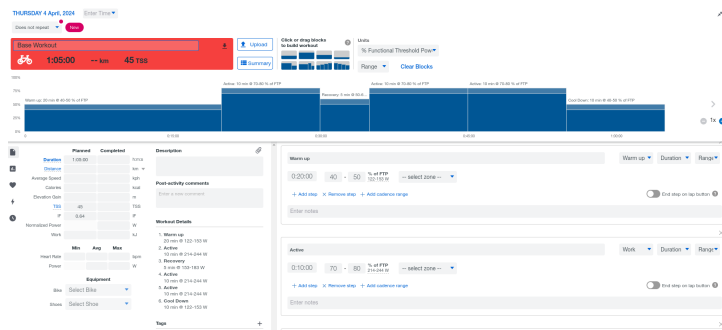


Figure 3: Screenshot showing <https://trainingpeaks.com>'s workout builder



It allows the user to drag and drop workload blocks into a single workout timeline. It uses an intensity measurement specified by the user (in this case, percentage of functional threshold power). However, this method of creating training sessions can only be used in TrainingPeaks. Intervals.icu provides a similar workout builder to TrainingPeaks, alongside a rudimentary language implementation. The language is a very basic way of describing training which is then translated into the typical drag and drop workout builder. As stated in the initial document, the language lacks any proper documentation with the only way for users to learn being from information found on user contributed help forums. Again, this way of creating training is limited to Intervals.icu. LexiTrain aims to address these shortcomings and provide a way to write training that can be used on different platforms. LexiTrain also provides extensive documentation so that users do not have to rely on help forums to learn the language.

LexiTrain is intended to build upon the simplest method of training planning using a notepad and bring it to the web to make it accessible to more athletes. Planning training in written form can be a quick way to express the intended training. As an example, the training I have written for my own current macrocycle has all the mesocycle's overarching aims planned (this is shown in figure 4).

This way of writing training is fast and efficient as it allows the author to quickly outline what they want to achieve. It often uses shorthand for writing longer ideas. For example, in the notepad training, I shortened the type of ride I was going to do to tt (Time trial, meaning on my time trial bike) and r (Road, meaning on my road bike).

Providing training authors with the option to add quick notes and write common training terminology in a more concise way reduces the amount of time it takes to create the training. The ease of this approach to writing training should be retained when using a language like LexiTrain, enabling the writing of detailed training to be as succinct as possible.

Meso 1	Meso 2	Meso 3	Meso 4
Swimming	1 2 km	1 3 km	1 3.5 km
2.5 km set	2 2.25 km	2 3.25 km	2 3.75 km
3 km set	3 2.25 km	3 3.25 km	3 3.75 km
3.5 km set	4 2.5 km	4 3.5 km	4 4 km
4 km set			
6 km			
90 km			
120 km			
150 km			
180 km			

Figure 4: Planned training mesocycles written in a notepad

## 2.9 L<sup>A</sup>T<sub>E</sub>X Inspiration

LaTeX is a software system designed for typesetting documents that builds upon the TeX typesetting system. LaTeX is extensively employed within academia to produce scientific publications and documents. In LaTeX, markup is utilized to define both the content and layout of the document, in contrast to the direct manipulation of formatted text commonly found in WYSIWYG word processors such as Microsoft Word [26].

LaTeX produces high quality, professional looking documents with precise formatting. Like LaTeX, LexiTrain aims to include its formatting and content in the same file. This means that once the training has been written it can be stylised and formatted by the training platform or compiler it is used in. For example, the web-based calendar developed for LexiTrain styles and displays the written language files as a calendar.

LexiTrain also offers the possibility for both sessions and periods to be written in a single file. This further eliminates the need to think about the format of the compiled training when laying out the document.

LexiTrain provides users with a language to describe training. In the most basic sense, this language is a defined method of describing training. It can be written in any text editor as plain text. This approach is similar to writing training in notepads, as mentioned above.

This text file can then be compiled and turned into a formatted training document in the user's specified format.

## 2.10 Formal Languages

A formal language has a structure (including the syntax and semantics) that is 'precisely and algorithmically defined'. In other words, there are clear rules that dictate how sentences are constructed and what they mean within that language. This ensures consistency and clarity in communication and allows for rigorous analysis and manipulation.

Formal language theory validates that a string's format is valid (not that it has meaning) [17].

Applications of formal language theory are widely used. One example of this is in programming languages. Programming languages are written with strict syntax and semantics.

C++ has a formalisation defined by the International Organization for Standardization (ISO) C++ programming language standard. This formalisation gives C++ a clear and precise definition of syntax, semantics, and behaviour [13], guiding developers when writing code by reducing syntactical ambiguity.

Another example of a formal language implementation is eXtensible Markup Language (XML). XML is a markup language that defines rules for encoding documents in a format that is both human and computer readable. The syntax of XML is formally defined by the XML specification [16].

### 2.10.1 Grammars and Formal Language Theory

'A grammar can be regarded as a device that enumerates the sentences of a language.' [9].

To define the language for LexiTrain, a specific way to precisely define the syntax and structure of the language was needed. A grammar was created for this purpose.

Mathematically, a grammar is made up of an alphabet (a disjoint set of symbols) with a specific start symbol.

An alphabet is a set  $\Sigma$  of symbols or characters allowed in the language. The set  $\Sigma$  must contain terminal symbols, these are final symbols in a created word.

A start symbol  $S$  can be a terminal or non-terminal symbol, where a non-terminal symbol is a symbol that can be replaced (a syntactic variable). The start symbol represents the starting point of generating strings in a language.

A grammar then consists of a finite list of pairs  $(u, w)$  with  $u, w \in (\Sigma \cup N)^*$ .

Each pair is a production rule that specifies how the items in the alphabet can be used to generate valid strings of symbols.

A production rule can be written as  $u \rightarrow w$ . This is the syntax of the language.

A grammar  $G$  defines a language  $L(G) \subseteq \Sigma^*$   $t \in L(G)$  is true if, and only if we can make  $t$  with the grammar's production rules. If this statement is true then  $t$  is a word in the language  $L(G)$  [34].

An example of a grammar:

**Alphabet:**  $\Sigma = \{\text{the, man, woman, jumps, falls}\}$   
**Non-terminal symbols:**  $N = \{S, \text{NOUN, NP, VERB}\}$   
**Production rules:**  
 $S \rightarrow \text{NP VERB}$   
 $\text{NP} \rightarrow \text{the NOUN}$   
 $\text{NOUN} \rightarrow \text{man}$   
 $\text{NOUN} \rightarrow \text{woman}$   
 $\text{VERB} \rightarrow \text{jumps}$   
 $\text{VERB} \rightarrow \text{falls}$

A valid word in the language would be 'the man falls'.

## 2.10.2 Types of Grammars

Chomsky defines different types of grammars based on their expressive power and the types of languages they can generate. There are four different classes of formal grammars: regular, context-free, context-sensitive, and recursively enumerable [8].

A regular grammar is the least powerful type in Chomsky's hierarchy. Their production rules consist of  $A \rightarrow aB$ ,  $A \rightarrow Ba$  and  $A \rightarrow a$  where  $a$  is a terminal symbol and  $A, B$  are non-terminal. A regular grammar can generate a regular language. Regular languages can be used in various areas such as lexical analysis in compilers [11].

A context-free grammar has the production rules of the form  $A \rightarrow \alpha$  where  $A$  is a non-terminal symbol and  $\alpha$  is a string of terminals and/or non-terminals [10]. With a context-free grammar, a context-free language can be generated which is more expressive than a regular language and includes many programming languages [6].

A context sensitive grammar allows production rules where the left-hand side can be replaced by the right-hand side in the context of a larger string. This can be written as  $\alpha A \beta \rightarrow \alpha \gamma \beta$  where  $\alpha$ ,  $\beta$ , and  $\gamma$  are strings of terminals and/or non-terminals, with  $\gamma$  having the length of  $\alpha$  and  $\beta$  combined.

An unrestricted grammar has production rules without any restrictions. They can generate recursively enumerable languages, encompassing all formal languages that can be recognized by Turing machines [29].

## 2.11 Backus-Naur Form

Backus-Naur Form (BNF) is a meta-language that syntactically describes another formal language. It is used when an exact and precise description of a language is needed. It helps in defining the structure and rules of a language. Using metalinguistic variables (or metavariables), BNF can denote strings of symbols permitted in the given language [27]. An example implementation of a metalinguistic variable can be seen in appendix listing 8.

In the example (appendix listing 8), *<digit>*, *<unsigned integer>*, and *<integer>* are metavariables. They are defined using other metavariables or terminal symbols. The *<unsigned integer>* can also be applied recursively, allowing for the construction of an unsigned integer of unlimited length [27].

BNF is useful for LexiTrain as it presents rules in a manner accessible to users without prior knowledge of LexiTrain's syntax. BNF could be used in documentation to explain specific rules syntax comprehensively.

## 2.12 Making and Using a Formal Language

'A compiler is a tool that translates a program from one language to another language.' [3] Once the formal language has been designed and created, it must be compiled into a format that is easily digestible. First the language must be analysed. This analysis can be broken into multiple stages.

The first stage is the lexical analysis of the language using a tokenizer (or lexer). This stage breaks the raw input string into tokens, where tokens are the smallest unit of the language (symbols) [3].

Once the input string has been split into tokens, the tokens can be used to make language rules. The rules specify the syntax and structure of the language.

Using these defined rules, the input can be parsed. The parser analyses the structure of the input string and tokens according to the language's rules. This process typically involves building a parse tree representing the hierarchical structure of the parsed input.

A parse tree generated for an input string of  $n$  number of tokens has  $n$  nodes belonging to terminal symbols as well as additional nodes belonging to non-terminals.

Parsing a sentence can be completed with two simple methods: top-down parsing and bottom-up parsing:

- Top-down parsing begins at the start symbol  $S$  and recursively expands non-terminal sym-

bols by predicting and matching valid rules with the input string until all leaves of the tree are terminal symbols.

- Bottom-up parsing starts with the input tokens and works towards the start symbol  $S$ , resulting in a complete parse tree. Like top-down parsing, it matches tokens with fitting rules, until it reaches the start symbol  $S$  [33].

Using the generated parse tree from one of the above methods, the analysed input string can be checked for syntax errors (there is no logic checking at this stage). If any errors are found in the parsing process, the user can be given an error and the input string can be rejected.

A correct input string's parse tree can then be read through (at this stage checking for logic errors) to extract needed data and information to be used in the language's compiled format.

## 2.13 Language Parsers

To implement a formal language parser, a few methods can be employed.

The most rudimentary method for writing a parser is with regular expressions. A regular expression (regex) is a sequence of characters that forms a search pattern, used mainly for pattern matching with strings. A formal language's rule set is essentially a list of search patterns, each rule being a different search pattern. The main limitation of using a regular expression is its lack of recursion, as a regular expression cannot be found inside another one. Therefore, regular expressions could be used to parse regular languages [28] but would pose a problem for non-regular grammar rules such as  $\text{SENTENCE} \rightarrow \text{SENTENCE NP VERB}$  where  $\text{SENTENCE}$  can be applied recursively.

Another option is to write a parser by hand. This gives programmers increased flexibility and better error handling. However, writing a parser by hand is a complex and extensive task. Hand-written parsers are often prone to bugs and errors, and are dependent on the language used to write them.

To simplify this process, a parser generator can be used. A parser generator is a tool that automates the process of creating a parser for a formal grammar. It takes a formal language's grammar as input and produces code that implements a parser capable of token recognition and parse tree generation.

One such parser generator is ANTLR (ANother Tool for Language Recognition). ANTLR generates 'human-readable recursive-descent parsers' [31]. Using a parser generator like ANTLR saves the developer the time of implementing a parsing algorithm from scratch and can allow them to focus on defining the grammar.

## 2.14 ANTLR4

ANTLR4 is the fourth iteration of the ANTLR parser generator. 'ANTLR v4 is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files' [32]. ANTLR4 is maintained and continuously developed by Professor Terence Parr of the University of San Francisco. ANTLR4 takes an input grammar, a file split into a lexer and a parser. The lexer defines the tokens (terminal symbols on some non-terminal symbols) that ANTLR should match from the input string. The parser contains the rules on how these tokens

can be put together (much like production rules). An ANTLR4 parser can match six different pattern types. These are:

Sequence: ‘a finite or arbitrarily long sequence of tokens or subphrases’

Sequence with a terminator: ‘an arbitrarily long, potentially empty sequence of tokens or subphrases separated by a token’

Sequence with a separator: ‘a nonempty arbitrarily long sequence of tokens or sub-phrases separated by a token’

Choice: ‘a set of alternative phrases’

Token dependency: ‘the presence of one token requires the presence of one or more subsequent tokens’

Nested phrase: ‘a self-similar language structure’

*(All quotes from the above list come from [32])*

A basic example of a grammar in ANTLR4 is as shown in appendix listing 11. *expression* is the top-level (or entry point) rule which is the starting point for the ANTLR4 parser (like a start symbol), and parse trees are generated from here. Next the parser rules are listed, which consist of sequences of other parser rules or tokens. For example, the ‘expressions’ rule matches a ‘num’ rule and a (potentially empty) sequence of numbers (from the ‘num’ rule). This rule matches the sequence with separator pattern type.

This grammar file can be used to generate a parser in the chosen output language (by default this is Java). It can then be used with the ANTLR4 test rig to list the tokenized input string (shown in figure 5a) and produce a visual parse tree (shown in figure 5b).

ANTLR4 stands as a robust and comprehensive parser generator, capable of generating parsers for a variety of popular formal languages. One such example of this is the implementation of C++ as an ANTLR4 grammar. As noted earlier, C++ has a formal language specification. An iteration of the ‘ISO/IEC 14882:2014’ C++ specification has been implemented in ANTLR4 [18], demonstrating ANTLR4’s capabilities.

Using ANTLR4 for the development of a parser comes with several advantages. The primary advantage of using ANTLR4 is its detailed documentation written by Prof. Parr in ‘The Definitive ANTLR4 Reference’ [32]. The textbook details everything that is needed to create an ANTLR4 grammar as well as examples of grammars that have already been implemented.

Another advantage is the plethora of languages that parsers can be generated in. Allowing the developer the choice of different languages enables them to make a diverse set of programs and take advantage of different languages.

ANTLR4 is also free to use and open source (source code available to be viewed at [4]). This encourages developers to use and contribute to the development of ANTLR4.

ANTLR4 grammar files are concise and easy to write. Allowing developers to quickly generate formal specification for a language means that the in-depth particulars about a language can be studied closely. ANTLR4’s easy parser generation means that prototyping and testing a formal language is quick and simple. In the development of LexiTrain, a short bash script was written (appendix listing 3) that could generate and test a parser quickly, allowing the generation of

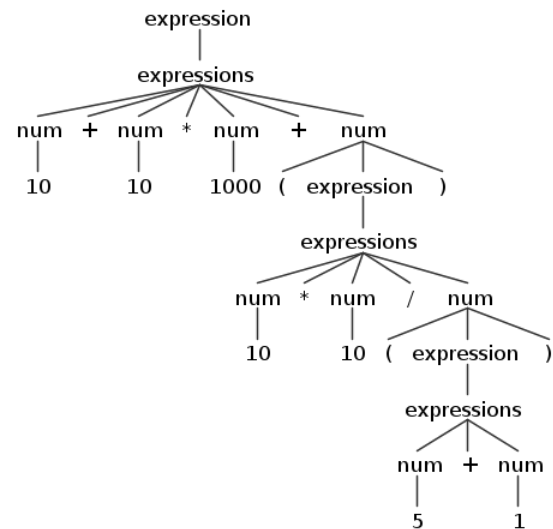


```

[ @0,0:1='10',<NUMBER>,1:0]
[ @1,2:2='+',<'+'>,1:2]
[ @2,3:4='10',<NUMBER>,1:3]
[ @3,5:5='*',<'*'>,1:5]
[ @4,6:9='1000',<NUMBER>,1:6]
[ @5,10:10='+',<'+'>,1:10]
[ @6,11:11='(',<'('>,1:11]
[ @7,12:13='10',<NUMBER>,1:12]
[ @8,15:15='*',<'*'>,1:15]
[ @9,17:18='10',<NUMBER>,1:17]
[ @10,20:20='/',<'/'>,1:20]
[ @11,22:22='(',<'('>,1:22]
[ @12,23:23='5',<NUMBER>,1:23]
[ @13,25:25='+',<'+'>,1:25]
[ @14,27:27='1',<NUMBER>,1:27]
[ @15,28:28=')',<'>',1:28]
[ @16,29:29=')',<'>',1:29]
[ @17,30:29='<EOF>',<EOF>,1:30]

```

(a) Input tokenized by the ANTLR4 test rig



(b) A parse tree generated by ANTLR4

Figure 5: Tokenization and Parse Tree for the input: ‘10 + 10 \* 1000 + (10 \* 10 / (5 + 1))’

visual parse trees throughout development.

A disadvantage of using ANTLR4 to generate parsers is the lack of customisation compared with a parser that was written by hand. Errors can often be less specific (although ANTLR provides a way to change errors) and the number and length of files generated can be excessive.

## 2.15 Web Development Tools

Svelte is a modern JavaScript framework used for building powerful user interfaces. Svelte compiles its files to specialized code that manipulates the DOM directly. Svelte’s architecture revolves around component-based development, facilitating the creation of highly efficient front-end applications.

Within Svelte, each component integrates script code, styling code, and the HTML markup in the same file, simplifying the development workflow and drastically reducing the need for boilerplate code.

Svelte provides support for TypeScript to be written directly within Svelte components.

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript. It combines the versatility of JavaScript with the specification of static typing, allowing developers to write more robust and maintainable code. Using TypeScript means that data types can be strictly defined, lending itself to interacting with the specification of a formal language. TypeScript is transpiled to plain JavaScript before deployment, meaning that the user’s browser has only to run JavaScript code that is widely supported by most modern browsers.

### 3 General structure of LexiTrain

To complete the objectives of the project, several key components had to be completed. These components, illustrated in the project aims, consisted of the language design, a formal language specification, a web calendar to display and test the formal language (including the design of the web calendar), compilers to convert the LexiTrain language into different file formats, and documentation to guide users through the language's utilization.

#### 3.1 Language Design

The first stage of the project was to design the formal language. This included creating examples of how the language would be used and to test more complex ideas for the language's functionality.

Basic rules and tokens were designed to match the chosen structure and written examples that had been researched. This stage highlighted any flaws or missing rules and meant that alongside the completed research, the language's requirements were full understood.

For more complex sessions, when designing LexiTrain it was decided that writing would be more user friendly if the author could refactor in-line sessions in to separate files.

This change means that LexiTrain files are split into period files (file extension *.lt*) and session files (file extension *.slt*).

Another consideration in the design phase for the language was the creation of a parser for the import section of a session file. This parser would discard all parts of the input that weren't import statements. This means that a period file can quickly be scanned for the necessary session files.

This is required as to compile the period file, any imported session files also must be parsed, valid, and then moved into the period file.

This change means that a period file can be submitted with syntactical errors that won't be checked until a full parse is completed. However, without the imported session files, a period file could also be syntactically wrong. Therefore, a preliminary parse was decided upon.

As mentioned in the background research, the formal language specification would be made using ANTLR4's grammar files. Writing the specification directly in ANTLR4 grammar files means that it can be used with the parser generator that comes with ANTLR4 without any additional manipulation.

To create a grammar file, a lexer and parser are needed. These can be written in the same file or split across multiple files. In this project, the lexer and parser are split into separate files for code readability and potential reuse of the lexer.

As stated in the design phase, a parse for just imports is necessary. With the addition of session files, another separate parse is required for each session file to provide all the data required for full compilation of the period file.

Therefore, LexiTrain must be parsed in two or three parsing stages, although if no session imports are found in the period file, no session parses are required. LexiTrain therefore has three different grammars, one each for: a period file, a session file, and an imports search.



### **3.2 Web Calendar Compiler**

A web calendar was designed to demonstrate use of the language in a real-world application of value to athletes and coaches. The calendar asks the user to input a period file to be compiled. Using the import grammar, it then checks what session files are imported and required, prompting the user to input these as well. Once all required session files have been input, the website compiles the given input into a calendar format, displaying the required training session on different days in the month.

These input requesting stages are similar in the other compiler.

The calendar is a demonstration of how the LexiTrain language could be used with other training platforms (like those mentioned in the background research).

A user looking to write training could use the LexiTrain language, uploading the files to their chosen training platform for compilation.

### **3.3 Portable Document Format Compiler**

A compiler for LexiTrain language to Portable Document Format (PDF) format has also been created. This allows users to upload written period files (and their required session files) to be converted into a static training plan.

Along with the web-based training calendar, this completes Aim Two in the project's initial document. No other compilers were deemed necessary. Initially, the language was intended to be compiled to other formats like .ERG or .ZWO. However, these further formats were avoided because they were not relevant to the final LexiTrain scope. After designing the language, it was clear that the data needed to create a training plan was not the same data as needed for these file formats. Adding the requirement for this data in the language would have added parts that are redundant in most cases and only used for these compilers. Keeping the language free of irrelevant data makes it easier to understand and write.

The file formats presented in the initial document are not equivalent to the training plans created by LexiTrain. These file formats are a different part of training platforms that LexiTrain does not cover as it is irrelevant to the overall goal of the project. Initially, it was stated that sessions would be converted into different file formats to be used on other training platforms. However, with the scope of the project moving to the creation of training periods, it was deemed not necessary to allow users to compile session files alone.

### **3.4 Language Documentation**

Aim four in the initial document was the creation of 'a well-structured set of documentation for the training plan language'. This documentation has been created and is hosted along side the web training calendar. The documentation is written in simple terms in order to be accessible to all users of LexiTrain at whatever level of competency. The documentation contains details of each rule and examples of their use.

## 4 Details of Implementation

### 4.1 Project Design

The first stage in the implementation was the design of the project. This stage consisted of designing example use case files, the web calendar, and the language. This address different aims in one overarching phase.

#### 4.1.1 Language Design

For the example file creation in the LexiTrain language design, some training periods were created. These periods are based on my actual training, to give in insight into LexiTrain's use in real-world training scenarios. In addition to these real-life periods, some examples were also created to test LexiTrain's full rule set including unusual and edge-case usages of the language.

One example was based on my third Ironman mesocycle to be completed through April 2024. This training plan can be seen in appendix figure 11 as it was written in Intervals.Icu and can be seen in appendix listing 4 as a LexiTrain period file. The example period file includes a number of microcycles (weeks), spanning a single mesocycle (three weeks), within the macrocycle (a year).

The period includes twenty-two sessions split between four different activity types. These activity types are: swimming, cycling, running, and triathlon.

In Intervals.icu these are displayed as icons and in a LexiTrain period file, they are written between parenthesis (e.g. (*run*)).

These are valid sport types in LexiTrain and will be approved during compilation. By restricting the sports that a user can enter, it ensures that data for each sport has been considered and can be correctly displayed. This is not checked during parsing as syntactically any *WORD* is correct.

The example also includes notes on some days. This is used to inform the athlete about training and can be a form of communication from coach to athlete. Using notes, the long sessions of the week have been labelled as such, letting the athlete know the type of session to expect.

Writing notes on any part of training in a LexiTrain period file just requires the user to write *note=* followed by the note as a single word or a string in speech marks.

The mesocycle also has a recurring *base* run on a Thursday. In intervals.icu that is written as a separate workout each time. In a LexiTrain period file, this workout is imported, making use of the separate session files, and only the import name must be written (they syntax is the import name in square brackets e.g. [*run\_base*]).

In Intervals.icu, the calculated (or specified) session load is displayed alongside each workout. In LexiTrain, this can be specified or left blank. Depending on the compilation type, a load can be calculated based on the workloads given with the session description or specified (greatly shortening the file in the former case).

As previously stated, session and period files can be split. The imported *run\_base* was created as a session file for the testing of session parsing and for testing in use within period files. This session can be seen in appendix listing 5. It includes some metadata at the top of the file and then the sections of the sessions. These sections are split into *warmup*, *main*, and *cool down w/ strides*. Using an imported session removes the need to rewrite the same common session every time it is needed. It also makes the period file shorter and more readable.

### 4.1.2 Website Design

After the period and session example had been created, mock-ups were created for how the session files may look once compiled. This involved creating website designs in the design tool Figma (<https://www.figma.com/>). In figure 6, the design displays the example period file in a web calendar design.

April ← →





















Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
01	02 Long ride road 4 hours load 120 	03 Swim lesson 30mins load 20 	04 base 1hr 15mins load 50 	05 Long swim 1hr 30mins load 70 	06 base 1hr30mins load 75 Tech swim 1hr load 40 	07 Long run 2hrs 30mins 160TSS 
08 Note: Slight deload week with varsity. Could cycle to watch cycling varsity?	09 Long ride road 3 hours load 100 	10 Varsity Race! 60mins load 60 	11 base 1hr 15mins load 50 	12 Long swim 1hr 30mins load 70 	13 base 1 hour load 75 Tech swim 1hr load 40 	14 Long run 2hrs 30mins 160TSS 
15	16 Long ride road 4 hours load 120 	17 Swim lesson 30mins load 20 	18 base 1hr 15mins load 50 	19 Long swim 1hr 30mins load 70 	20 base 1hr30mins load 75 Tech swim 1hr load 40 	21 Long run 2hrs 30mins 160TSS 
22	23 Long ride road 4 hours load 120 	24 Tech work 30mins load 20 	25 base 1hr 15mins load 50 	26 base 1hr load 60 Long swim 1hr 30mins load 40 	27 base 1hr30mins load 75 Tech swim 1hr load 40 	28 BUCS sprint tri load 60 
29	30					

Figure 6: Design made in Figma for a web calendar displaying compiled training

Each day in the calendar displays the sessions that require completion by the athlete. This information is presented with the name of the session, the duration of the session, the session load (calculated or submitted), and the sport type displayed as an icon. The calendar can be navigated to display a month at a time.

### 4.1.3 Documentation Design

In addition to the design of the calendar, a basic design for how the documentation may be structured and styled was also created. This can be seen in figure 7

The documentation design describes a part of a file (the metadata) with a brief description. It also defines metadata as that term could be confusing for users who are unfamiliar with it. The design also shows a code example of the documentation page's focus (metadata).

### 4.1.4 Rule Design

The main part of the design phase for the project was to design the language itself. With reference to the examples written above, the syntax for some basic rules could be designed.

The first rule was for how a single period would be expressed. A period in a LexiTrain file is equivalent to a microcycle, containing days and the training sessions to be completed on those days. A group of periods (i.e. the contents of a period file <sup>1</sup>) makes up a mesocycle. A period

<sup>1</sup>This confusing nomenclature derives from a divided naming system in the researched literature, multiple periods here can be expressed as a singular period

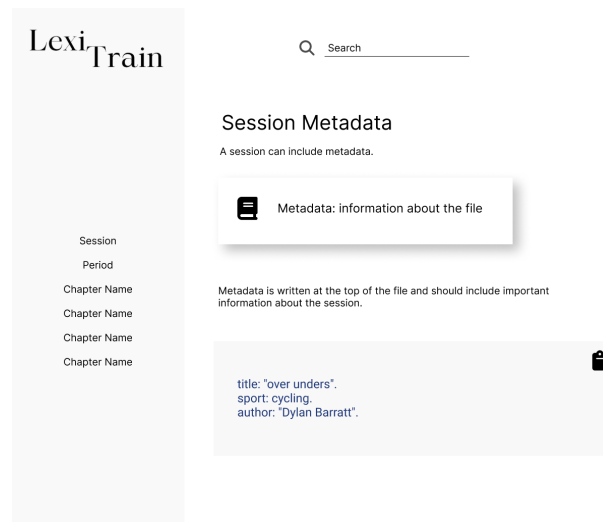


Figure 7: Design made in Figma for a documentation page

should be nameable (for example *week one* or *training camp*) to give the athlete an idea of what the week may entail. A list of days should be contained within this period.

The second rule was for a day. A day is one part of a period and contains sessions to be completed. There are three ways to express a day in LexiTrain: explicitly, inexplicitly, or repetitively.

**Explicitly** - the day is written with the day of the week it should be completed on, for example *Monday*.

**Inexplicitly** - the sessions for that day can be completed by the athlete at any point in the period.

**Repetitively** - an inexplicit way of setting sessions but denotes that a session should be completed multiple times during a period. For example, *3 \* [base\_run]* meaning at some point in the period, the athlete has to complete three *[base\_run]* sessions. This allows authors to write a session once that may appear inexplicitly in a period multiple times.

A day is made up of a non-empty list of sessions to be completed.

The final rule was to describe the sessions in a day. The day data is split into four types: imported, workout, session, and note.

**Imported** - tells the compiler that the specified imported session name should be used here.

**Workout** - a simple session, only including the session's basic information like time, intensity, and load.

**Session** - an in-line session file. It has the same layout as a session file but is written within the period file.

**Note** - a place where the author can tell an athlete something in their training plan on a specific day, for example “*Rest Day!*”.

The option to write an unspecified day lets the author detail that a session must be completed on any day in that period, which provides the athlete with flexibility.

#### 4.1.5 Backus-Naur Form Design

Backus-Naur form (BNF) rules are very similar to writing ANTLR4 grammar rules. Therefore, in addition to the written descriptions of the above rules, BNF definitions were made.

The first rule can be seen in appendix listing 9.

It defines the period name as a `<word>` followed by curly braces and comma separated list of days (at least one day). A `<word>` is defined as a single word without spaces (*hello!*) or any characters within speech marks (“*Hello World!!*”). The requirement for speech marks is so that in all other places in the code, white space can be ignored.

The second rule is for a `<day>`, it can be seen in appendix listing 10

The three types of day are split across the three BNF choices. The first type uses a `<word>` to write the explicit day. Provided this input is a word, it is syntactically correct. However, if the word is not a valid day of the week the code is logically incorrect. This logic must therefore be checked in the compilation stage. An inexplicit day is defined without a word before the day data. A repeated day is defined with a double ampersand separated list of day data.

The final rule designed in BNF was the `<dayData>` rule, it can be seen in appendix listing 19. Each choice in `<dayData>` is denoted by another rule in the grammar. This is because these rules can be complex and splitting them into separate parts of the file makes the grammar more readable. The only exception to this is the note which is simply denoted by “*note=*” followed with the note as a string. This rule is simple so has not been split.

## 4.2 Language Implementation

After the language had been designed, it could be created and tested.

When designing each grammar’s rules, it was important that the end input file’s readability was maintained. This meant that rules should be descriptive but not excessive with syntax requirements and keyword annotations. Keeping the final period and session files concise means that the language is easier to write for someone without an extensive knowledge of LexiTrain, while also maintaining all the required training features (and the syntax to describe these features) for authors looking to write more complex training plans.

Using ANTLR4’s grammar files, the set of rules that define a LexiTrain language file could be specified. As it was decided that periods, sessions, and import scanning were to be split into separate parsing stages, three different grammars were needed.

However, using a single lexer file, common tokens through the three languages do not need to be redefined. This base lexer can be seen in appendix listing 6.

### 4.2.1 Tokenization

The first tokens in the base lexer are symbols with a specific meaning in the language. These are `<`, `>`, and `-`. These symbols are restricted for use with intensity descriptions.

The second part of the lexer tokenizes keywords in the code. These are *import*, *load=*, and *note=*. These keywords are used for special statements in the code that are to be compiled in a different way. For example, a note can be placed in most places in the code and denotes that the closest code block should be supplemented with *WORD* following the note statement.

The lexer then tokenizes types of data based on their surrounding brackets. A *SPORT* is defined between `()` and an imported session between `[]`.

The final tokens are *NUM* and *WORD*. These tokens are the building blocks of the language. *NUM* matches any length positive number. *WORD* matches any single string without spaces or any string between `"`. Comments and white space are ignored by the tokenizer, this is specified at the bottom of the lexer. Comments can be used to annotate the code with details that should not be compiled. White space is ignored so that the format of the file can be decided by the author. Ignoring white space also makes the code less syntactically strict. A fragment is used to allow the author to write restricted symbols in the *WORD* token between `"` if it is preceded by the escape key

.

### 4.2.2 Period File Grammar

The first grammar created was for the period file.

The structure of a period file is as follows: Metadata, session imports, periods.

The metadata in a period file can be used to store details about the file. A period can contain any number (including zero) of metadata lines. The syntax is in appendix listing 20.

Metadata is defined as the type (or tag) on the left side of the colon and the data to store on the right. Multiple lines of metadata are split by the period at the end of each. Syntactically, metadata will be parsed as long as it is in this form. However, only a limited amount of metadata types are supported logically. These are *title*, *author*, *date*, *start\_date*, and *end\_date*. This is validated during the compilation stage. Restricting the metadata types means that it can be ensured that the data is properly displayed in the final compiled form. It also gives authors an idea of what they should enter in the Metadata.

The next part of a period file is the session import (appendix listing 21).

Syntactically it is the import token followed by a *WORD* storing the name of an imported file. Multiple files can be imported in a single period file and each import line is separated by a period at the end.

The final parts of a period file are the periods themselves. These rules were implemented based on the prior designs and BNF rules. The rule can be seen in appendix listing 22.

Using a *WORD* to store the name of the period and a non-empty comma separated list of days, a day is defined as in appendix listing 23.

This is a simplified version of the BNF designed rule. During implementation, a repeated token

match was replaced with *dayLoop*. This made the grammar file cleaner with less unnecessary code.

A *day* from the *dayLoop* is defined in appendix listing 24

This rule format is almost identical to the BNF design. Each choice is a different rule in the grammar file.

The *imported* rule matches the *IMPORTED* token in the base lexer.

The *note* rule is the *NOTE* token in the base lexer.

*note* and *imported* have been separated into rules so that a listener function is generated for each. This means the rules are given nodes in the parse tree that can be used to extract data during a parse tree walk.

*workout* and *session* are longer rules. This is defined in appendix listing 25

A *workout* is defined by the *SPORT* token and a *workloadL* rule. The *workloadL* rule is a list of workloads that describe a session (appendix listing 26).

A *workloadL* can be empty or contain a *workload*, load number, or note.

The *workload* matches an amount of time and an intensity description. The amount of time is syntactically just a *WORD*, and logically it must be a valid time, checked at compilation. A valid time format has the hour then minute with hour written as “hour”, “hours”, “hrs”, or “hr” and minute written as “minute”, “minutes”, “min”, or “mins”.

An intensity description is a valid zone from one of the three presented zone models in section 2.6.1. The zone is a *WORD* that is logically checked at compilation to be a zone from one of the these intensity models.

A zone is then described with an intensity range. There are five types of intensity ranges. These are: unspecified, less than, greater than, between, and at.

The intensity ranges are defined as follows:

**Unspecified** - This workload is just an amount of time with no zone given.

**Less than** - This workload is to be completed at less than the specified zone.

**Greater than** - This workload is to be completed at higher than the specified zone.

**Between** - This workload is to be completed between two zones.

**At** - This workload is to be completed at a specified zone.

The final choice for *dayData* is an in-line session. This in appendix listing 27.

A *session* (representing an in-line session) is defined between braces *{}*. It has a sport (a token that captures a *WORD* between *()*) and one or more sections. A section (*sessionSection*) requires a name using a *WORD* token and a workload. This workload can either be one single *workloads* (a double ampersand separated list of the previously defined *workloadL*) or a repeated number of *workloads*, with the number of iterations as a *NUM* token. A repeated set of workloads allows the author to shorten repeated code to only one block.

### 4.2.3 Session File Grammar

The next created grammar was for the session file. Based on the requirements of the session example (appendix listing 5), the designed file structure for a session file is: metadata and the session's sections.

The session grammar also uses the base lexer seen in appendix listing 6.

The metadata in a session file is analogous to the period file equivalent, with the only difference being the logically correct metadata types. These are: *title*, *author*, *sport* (required), *load*, and *note*. The metadata is used to describe information about the session file to be used in compilation. The *sport* metadata is required so that the session can be compiled properly (the same as *sport* being required for session definitions in a period file). The grammar rule is the same as the *metaData* rule (listing 20).

A section in a session is a grouped set of workloads. A section requires a name, for example "warm up". A section can be suffixed with a note to describe the session (appendix listing 28).

*sectionContents* is a set of workloads. Workloads are separated by double ampersands. A workload can either be a single workload or a repeated workload (a *structure*) (appendix listing 29).

*workload* has the same definition as in the period file grammar (listing 26).

A structure captures the amount of repetitions as a *NUM*. The repetition number can be used in compilation to display the workloads multiple times.

### 4.2.4 Import Grammar

The final grammar created was for the session imports. This grammar is used to match only the import lines in a period file. This allows the compilation to determine what session files need to be uploaded in order for a successful full parse without needing to run the full period grammar parse. The code can be seen in appendix listing 30.

The grammar does not use all the base tokens used in the other two as most are unnecessary. It only matches the import keyword, followed by the same token recognition as *WORD*, ended with a period. Everything else in the file is discarded.

Using these completed grammars, ANTLR4 generates token matching and parse trees for a given input string. It also provides default error messages for syntax errors, which is useful for testing.

## 4.3 Compilation

The next stage was the creation of the compiler to a web-based training calendar. The order of this differs from the initial document's Gantt chart. During the creation of this compiler, I realised that the first stage of turning the parse tree into a data object would be very similar for any other compilers. I therefore decided that making one compiler to a high standard and then using it in a real-life scenario (an actual training calendar) would provide a blueprint for other compilers, making the process of writing subsequent compilers shorter and more straightforward.



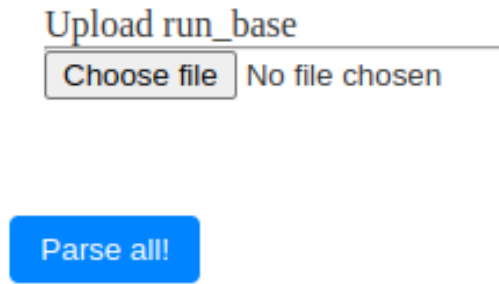


Figure 8: A session import file requested

This compiler is the most complete of any possible created compilers and therefore stores the most parsed data.

The compiler for the web-based training calendar was first written in JavaScript. ANTLR4 can generate parsers in JavaScript so creating a compiler in this language was very straightforward. However, during development, the lack of strict typing in JavaScript meant that the code development became convoluted with undefined data structures and type mismatching errors. To avoid this problem, the compiler language was changed to TypeScript. Using TypeScript, all the data structures to contain parsed data could be clearly defined (example in appendix listing 31).

TypeScript type classes can be used within each other as seen in *Sections*, an array of a type class *Section*.

TypeScript also allows Enums to be used. This makes it easier to document distinct choices for a type (example in appendix listing 32).

The compilation of a period file uses the parse tree generated with the grammar file from a user input string. ANTLR4 generates listeners to walk through the generated parse tree and react to events triggered by entering and exiting rules. Each rule in a grammar has a generated abstract (or empty for non object orientated languages) enter and exit function. The generated methods can be overwritten to implement custom functionality (example in appendix listing 33).

#### 4.3.1 Web Calendar Compiler

Using the generated listener for the import grammar on inputted period files, the functions are changed so that each import line found is saved.

The user is then requested to import the necessary session files. This can be seen in figure 8. Once submitted, each imported session is parsed and checked for correct syntax and logic. If it passes, it is stored in a data object. Then the period file can be fully parsed and checked for any syntactical errors. If there are none, the entire parse tree has been walked, completing the data needed to generate a calendar. This is handled by a Svelte front end.

During the parse tree generation, any syntax errors are caught by the ANTLR4 generated files and subsequently caught by the Svelte front end. In the case of any errors, the data is not compiled, and the user is provided with the relevant error.

The logic checking for user input is handled during the parse tree walk before data is saved. One example of this is matching the intensity zone in a workload to one from the three intensity models presented in the background research. This is done using a TypeScript defined enumerated type (enum), seen in appendix listing 34, and a helper function to convert an intensity zone value string to a value in this enum. If the string does not match any intensity zone, an error can be thrown here and compilation stopped.

Once the compiled data is complete, it is then processed using a Svelte front end.

In the initial planning of the project, the front end was to be made using React. However, with experience and further research, it was decided that the development lifecycle and code structure that makes a Svelte application was more relevant to how LexiTrain is compiled. Svelte works in combination with the TypeScript library files (the listeners). Svelte greatly simplifies the process of creating web UI alongside functional code, with support for TypeScript, allowing seamless integration of the created libraries.

Once all the parse tree's nodes have been visited, a TypeScript function flattens the generated data object and assigns each day an actual date. This flattened array can then be searched by the calendar for days in the currently displayed month.

The start date is used to date all the other days. It is calculated in one of four ways:

1. If the user has specified a start date in the metadata, this is used.
2. If the user has specified an end date, a start date can be calculated as the end date minus the number of days (including empty days).
3. If no date is specified, a start date is calculated using the first specified day name, setting the start date as the next occurring instance of this day of the week after the date of compilation.
4. If no training is on a specified day, the day of compilation is used.

The flattened array of dated days is passed to a Svelte calendar component.

The current month is shown to the user on page load. If any days from the array of days is in this month, it is displayed to the user. The user can navigate between months, with data being reloaded each month.

The design of the calendar is very similar to the initial design presented in the design phase (figure 6). The only change is how days are presented. The initial design was crowded with data on each day. The final web calendar simply shows the icons for the sports to be completed that day (figure 9). Each day can then be selected for more information about the sessions on that day. Longer sessions, such as those that may come from a session file, can therefore be presented in full detail.








Mon	Tue	Wed
29	30 	01 
06 	07 	08   

Figure 9: Sport icons shown for different days in the calendar

### 4.3.2 Client Side vs Server Side Compilation

The compilation of the LexiTrain files is done entirely on the client side. When creating the scripts to compile the files, both server side and client side compilation were considered.

The main benefit for compiling in the client's web browser is that the hosting of LexiTrain does not rely on a computationally powerful server. This is cheaper to deploy as only a site hosting server is needed.

Using the user's web browser for compilation also means that the server is protected from potentially malicious code being uploaded by a user.

The downside of completing compilation on the client's side is that the speed of compilation is dependent on the user's hardware:

- Firstly, the scripts need to be received by the user's browser. The amount of time this takes depends on the user's network speed.
- Secondly, once the user has the scripts, the web browser needs to run them. This depends on the user's hardware processing speed.

The benefit of using a server to host compilation is that data is compiled on the server. This means that if a computationally powerful server is used, the compilation can be fast. The user would then only be sent the final compiled data to be displayed (which still depends on the user's browser speed).

A server can slow down quickly if lots of users are requesting data at the same time. This means that the scope of server hardware may need to be expanded to compensate a larger user base.

Ultimately, client side compilation was chosen because most modern computers and web browsers are capable of running the compilation scripts.

If the project is deployed, to aid with the speed of data transfer, the scripts are minified. This reduces the amount of data sent to the user's browser while maintaining the exact same functionality.

Once the project is complete, the compilation scripts can also be made public and distributed to users so that they can run them entirely on their own machine.

### 4.3.3 PDF Compiler

To compile the parsed data into a PDF, a similar process to the web calendar compilation was required. The only difference being the data is not flattened and given a date.

Therefore, before creating the PDF compiler, the functions that could be used in both compilers were moved to a TypeScript library file. They were then edited so that rather than changing data stored with a Svelte component, they took references to the data and returned updated versions. This code structure approach meant that the same functionality was not duplicated. However, it still requires the exact same data variable declarations to be repeated in different compilation Svelte components. The advantage of this approach is that it ensures no data aliasing due to each component having its own separate data rather than an instance of the library's data.

The PDF is generated using a node module called `jspdf` [24] with the plugin `jspdf-autotable` [25]. Using `jspdf`, a PDF file can be generated in JavaScript code. This meant that the PDF generating code could be written with the TypeScript compiler library.

With the `jspdf-autotable` plugin, tables can be generated from JavaScript arrays. Therefore, the final data object from the compilation functions could be compiled to a PDF.

This is done by converting each day of each period in the final data object to a string, storing them as an array. The array could then be given to `jspdf-autotable` to generate a table within the PDF. The period's metadata could then be suffixed and a final PDF generated.

This approach was straightforward and didn't require any interaction with raw PDF code as that was handled by the node modules.

## 4.4 Documentation

The documentation for LexiTrain is written using Markdown. Markdown is a lightweight markup language that simplifies formatting of plain text, allowing the creation of structured documents. Markdown is simple to write and has plenty of information online to help with development.

The documentation consists of four main Markdown files, each corresponding to a different page: index, period file, session file, and tokens.

The index page is a landing page for any users viewing the documentation, which tells the user how to get started writing LexiTrain.

The period file page describes the structure of a period file and how to write each part in the structure. It also includes the details for valid syntax and logic (such as valid sport types). This can be viewed in appendix listing 12.

The session file page describes the structure of a session file, including its specific metadata and how to write each section in the session.

The tokens page describes tokens used by the formal language such as *WORD* and *NUM*. These tokens are referenced in other pages of the documentation and links to this page provide users with a full definition.

The Markdown is then compiled by Material for MkDocs [38]. Material for MkDocs is a documentation framework built on MkDocs, giving MkDocs generated files a unique style and other additional features like indexing and search functionality.

MkDocs is a static site generator that turns Markdown files into a static html website. The MkDocs generated site is deployed to a web server for the user to view the styled Markdown files. The generated period file page markdown file (appendix listing 12) can be seen in appendix figure 10.

## 5 Testing and Results

The completed compilers were tested against the examples made in the design in addition to some extra test cases to check edge case scenarios.

For both the compilers a correct compilation will have:

- All text matched to the correct tokens
- A complete parse tree with these tokens
- Correctly compiled file contents

Individually, each compiler was tested to check that the outputted format properly represents the given data.

For the calendar, the week must start on the correct day with data shown on the correct days of the week.

For the PDF, all the specified periods should be displayed in the final format.

During the first four tests, tokenization and parse tree generation for different inputs and grammars were tested.

When testing the tokenization of an input string, it is said to be correct if no characters are left over (not tokenized) once the *EOF* (end of file) token has been reached. This result means that the entire string has been tokenized in some form (not necessarily matching the correct rules syntax).

Checking the parse tree is tested to be correct if the given input matches the correct syntactical structure. The parse tree should also not include any tokens not matched to rules. This also validates that the string has been tokenized correctly as the token's syntax matches the defined rules.

As the tokenization and parsing steps are the same for both compilers and tested in test three, they are not tested again in tests five, six, and seven.

The last three test check that the tested input in test three is compiled into the correct format. This includes testing the code and file outputs.

### 5.1 Test One - Metadata

The first test completed was to check the period file metadata was parsed and partially compiled correctly. The tested period file can be seen in appendix listing 13.

The grammar correctly tokenized the input for test one (appendix figure 12a). The grammar also created the correct parse tree for the input in test one (appendix figure 12b).

The metadata was also checked in the first stage of compilation to ensure that it had been matched to the valid metadata types. This was a success and can be seen in appendix figure 13 as the output from a web browser's console and in appendix listing 14 as the actual stored data type. Overall, test one was a success.

## 5.2 Test Two - Imports and Session Files

The second test checked the importing of a session file, the session file parsing, and the import grammar. These test parts have been combined into one test due to the dependent nature of each one in the compilation stage. The test files can be seen in appendix listings 15 and 16. The period file (appendix listing 15) contains a single imported session and a period containing that session on two different days. The session file (appendix listing 16) contains some session metadata, including the required sport, and three workload sections.

The first part of the test was the tokenization and parse tree generation of the period file. This was a success and can be seen in appendix figures 14a and 14b.

The second part of the test was the tokenization and parse tree generation of the session file. This was also a success and can be seen in appendix figures 15a and 15b.

The final part of the test was the parse tree generated by the import grammar. This was to ensure that the session import line had been found and put in the tree.

This was a success and can be seen in appendix figure 16

All three parts of test two succeeded and therefore the full test was successful.

## 5.3 Test Three - Meso Three Period Test

Test three uses the meso three example as well as the run base session file written in the design phase. This test checks that the language can handle a typical use scenario. The test file can be seen in appendix listings 4 and 17. The file includes valid metadata, valid imports, and three periods populated with valid and varying days. The import parsing was not tested individually in this test as the previous test satisfied this.

The tokenization and parse tree generated for the period was completed successfully and can be seen in appendix figures 18 and 19.

The tokenization and parse tree generated for the session was also successful and can be seen in appendix figures 20 and 21.

The test was a success overall, demonstrating LexiTrain's capability to describe a real-life training plan.

## 5.4 Test Four - Full Period Test

Test four examines the full functionality of a period file including all day types and edge cases. The test file (appendix listing 18) included metadata, an imported session, and two periods including all the different day types.

The period file's tokenization succeeded, and the generated parse tree matched the tokens to all the rules in the test file. Therefore, the test was a success.

The tokens can be seen in appendix figures 23a and 23b. The generated parse tree can be seen in appendix figure 22.

The results of tests one to four show that the parser is capable of capturing all the designed and described rules.

## 5.5 Test Five - Function Testing

Before testing the full compilation outputs, the functions written to compute them were tested. Test five was tested using TS-Jest [1]. TS-Jest is a TypeScript preprocessor for Jest, a popular testing framework for JavaScript applications.

Using TS-Jest allows tests to be written in TypeScript and tested using the Jest framework, enabling the use of TypeScript's features (including the custom data classes) while testing the code.

The *HelperFunction.ts* file contains seven functions to be tested. Sixteen tests were written to test these functions including test correct and incorrect inputs.

The test results are in appendix figure 24. The full test file code has not been included due to it being too long. The file can be seen in the projects source code under */WEB/src/lib/HelperFunction.test.ts*

Test five was a success and therefore ensures that the compilation in tests six and seven have the correct calculated data.

## 5.6 Test Six - Meso Three Calendar

Test six checked the compilation of the test three period and session files (appendix listings 4 and 17) into the web based calendar.

To test the correctness of the compiled calendar, it was compared to the data in the parse tree from test three (appendix figure 21) to ensure that all the data in each leaf was present in the calendar.

The compiled calendar can be seen in appendix figures 25 and 26 in addition to the details of a single day (the first Thursday, including an imported session file's data) in appendix figure 27.

The data displayed in the calendar is complete and includes all of the data in the leaves of the parse tree (appendix figure 21). This includes the specified metadata, session imports, and all three periods' day data including the imported session.

The calendar also starts on the correct first day, with each subsequent day also being displayed properly, including across multiple months. Therefore, test six was a success.

## 5.7 Test Seven - Meso Three PDF

Test seven is also tests a compiled version of test three's period file and session file (appendix listings 4 and 17), looking at the compilation of the files into a PDF form. To test that the compiled PDF is correct, again the compiled version was checked against the data in the parse tree from test three (appendix figure 21).

The compiled PDF can be seen in appendix figure 28.

The PDF contains all the data from the parse tree including the relevant metadata and each period's sessions. It is therefore correct and test seven was a success.

All seven tests in the testing fulfilled the testing criteria. This ensures that all stages of the compilation and language creation are correct and fit the designed language. The tests also included tests that challenge examples from the language's real life use case by testing a period of training written for an athlete. The tests also include complete compiled versions of this written training, demonstrating LexiTrain's capability.

## 6 Conclusion

The original aim of this project was to create a simple and well documented language for the creation of endurance sport training plans. The project was to include a formal language that training planning could be expressed in, compilers for training plan display formats, and documentation.

The formal language created for the project can concisely detail complex training plans.

In the background research, a study was presented that detailed three of the most common methodologies for elite athlete's training plan structure.

Through testing the language's use cases and comprehensiveness, LexiTrain was shown to be capable of describing these methodologies. The *meso three* example (test three 5.3) fits the polarised training methodology and the edge case and full period file testing (test four 5.4) demonstrate the flexibility of the training description to fit the other two methodologies. By giving authors the ability to define workloads using the intensity zones from the three researched models, the training plans are able to conform to the intensity distribution in these established methodologies.

### 6.1 Project Development Lifecycle

In the initial document, the chosen software development life cycle (SDLC) for the project was the Waterfall model. The Waterfall model is an SDLC completed in sequences, each stage following the completion of the last [5]. This SDLC was followed throughout the implementation of the project as all components of the project were completed sequentially.

A Gantt chart was also proposed for the project lifecycle in the initial document. This Gantt chart mapped the different project stages into a Waterfall sequence from the start of the project's development to its completion. During the development of the project, less work was completed than expected during January. This was due to a high volume of other academic work. However, the Gantt chart accounted for this eventuality with additional time as contingency. This was therefore not a problem and development continued smoothly and finished within the allotted timeframe.



## 6.2 Critical Analysis

Overall the project was on time with all of the aims completed.

However, with hindsight some changes to the project would be advisable.

The first change to be made would be to skip the amount of development the project undertaken in JavaScript. JavaScript was the initial language chosen to write the compilers and the majority of the web-based calendar compiler had been made in JavaScript before the change to TypeScript. JavaScript presented many unnecessary problems that TypeScript solved. Translating the JavaScript code to TypeScript was not an arduous process but took time (including debugging JavaScript errors) that could have been avoided had the code initially been written in TypeScript.

A further change that would have been beneficial is the addition of another compilation format that was relevant to LexiTrain's final usage. In the initial document, several possible compilation formats were found that during the development of the final project were found to be inappropriate. With more time to research another compilation format, LexiTrain's output could be used by an even wider user-base.

In the initial document, potential risks were highlighted and evaluated. Three of these risks were relevant to the final LexiTrain project. These were:

1. Project not completed on time
2. Loss of work
3. Extenuating Circumstances

The first risk was to be mitigated with a generous allotted time frame for the overall project. This mitigation was helpful as some stages in the development, namely the creation of the web-calendar compiler, took longer than expected. Due to this mitigation the project was still completed on time.

The second risk was to be mitigated with a regular backup of all work completed. Throughout the project this was done using Git. Both the dissertation document and entire source code for the project were regularly backed up to private repositories on GitHub. This also meant that previous versions of the work could be reverted to quickly when necessary.

The final risk was to also be mitigated with a generous allotted time frame. This risk was correctly marked as very unlikely, however during the project development an unexpected extenuating circumstance occurred and the project development was delayed. This was not a problem due to the successful mitigation and flexible timescale for the project.

If the project were to be continued with further resources, two main improvements would be made:

- The first improvement is to the overall aesthetic of the developed web site and compiled formats. The project focused on language development for writing training plans and did not have much scope for aspects like front end styling. This improvement could make the project more appealing for the end user.
- The second improvement would be to look for the support of the training platforms mentioned in the background research. LexiTrain could be integrated into their systems to re-

place or supplement their different training plan creation methods, furthering LexiTrain's overarching goal of simplifying training creation methods.

### **6.3 Contribution to Computer Science**

The LexiTrain project has created a new formal language to describe training planning in a way that has previously not been fully explored, as explained in the background research of both this document and the initial document. As a computer science project, the creation of this language in a sports science field has created a formal specification for writing training.

Existing training platforms lack well defined and documented ways of writing training plans and functionality. This may be because the platforms are made predominantly by sports scientists that have great expertise in creating and analysing planned training but lack formal computational knowledge. This manifests in the creation of in-depth but sluggish and suboptimal programs, as well as unpolished solutions to problems such as data storage formats.

LexiTrain addresses these gaps with strong documentation and a formal specification in the form of the language's grammar files. Using a common parser generator like ANTLR4 to write the language's rules in a grammar file means that the language is defined in a way that any computer scientist with an understanding of ANTLR4 could understand.

## References

- [1] *A Jest Transformer with Source Map Support That Lets You Use Jest to Test Projects Written in TypeScript*. | *Ts-Jest*. <https://kulshekhar.github.io/ts-jest/>. (Visited on 30/04/2024).
- [2] Hunter Allen, Andrew R. Coggan and Stephen McGregor. *Training and Racing with a Power Meter: Third Edition*. VeloPress, Apr. 2019. ISBN: 978-1-948006-10-1.
- [3] *Anatomy of a Compiler and The Tokenizer*. <https://www.cs.man.ac.uk/~pjj/farrell/comp3.html>. (Visited on 10/04/2024).
- [4] *Antlr/Antlr4: ANTLR (ANother Tool for Language Recognition) Is a Powerful Parser Generator for Reading, Processing, Executing, or Translating Structured Text or Binary Files*. <https://github.com/antlr/antlr4>. (Visited on 11/04/2024).
- [5] Youssef Bassil. *A Simulation Model for the Waterfall Software Development Life Cycle*. Comment: LACSC - Lebanese Association for Computational Sciences, <http://www.lacsc.org>; International Journal of Engineering & Technology, Vol. 2, No. 5, May 2012. May 2012. DOI: [10.48550/arXiv.1205.6904](https://doi.org/10.48550/arXiv.1205.6904). arXiv: [1205.6904](https://arxiv.org/abs/1205.6904) [cs]. (Visited on 14/04/2024).
- [6] *Block-Based Syntax from Context-Free Grammars* | *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. <https://dl.acm.org/doi/abs/10.1145/3426425.3426948>. (Visited on 09/04/2024).
- [7] Arturo Casado et al. ‘Training Periodization, Methods, Intensity Distribution, and Volume in Highly Trained and Elite Distance Runners: A Systematic Review’. In: *International Journal of Sports Physiology and Performance* 17.6 (Apr. 2022), pp. 820–833. ISSN: 1555-0273, 1555-0265. DOI: [10.1123/ij spp.2021-0435](https://doi.org/10.1123/ij spp.2021-0435). (Visited on 01/04/2024).
- [8] N. Chomsky. ‘Three Models for the Description of Language’. In: *IRE Transactions on Information Theory* 2.3 (Sept. 1956), pp. 113–124. ISSN: 2168-2712. DOI: [10.1109/TIT.1956.1056813](https://doi.org/10.1109/TIT.1956.1056813). (Visited on 09/04/2024).
- [9] Noam Chomsky. ‘On Certain Formal Properties of Grammars’. In: *Information and Control* 2.2 (June 1959), pp. 137–167. ISSN: 0019-9958. DOI: [10.1016/S0019-9958\(59\)90362-6](https://doi.org/10.1016/S0019-9958(59)90362-6). (Visited on 01/04/2024).
- [10] Armin Cremers and Seymour Ginsburg. ‘Context-Free Grammar Forms’. In: *Journal of Computer and System Sciences* 11.1 (Aug. 1975), pp. 86–117. ISSN: 0022-0000. DOI: [10.1016/S0022-0000\(75\)80051-1](https://doi.org/10.1016/S0022-0000(75)80051-1). (Visited on 09/04/2024).
- [11] Franklin L DeRemer. ‘Lexical Analysis’. In: *Compiler Construction: An Advanced Course* (1974), pp. 109–120.
- [12] Eimear Dolan et al. ‘Energy Constraint and Compensation: Insights from Endurance Athletes’. In: *Comparative Biochemistry and Physiology Part A: Molecular & Integrative Physiology* 285 (Nov. 2023), p. 111500. ISSN: 1095-6433. DOI: [10.1016/j.cbpa.2023.111500](https://doi.org/10.1016/j.cbpa.2023.111500). (Visited on 02/04/2024).
- [13] Gabriel Dos Reis and Bjarne Stroustrup. *A Formalism for C++*. Tech. rep. Technical Report, 2005.

- [14] Michael K. Drew and Caroline F. Finch. ‘The Relationship Between Training Load and Injury, Illness and Soreness: A Systematic and Literature Review’. In: *Sports Medicine* 46.6 (June 2016), pp. 861–883. ISSN: 1179-2035. DOI: [10.1007/s40279-015-0459-8](https://doi.org/10.1007/s40279-015-0459-8). (Visited on 02/04/2024).
- [15] Naroa Etxebarria, Iñigo Mujika and David Bruce Pyne. ‘Training and Competition Readiness in Triathlon’. In: *Sports* 7.5 (May 2019), p. 101. ISSN: 2075-4663. DOI: [10.3390/sports7050101](https://doi.org/10.3390/sports7050101). (Visited on 02/04/2024).
- [16] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. <https://www.w3.org/TR/xml/>. (Visited on 10/04/2024).
- [17] *Formal Languages and Compilation*. ISBN: 978-3-030-04878-5. (Visited on 09/04/2024).
- [18] *Grammars-v4/Cpp at Master · Antlr/Grammars-V4*. <https://github.com/antlr/grammars-v4/tree/master/cpp>. (Visited on 11/04/2024).
- [19] Christophe Hausswirth and Jeanick Brisswalter. ‘Strategies for Improving Performance in Long Duration Events’. In: *Sports Medicine* 38.11 (Nov. 2008), pp. 881–891. ISSN: 1179-2035. DOI: [10.2165/00007256-200838110-00001](https://doi.org/10.2165/00007256-200838110-00001). (Visited on 02/04/2024).
- [20] *How to Connect an ELEMNT, BOLT, or ROAM to a Desktop or Laptop Computer*. <https://support.wahoofitness.com/hc/en-us/articles/115000127910-How-to-connect-an-ELEMNT-BOLT-or-ROAM-to-a-desktop-or-laptop-computer>. (Visited on 06/04/2024).
- [21] Inside Exercise. #40 - *The History of Endurance Training Methods with Dr Michael Joyner*. Apr. 2023. (Visited on 03/04/2024).
- [22] IOC. *Tokyo 1964 5000m Men Results - Olympic Athletics*. <https://olympics.com/en/olympic-games/tokyo-1964/results/athletics/5000m-men>. (Visited on 03/04/2024).
- [23] Vladimir B. Issurin. ‘New Horizons for the Methodology and Physiology of Training Periodization’. In: *Sports Medicine* 40.3 (Mar. 2010), pp. 189–206. ISSN: 1179-2035. DOI: [10.2165/11319770-000000000-00000](https://doi.org/10.2165/11319770-000000000-00000). (Visited on 03/04/2024).
- [24] *Jspdf*. <https://www.npmjs.com/package/jspdf>. Jan. 2022. (Visited on 22/04/2024).
- [25] *Jspdf-Autotable*. <https://www.npmjs.com/package/jspdf-autotable>. Feb. 2024. (Visited on 22/04/2024).
- [26] Helmut Kopka and Patrick W. Daly. *Guide to LaTeX*. Pearson Education, Nov. 2003. ISBN: 978-0-321-61774-3.
- [27] Daniel D. McCracken and Edwin D. Reilly. ‘Backus-Naur Form (BNF)’. In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., Jan. 2003, pp. 129–131. ISBN: 978-0-470-86412-8. (Visited on 01/04/2024).
- [28] Sergio Medeiros, Fabio Mascarenhas and Roberto Ierusalimschy. ‘From Regular Expressions to Parsing Expression Grammars’. In: *Brazilian Symposium on Programming Languages*. 2011.
- [29] Burkhard Monien. ‘About the Derivation Languages of Grammars and Machines’. In: *Automata, Languages and Programming: Fourth Colloquium, University of Turku, Finland July 18–22, 1977* 4. Springer, 1977, pp. 337–351.

- [30] Fernando Nanclerio, Jeremy Moody and Mark Chapman. ‘Applied Periodisation: A Methodological Approach’. In: (May 2022). (Visited on 03/04/2024).
- [31] T. J. Parr and R. W. Quong. ‘ANTLR: A Predicated-LL(k) Parser Generator’. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. ISSN: 1097-024X. DOI: [10.1002/spe.4380250705](https://doi.org/10.1002/spe.4380250705). (Visited on 10/04/2024).
- [32] Terence Parr. ‘The Definitive ANTLR 4 Reference’. In: (2013), pp. 1–326. (Visited on 01/04/2024).
- [33] *Parsing Techniques*. ISBN: 978-0-387-20248-8. (Visited on 10/04/2024).
- [34] Arno Pauly and Swansea University. *Day Three Formal Grammar Intro: 2223\_CS-275\_Automata and Formal Language Theory*. (Visited on 19/04/2024).
- [35] Stephen Seiler. ‘What Is Best Practice for Training Intensity and Duration Distribution in Endurance Athletes?’ In: *International Journal of Sports Physiology and Performance* 5.3 (Sept. 2010), pp. 276–291. ISSN: 1555-0273, 1555-0265. DOI: [10.1123/ij spp.5.3.276](https://doi.org/10.1123/ij spp.5.3.276). (Visited on 03/04/2024).
- [36] *Six Weeks Is All It Takes: Follow Our Couch to 5K Plan and Start Your New Running Habit Today*. <https://www.runnersworld.com/uk/training/5km/a760067/six-week-beginner-5k-schedule/>. Dec. 2023. (Visited on 04/04/2024).
- [37] Torbjørn Soligard et al. ‘How Much Is Too Much? (Part 1) International Olympic Committee Consensus Statement on Load in Sport and Risk of Injury’. In: *British Journal of Sports Medicine* 50.17 (Sept. 2016), pp. 1030–1041. ISSN: 0306-3674, 1473-0480. DOI: [10.1136/bjsports-2016-096581](https://doi.org/10.1136/bjsports-2016-096581). (Visited on 02/04/2024).
- [38] *Squidfunk/Mkdocs-Material: Documentation That Simply Works*. <https://github.com/squidfunk/mkdocs-material>. (Visited on 22/04/2024).
- [39] Espen Tønnessen, Jonny Hisdal and Bent R. Ronnestad. ‘Influence of Interval Training Frequency on Time-Trial Performance in Elite Endurance Athletes’. In: *International Journal of Environmental Research and Public Health* 17.9 (Jan. 2020), p. 3190. ISSN: 1660-4601. DOI: [10.3390/ijerph17093190](https://doi.org/10.3390/ijerph17093190). (Visited on 03/04/2024).

## A Appendix

Note: this is not a complete set of program code. The listings provided are to illustrate the writing process. Many of these listings will not work independently of the rest of the project's source code.

```
1 #!/bin/bash
2
3 rm -rf ./Java
4 mkdir ./Java
5
6 java -jar /usr/local/lib/antlr-4.13.1-complete.jar ./SessionFile.g4 -
    o ./Java -visitor
7 ls ./Java/*.java
8 echo "Session File Java files generated"
9
10 java -jar /usr/local/lib/antlr-4.13.1-complete.jar ./PeriodFile.g4 -o
    ./Java -visitor
11 ls ./Java/*.java
12 echo "Period File Java files generated"
13
14 java -jar /usr/local/lib/antlr-4.13.1-complete.jar ./Imports.g4 -o ./
    Java -visitor
15 ls ./Java/*.java
16 echo "Imports File Java files generated"
17
18 javac ./Java/$TYPE*.java
19 ls ./Java/
20 echo "Java files compiled"
21
22 TEST_FILE="../examples/period_imports"
23 TYPE="Imports"
24
25 echo "Testing $TEST_FILE"
26 cd Java
27 cp ../$TYPE.g4 ./
28 java -Xmx500M -cp "/usr/local/lib/antlr-4.13.1-complete.jar:
    $CLASSPATH" org.antlr.v4.gui.TestRig $TYPE file -tokens $TEST_FILE
29 java -Xmx500M -cp "/usr/local/lib/antlr-4.13.1-complete.jar:
    $CLASSPATH" org.antlr.v4.gui.TestRig $TYPE file -gui $TEST_FILE
30 rm $TYPE.g4
```

Listing 3: A bash script for the compilation and testing of the LexiTrain ANTLR4 grammar files

```
1 title: "IM meso 3".
2 author: "Dylan Barratt".
3
4 import run_base.
5
6 "week 1" {
```

```

7   Tue: (bike) 4hr < HRZ3 note="long ride road",
8   Wed: (swim) 30min load=20 note="swim lesson",
9   Thu: [run_base],
10  Fri: (swim) 1hr30min load=70 note="long swim",
11  Sat: (swim) 1hr load=40 note="tech swim" && (bike) 1hr30min HRZ2,
12  Sun: (run) 2hr30min < HRZ3 note="long run"
13 }
14
15
16 "week 2" {
17   Mon: note="Slight deload week with varsity. Could cycle to watch
18     cycling varsity?",
19   Tue: (bike) 3hr HRZ2,
20   Wed: (tri) 60min load=60 note="Varsity Race!",
21   Thu: [run_base],
22   Fri: (swim) 1hr30min load=70 note="long swim",
23   Sat: (swim) 1hr load=40 note="tech swim" && (bike) 1hr HRZ2,
24   Sun: (run) 2hr30min < HRZ3 note="long run"
25 }
26
27 "week 3" {
28   Tue: (bike) 4hr < HRZ3 note="long ride road",
29   Wed: (swim) 30min load=40 note="tech work",
30   Thu: [run_base],
31   Fri: (swim) 1hr load=50 note="long swim" && (bike) 1hr HRZ2,
32   Sat: (swim) 1hr load=40 note="tech swim" && (bike) 1hr30min HRZ2,
33   Sun: (tri) load=60 note="bucs sprint tri"
34 }

```

Listing 4: Meso three written in LexiTrain

```

1  title: "run base".
2  sport: "running".
3  author: "Dylan Barratt".
4  load: 50.
5
6  warmup {
7    15min HRZ1
8  }
9
10 main {
11   45min < HRZ3
12 }
13
14 note="include some strides"
15 "cool down w/ strides" {
16   15min < HRZ2
17 }

```

Listing 5: A session file detailing a run session

```

1  lexer grammar BaseLexer;
2
3  LT: '<';
4  GT: '>';
5  BW: '-';
6
7  IMPORT: 'import';
8  LOAD: 'load=';
9  NOTES: 'note=';
10
11 SPORT: '(' WORD ')';
12 IMPORTED: '[' WORD ']';
13
14 NUM: '[0-9]+';
15 WORD: '[a-zA-Z0-9_/\+ | ''' (ESC|.)*? '''';
16
17 LINE_COMMENT : '//'.*? '\r'? '\n' -> skip ;
18 WS : [ \t\r\n]+ -> skip ;
19
20 fragment ESC : '\\'' | '\\\\' ;

```

Listing 6: The base lexer for lexiTrain grammar files

```

1  [@0,0:4='title',<WORD>,1:0]
2  [@1,5:5=':',<':'>,1:5]
3  [@2,7:17='''IM meso 3''',<WORD>,1:7]
4  [@3,18:18='.',< '.'>,1:18]
5  [@4,20:25='author',<WORD>,2:0]
6  [@5,26:26=':',< ':'>,2:6]
7  [@6,28:42='''Dylan Barratt''',<WORD>,2:8]
8  [@7,43:43='.',< '.'>,2:23]
9  [@8,46:51='import',<'import'>,4:0]
10 [@9,53:60='run_base',<WORD>,4:7]
11 [@10,61:61='.',< '.'>,4:15]
12 [@11,64:71='''week 1''',<WORD>,6:0]
13 [@12,73:73='{',<'{'>,6:9]
14 [@13,76:78='Tue',<WORD>,7:1]
15 [@14,79:79=':',< ':'>,7:4]
16 [@15,81:86='(bike)',<SPORT>,7:6]
17 [@16,88:90='4hr',<WORD>,7:13]
18 [@17,92:92='<',<'<'>,7:17]
19 [@18,94:97='HRZ3',<WORD>,7:19]
20 [@19,99:103='note=',<'note='>,7:24]
21 [@20,104:119='''long ride road''',<WORD>,7:29]
22 [@21,120:120=',',< ','>,7:45]
23 [@22,123:125='Wed',<WORD>,8:1]
24 [@23,126:126=':',< ':'>,8:4]
25 [@24,128:133='(swim)',<SPORT>,8:6]
26 [@25,135:139='30min',<WORD>,8:13]

```



```

27 [ @26,141:145= 'load=' , < 'load=' > ,8:19]
28 [ @27,146:147= '20' , < NUM > ,8:24]
29 [ @28,149:153= 'note=' , < 'note=' > ,8:27]
30 [ @29,154:166= '''swim lesson''' , < WORD > ,8:32]
31 [ @30,167:167= ' , ' , < ' , ' > ,8:45]
32 [ @31,170:172= 'Thu' , < WORD > ,9:1]
33 [ @32,173:173= ' : ' , < ' : ' > ,9:4]
34 [ @33,175:184= '[run_base]' , < IMPORTED > ,9:6]
35 [ @34,185:185= ' , ' , < ' , ' > ,9:16]
36 [ @35,188:190= 'Fri' , < WORD > ,10:1]
37 [ @36,191:191= ' : ' , < ' : ' > ,10:4]
38 [ @37,193:198= '(swim)' , < SPORT > ,10:6]
39 [ @38,200:207= '1hr30min' , < WORD > ,10:13]
40 [ @39,209:213= 'load=' , < 'load=' > ,10:22]
41 [ @40,214:215= '70' , < NUM > ,10:27]
42 [ @41,217:221= 'note=' , < 'note=' > ,10:30]
43 [ @42,222:232= '''long swim''' , < WORD > ,10:35]
44 [ @43,233:233= ' , ' , < ' , ' > ,10:46]
45 [ @44,236:238= 'Sat' , < WORD > ,11:1]
46 [ @45,239:239= ' : ' , < ' : ' > ,11:4]
47 [ @46,241:246= '(swim)' , < SPORT > ,11:6]
48 [ @47,248:250= '1hr' , < WORD > ,11:13]
49 [ @48,252:256= 'load=' , < 'load=' > ,11:17]
50 [ @49,257:258= '40' , < NUM > ,11:22]
51 [ @50,260:264= 'note=' , < 'note=' > ,11:25]
52 [ @51,265:275= '''tech swim''' , < WORD > ,11:30]
53 [ @52,277:278= '&&' , < '&&' > ,11:42]
54 [ @53,280:285= '(bike)' , < SPORT > ,11:45]
55 [ @54,287:294= '1hr30min' , < WORD > ,11:52]
56 [ @55,296:299= 'HRZ2' , < WORD > ,11:61]
57 [ @56,300:300= ' , ' , < ' , ' > ,11:65]
58 [ @57,303:305= 'Sun' , < WORD > ,12:1]
59 [ @58,306:306= ' : ' , < ' : ' > ,12:4]
60 [ @59,308:312= '(run)' , < SPORT > ,12:6]
61 [ @60,314:321= '2hr30min' , < WORD > ,12:12]
62 [ @61,323:323= ' < ' , < ' < ' > ,12:21]
63 [ @62,325:328= 'HRZ3' , < WORD > ,12:23]
64 [ @63,330:334= 'note=' , < 'note=' > ,12:28]
65 [ @64,335:344= '''long run''' , < WORD > ,12:33]
66 [ @65,346:346= '}' , < '}' > ,13:0]

```

Listing 7: The first section (reduced due to excessive length) of the tokenized version of mesocycle three

```

1 <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2
3 <unsigned integer> ::= <digit>
4 | <unsigned integer> <digit>
5

```

```
6 <integer> ::= <unsigned integer>  
7   | + <unsigned integer>  
8   | - <unsigned integer>
```

Listing 8: An integer defined using BNF [\[27\]](#)

## A.1 BNF Rules

```
1 <period> ::= <word> '{' <day> (',' <day>)* '}'
```

Listing 9: A period written as BNF

```
1 <day> ::= <word> ":" <dayData> ("&&" <dayData>)*  
2 | <dayData> ("&&" <dayData>)*  
3 | <num> "*" "{" <dayData> ("&&" <dayData>)* "}"
```

Listing 10: A day written as BNF

```

1 <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2
3 <unsigned integer> ::= <digit>
4     | <unsigned integer> <digit>
5
6 <integer> ::= <unsigned integer>
7     | + <unsigned integer>
8     | - <unsigned integer>

```

Listing 11: An example ANTLR4 grammar file including both lexer and parser

```

1 # Period File
2 A period file is the main file in a LexiTrain training plan. The
   period file is used to describe all the training for the given
   training plan, split into different periods or [mesocycles](https
   ://www.trainingpeaks.com/blog/macrocycles-mesocycles-and-
   microcycles-understanding-the-3-cycles-of-periodization/?
   utm_source=instagram&utm_medium=social&utm_content=image&
   utm_campaign=athlete2023_performance).
3
4 A period file is made up of three parts.
5 These are: [metadata](#metadata), [session imports](#importing-
   session-files), and [periods](#periods).
6
7 A period file is saved with a '.lt' file extension.
8 For example: 'meso_3.lt'
9
10 ## Comments
11 Comments can be included in a period file with a '//'.
12 Any text after the '/' on the same line is ignored when the file is
   compiled.
13 Comments can be used to annotate parts of the code that only the
   author (or anyone with access to the source files can see).
14 ### Example Comment
15 ''//this is a comment! Hello World''
16
17 ## Metadata
18 *Metadata: information about the file*
19 Metadata is written at the top of the period file and should include
   important information about the session.
20 Metadata lines are made up of two parts. The tags and data.
21 Tags are not case sensitive.
22 Data must be in [WORD](Tokens.md#word) format
23
24 ### Valid Metadata Tags
25 File name ('title')<br>
26 Author ('author')<br>
27 Date written ('date')<br>
28 Starting date ('start_date')<br>

```

```

29 Ending date ( 'date_date ' )<br>
30 ‘ ‘ ‘

```

Listing 12: A bit of the markdown for the period file page in the documentation.

```

1 title: "Test File".
2 author: "Dylan Barratt".
3 date: 15/04/2024.
4 start_date: 15/04/2024. //note: both start and end are not valid

```

Listing 13: The period file used in test one

```

1 {
2   "Title": "Test File",
3   "Author": "Dylan Barratt",
4   "Date": "2024-04-14T23:00:00.000Z",
5   "Start_Date": "2024-04-14T23:00:00.000Z",
6   "End_Date": null,
7   "Start_date": "15/04/2024"
8 }

```

Listing 14: The data object generated from the metadata in test one

```

1 import test_session.
2
3 "week one" {
4   mon: [ test_session ],
5   [ test_session ]
6 }

```

Listing 15: Test two period file

```

1 title: test_session.
2 sport: run.
3
4 warmup {
5   20mins HRZ2
6 }
7
8 main {
9   1hr PWZ3
10 }
11
12 cooldown {
13   20mins HRZ2
14 }

```

Listing 16: Test two session file

```

1 title: "run base".
2 sport: "running".

```

```

3 author: "Dylan Barratt".
4 load: 50.
5
6 warmup {
7     15min HRZ1
8 }
9
10 main {
11     45min < HRZ3
12 }
13
14 note="include some strides"
15 "cool down w/ strides" {
16     15min < HRZ2
17 }

```

Listing 17: Test three session file

```

1 title: "title 123".
2
3 import test_session.
4
5 weekone {
6     (bike) 1hr HRZ2,
7     tue: (swim) 1hr HRZ1 load=40 note="test note",
8     3 * {(run) 30min},
9     sun: (bike) 1hr HRZ2 - HRZ3 && (run) 1hr < HRZ4,
10 }
11
12 "week one" {
13     mon: [test_session],
14     [test_session],
15     wed: {(run) warmup {1hr} main {1hr PWZ7} cooldown {30min PW2}}
16 }

```

Listing 18: Test four period file

```
1 <dayData> ::= <imported>
2 | <workout>
3 | <session>
4 | "note=" <WORD>
```

Listing 19: Day data written as BNF

## A.2 Grammar File Rule Examples

```
1  metaData: WORD ':' (WORD|NUM) '.';
```

Listing 20: Metadata grammar rule

```
1  sessionImport: IMPORT WORD '.';
```

Listing 21: Session import grammar rule

```
1  period: WORD '{' day (',' day)* ('')? '}' ;
```

Listing 22: Period grammar rule

```
1  day
2  : WORD ':' dayLoop //specified day
3  | NUM '*' '{' dayLoop '}' //looped days (unspecified day)
4  | dayLoop //non-specified day
5  ;
6
7  dayLoop: dayData ('&&' dayData)* ('&&')?;
```

Listing 23: Day grammar rule

```
1  dayData
2  : imported //imported
3  | workout //single line session
4  | session //inline session
5  | dayNotes
6  ;
7
8  dayNotes: NOTES WORD;
9
10 imported: IMPORTED;
```

Listing 24: Day data grammar rule

```
1  workout: SPORT workloadL;
2
3  session //inline session
4  : '{' SPORT (sessionSection)+ '}' //name of sport and the sections of
   the session
5  ;
```

Listing 25: Workout and session grammar rules

```
1  workloadL: (workload)? (LOAD NUM)? (NOTES WORD)?; // workload load
   notes (all are optional)
2
3  workload
4  : WORD // no specified intensity (just time)
5  | WORD lt
```



```

6 | WORD gt
7 | WORD between
8 | WORD WORD //at intensity
9 ;
10
11
12 lt : LT WORD; // < HRZ2
13 gt : GT WORD; // > HRZ2
14 between : WORD BW WORD; //HRZ1 - HRZ3

```

Listing 26: Workload rules

```

1 session //inline session
2 : '{' SPORT (sessionSection)+ '}'
3 ;
4
5 sessionSection: WORD '{' (workloads | NUM '*' '{' workloads '}') '}'
6 ;
7 workloads: workloadL ('&&' workloadL)* ('&&')?;

```

Listing 27: In-line session rules

```

1 section: WORD '{'sectionContents'}' note?;

```

Listing 28: Section rule

```

1 sectionContents
2 : workloads
3 | (workloads '&&')? structure ('&&' workloads)?
4 ;
5
6 structure: NUM '*' '{' workloads '}';
7
8 workloads: workload ('&&' workload)*;

```

Listing 29: Section content rules

```

1 file: (importStatement)* EOF;
2
3 importStatement: IMPORT;
4
5 IMPORT: 'import ' ([a-zA-Z0-9_]+ | '"' (ESC|.)*? '"') '.';
6 fragment ESC : '\\"' | '\\\\' ;
7
8 OTHER: . -> skip;

```

Listing 30: Full import parse grammar file

### A.3 TypeScript Examples

```
1 class DayData {
2     Sport: ValidSport = null;
3     Sections: Array<Section>; //this is where the workloads are stored.
4     By default only one section
5     Notes: string = null;
6 }
7
```

Listing 31: A TypeScript data class created to store day data

```
1 enum WLType {
2     LessThan = 'lt',
3     GreaterThan = 'gt',
4     Between = 'bt',
5     At = 'at',
6     None = 'none'
7 }
```

Listing 32: A TypeScript enum used to match the type of intensity description in a workload

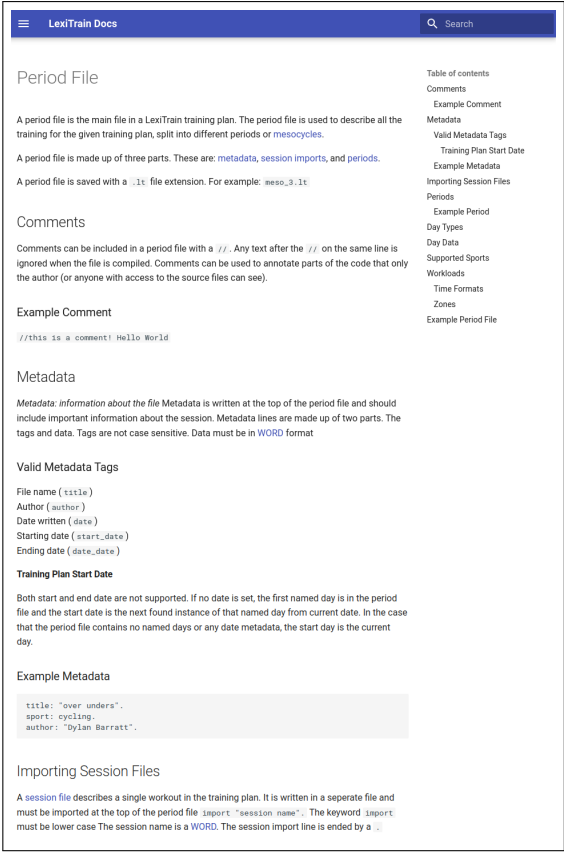
```
1 private metadata: PeriodMetadata = new PeriodMetadata;
2 exitMetaData(ctx: any) {
3     this.metadata[capitalizeFirstLetter(ctx.children[0].getText())] =
4         removeSpeechMarks(ctx.children[2].getText());
5
6     switch (capitalizeFirstLetter(ctx.children[0].getText())) {
7         case "Title":
8             this.metadata.Title = removeSpeechMarks(ctx.children[2].getText());
9             break;
10        case "Author":
11            this.metadata.Author = removeSpeechMarks(ctx.children[2].getText());
12            break;
13        case "Date":
14            this.metadata.Date = stringToDate(ctx.children[2].getText());
15            break;
16        case "Start_date":
17            this.metadata.Start_Date = stringToDate(ctx.children[2].getText());
18            break;
19        case "End_date":
20            this.metadata.End_Date = stringToDate(ctx.children[2].getText());
21            break;
22    }
23
24    if (this.metadata.Title == null) {
25        throw new Error("Title required in session file");
26    }
27 }
```

26 }

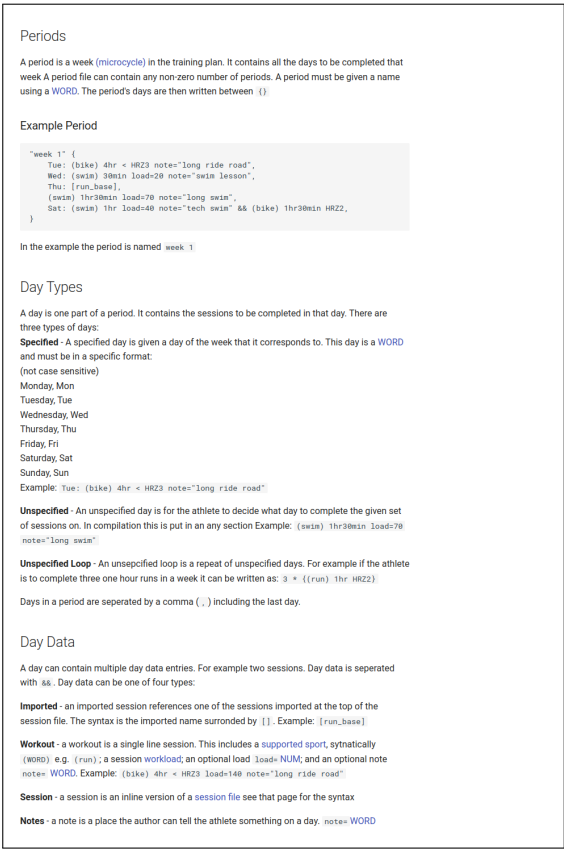
Listing 33: ANTLR4 listener enter and exit functions implemented for the *metaData* rule

```
1 enum IntensityZone {  
2     HRZ1, HRZ2, HRZ3, HRZ4, HRZ5, //matching the Norwegian Olympic model  
3     LTZ1, LTZ2, LTZ3, //matching Dr Seileir's model  
4     PWZ1, PWZ2, PWZ3, PWZ4, PWZ5, PWZ6,PWZ7 //matching Dr Coggan's model  
5 }
```

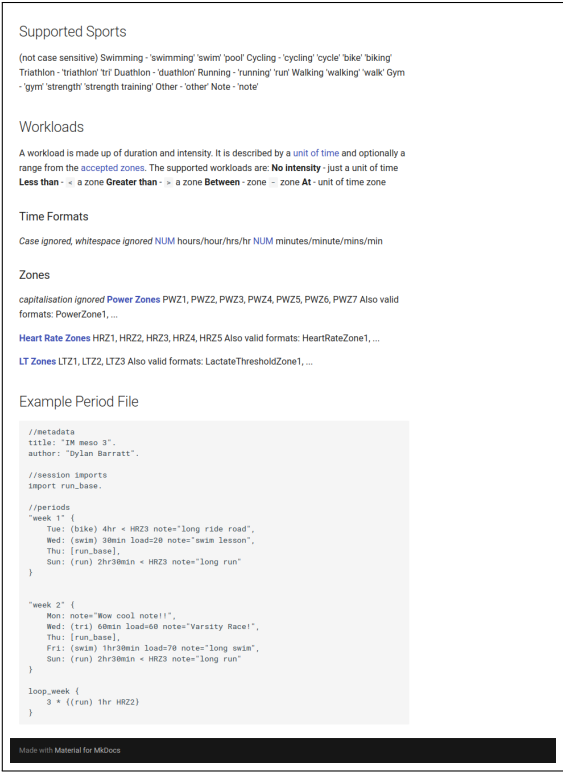
Listing 34: Intensity enum



(a) part one



(b) part two



(c) part three

Figure 10: MkDocs generated web page for period file page markdown file (appendix listing 12) (one page split into three images)

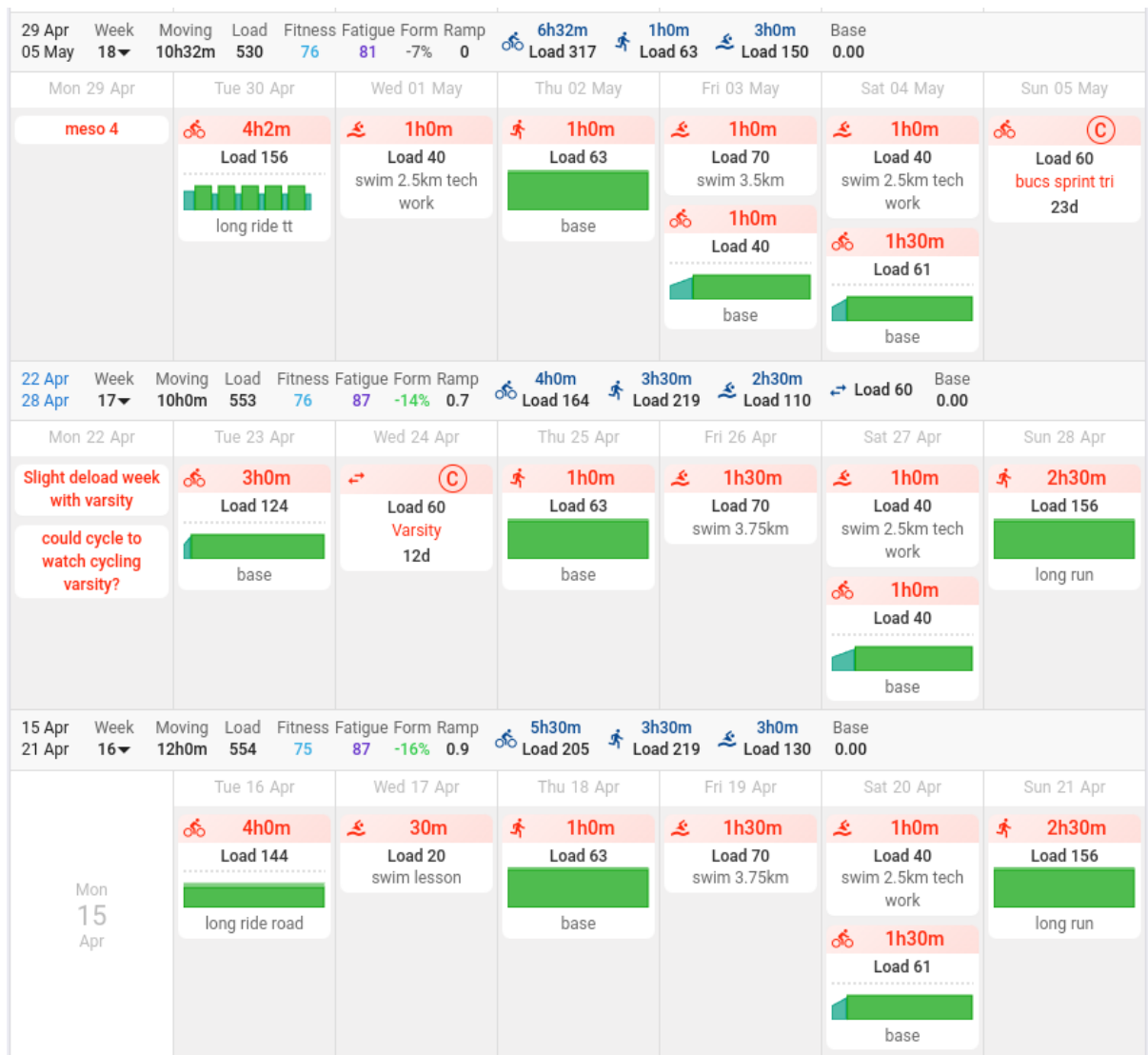


Figure 11: Mesocycle three written in Intervals.ICU

## A.4 Test Results

### A.4.1 Test One Results

```
[@0,0:4='title',<WORD>,1:0]
[@1,5:5=':',<':'>,1:5]
[@2,7:17='"Test File"',<WORD>,1:7]
[@3,18:18='.',< '.'>,1:18]
[@4,20:25='author',<WORD>,2:0]
[@5,26:26=':',< ':'>,2:6]
[@6,28:42='"Dylan Barratt"',<WORD>,2:8]
[@7,43:43='.',< '.'>,2:23]
[@8,45:48='date',<WORD>,3:0]
[@9,49:49=':',< ':'>,3:4]
[@10,51:60='15/04/2024',<WORD>,3:6]
[@11,61:61='.',< '.'>,3:16]
[@12,63:72='start_date',<WORD>,4:0]
[@13,73:73=':',< ':'>,4:10]
[@14,75:84='15/04/2024',<WORD>,4:12]
[@15,85:85='.',< '.'>,4:22]
[@16,128:127='<EOF>',<EOF>,5:0]
```

(a) Test one tokens



(b) Test one parse tree

Figure 12: Test one parse results

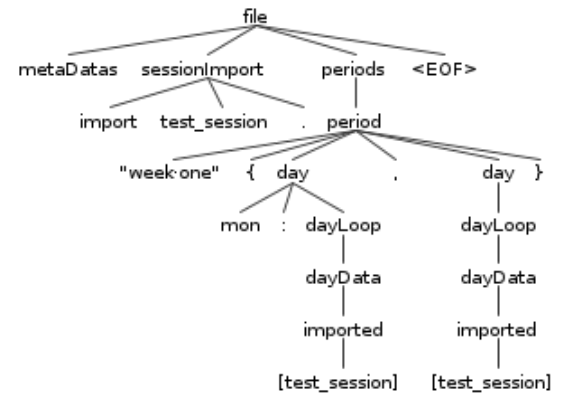
```
IdeRoute.svelte:42
PeriodMetadata {Title: 'Test Fil
e', Author: 'Dylan Barratt', Dat
e: Mon Apr 15 2024 00:00:00 GMT+0
100 (British Summer Time), Start_
Date: Mon Apr 15 2024 00:00:00 GM
T+0100 (British Summer Time), End
_Date: null, ...}
> |
```

Figure 13: Test one data object in a web browser (where the compilation is run)

### A.4.2 Test Two Results

```
[@0,0:5='import',<'import'>,1:0]
[@1,7:18='test_session',<WORD>,1:7]
[@2,19:19='.',< '.'>,1:19]
[@3,22:31='"week one"',<WORD>,3:0]
[@4,33:33='{',< '{'>,3:11]
[@5,39:41='mon',<WORD>,4:4]
[@6,42:42=':',< ':'>,4:7]
[@7,44:57='[test_session]',<IMPORTED>,4:9]
[@8,58:58='.',< '.'>,4:23]
[@9,64:77='[test_session]',<IMPORTED>,5:4]
[@10,79:79='}',< '}'>,6:0]
[@11,81:80='<EOF>',<EOF>,7:0]
```

(a) Test two period tokens



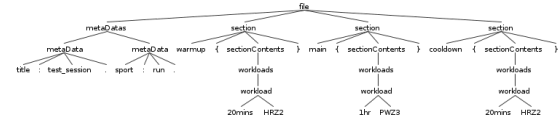
(b) Test two period parse tree

```

[0,0:4='title',<WORD>,1:0]
[1,5:5=':',<':'>,1:5]
[2,7:18='test_session',<WORD>,1:7]
[3,19:19='.',< '.'>,1:19]
[4,21:25='sport',<WORD>,2:0]
[5,26:26=':',< ':'>,2:5]
[6,28:30='run',<WORD>,2:7]
[7,31:31='.',< '.'>,2:10]
[8,34:39='warmup',<WORD>,4:0]
[9,41:41='{',< '{ '>,4:7]
[10,47:52='20mins',<WORD>,5:4]
[11,54:57='HRZ2',<WORD>,5:11]
[12,59:59='}',< '}'>,6:0]
[13,62:65='main',<WORD>,8:0]
[14,67:67='{',< '{ '>,8:5]
[15,73:75='1hr',<WORD>,9:4]
[16,77:80='PWZ3',<WORD>,9:8]
[17,82:82='}',< '}'>,10:0]
[18,85:92='cooldown',<WORD>,12:0]
[19,94:94='{',< '{ '>,12:9]
[20,100:105='20mins',<WORD>,13:4]
[21,107:110='HRZ2',<WORD>,13:11]
[22,112:112='}',< '}'>,14:0]
[23,113:112='<EOF>',<EOF>,14:1]

```

(a) Test two session tokens



(b) Test two session parse tree

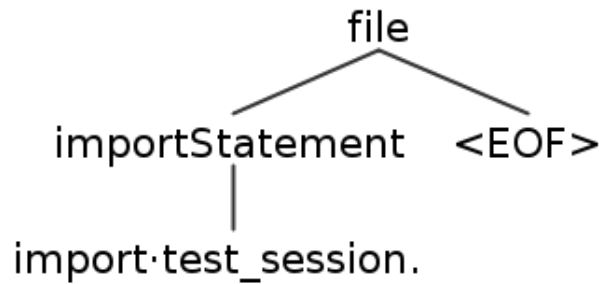


Figure 16: Test two import parse tree

#### A.4.3 Test Three Results

```
[@0,0:4='title',<WORD>,1:0]
[@1,5:5=':',<':'>,1:5]
[@2,7:17='"IM meso 3"',<WORD>,1:7]
[@3,18:18='.',< '.'>,1:18]
[@4,20:25='author',<WORD>,2:0]
[@5,26:26=':',< ':'>,2:6]
[@6,28:42='"Dylan Barratt"',<WORD>,2:8]
[@7,43:43='.',< '.'>,2:23]
[@8,46:51='import',<'import'>,4:0]
[@9,53:60='run_base',<WORD>,4:7]
[@10,61:61='.',< '.'>,4:15]
[@11,64:71='"week 1"',<WORD>,6:0]
[@12,73:73='{',<'{'>,6:9]
[@13,76:78='Tue',<WORD>,7:1]
[@14,79:79=':',< ':'>,7:4]
[@15,81:86='(bike)',<SPORT>,7:6]
[@16,88:90='4hr',<WORD>,7:13]
[@17,92:92='<',<'<'>,7:17]
[@18,94:97='HRZ3',<WORD>,7:19]
[@19,99:103='note=',<'note='>,7:24]
[@20,104:119='"long ride road"',<WORD>,7:29]
[@21,120:120=',',< ','>,7:45]
[@22,123:125='Wed',<WORD>,8:1]
[@23,126:126=':',< ':'>,8:4]
[@24,128:133='(swim)',<SPORT>,8:6]
[@25,135:139='30min',<WORD>,8:13]
[@26,141:145='load=',<'load='>,8:19]
[@27,146:147='20',<NUM>,8:24]
[@28,149:153='note=',<'note='>,8:27]
[@29,154:166='"swim lesson"',<WORD>,8:32]
[@30,167:167=',',< ','>,8:45]
[@31,170:172='Thu',<WORD>,9:1]
[@32,173:173=':',< ':'>,9:4]
[@33,175:184='[run_base]',<IMPORTED>,9:6]
[@34,185:185=',',< ','>,9:16]
[@35,188:190='Fri',<WORD>,10:1]
```

Figure 17: Only the first set of tokens (this) and the final set of tokens (figure 18) (up to the EOF) have been provided due to the length of the token list. The final screenshot of tokens shows that no string was not tokenized as it reaches the EOF.



```

[@143,796:798='Thu',<WORD>,29:1]
[@144,799:799=':',<':'>,29:4]
[@145,801:810='[run_base]',<IMPORTED>,29:6]
[@146,811:811=',',<','>,29:16]
[@147,814:816='Fri',<WORD>,30:1]
[@148,817:817=':',<':'>,30:4]
[@149,819:824='(swim)',<SPORT>,30:6]
[@150,826:828='1hr',<WORD>,30:13]
[@151,830:834='load=',<'load='>,30:17]
[@152,835:836='50',<NUM>,30:22]
[@153,838:842='note=',<'note='>,30:25]
[@154,843:853='"long swim"',<WORD>,30:30]
[@155,855:856='&&',<'&&'>,30:42]
[@156,858:863='(bike)',<SPORT>,30:45]
[@157,865:867='1hr',<WORD>,30:52]
[@158,869:872='HRZ2',<WORD>,30:56]
[@159,873:873=',',<','>,30:60]
[@160,876:878='Sat',<WORD>,31:1]
[@161,879:879=':',<':'>,31:4]
[@162,881:886='(swim)',<SPORT>,31:6]
[@163,888:890='1hr',<WORD>,31:13]
[@164,892:896='load=',<'load='>,31:17]
[@165,897:898='40',<NUM>,31:22]
[@166,900:904='note=',<'note='>,31:25]
[@167,905:915='"tech swim"',<WORD>,31:30]
[@168,917:918='&&',<'&&'>,31:42]
[@169,920:925='(bike)',<SPORT>,31:45]
[@170,927:934='1hr30min',<WORD>,31:52]
[@171,936:939='HRZ2',<WORD>,31:61]
[@172,940:940=',',<','>,31:65]
[@173,943:945='Sun',<WORD>,32:1]
[@174,946:946=':',<':'>,32:4]
[@175,948:952='(tri)',<SPORT>,32:6]
[@176,954:958='load=',<'load='>,32:12]
[@177,959:960='60',<NUM>,32:17]
[@178,962:966='note=',<'note='>,32:20]
[@179,967:983='"bucs sprint tri"',<WORD>,32:25]
[@180,985:985='}',<'}'>,33:0]
[@181,987:986='<EOF>',<EOF>,34:0]

```

Figure 18: Test three period tokens part 5 (last set of tokens up to the EOF)

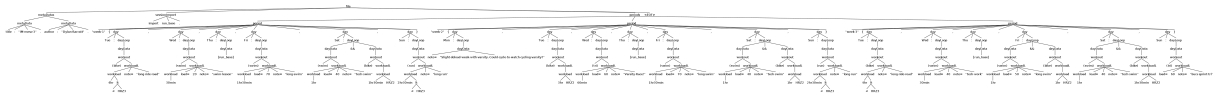


Figure 19: Test three period full parse tree, split version at bottom of appendix for image clarity

```
[@0,0:4='title',<WORD>,1:0]
[@1,5:5=':',<':'>,1:5]
[@2,7:16='"run base"',<WORD>,1:7]
[@3,17:17='.',< '.'>,1:17]
[@4,19:23='sport',<WORD>,2:0]
[@5,24:24=':',< ':'>,2:5]
[@6,26:34='"running"',<WORD>,2:7]
[@7,35:35='.',< '.'>,2:16]
[@8,37:42='author',<WORD>,3:0]
[@9,43:43=':',< ':'>,3:6]
[@10,45:59='"Dylan Barratt"',<WORD>,3:8]
[@11,60:60='.',< '.'>,3:23]
[@12,62:65='load',<WORD>,4:0]
[@13,66:66=':',< ':'>,4:4]
[@14,68:69='50',<NUM>,4:6]
[@15,70:70='.',< '.'>,4:8]
[@16,73:78='warmup',<WORD>,6:0]
[@17,80:80='{',< '{>,6:7]
[@18,83:87='15min',<WORD>,7:1]
[@19,89:92='HRZ1',<WORD>,7:7]
[@20,94:94='}',< '}>,8:0]
[@21,97:100='main',<WORD>,10:0]
[@22,102:102='{',< '{>,10:5]
[@23,105:109='45min',<WORD>,11:1]
[@24,111:111='<',< '<>,11:7]
[@25,113:116='HRZ3',<WORD>,11:9]
[@26,118:118='}',< '}>,12:0]
[@27,122:126='note=',< 'note=>,14:0]
[@28,127:148='"include some strides"',<WORD>,14:5]
[@29,150:171='"cool down w/ strides"',<WORD>,15:0]
[@30,173:173='{',< '{>,15:23]
[@31,176:180='15min',<WORD>,16:1]
[@32,182:182='<',< '<>,16:7]
[@33,184:187='HRZ2',<WORD>,16:9]
[@34,189:189='}',< '}>,17:0]
[@35,191:190='<EOF>',<EOF>,18:0]
```

Figure 20: Test three session tokens

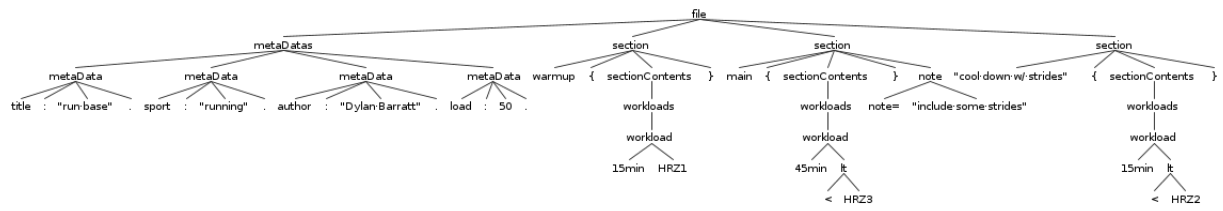


Figure 21: Test three session parse tree

#### A.4.4 Test Four Results

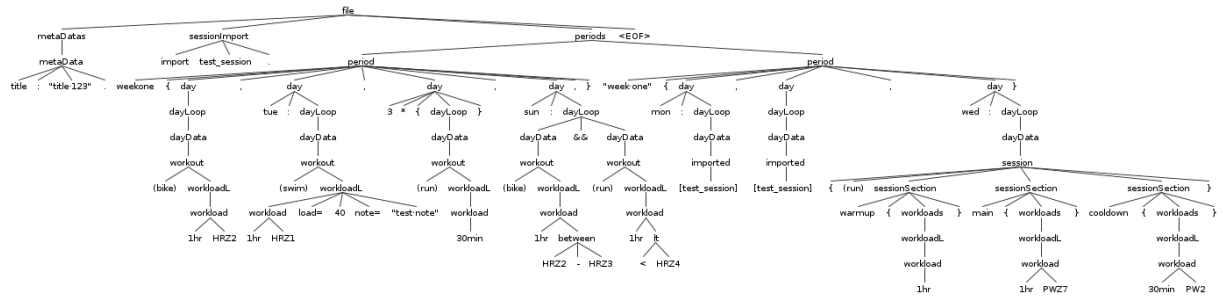


Figure 22: Test four parse tree

```
[@0,0:4='title',<WORD>,1:0]
[@1,5:5=':',<':'>,1:5]
[@2,7:17='title 123"',<WORD>,1:7]
[@3,18:18='.',< '.'>,1:18]
[@4,21:26='import',<'import'>,3:0]
[@5,28:39='test_session',<WORD>,3:7]
[@6,40:40='.',< '.'>,3:19]
[@7,43:49='weekone',<WORD>,5:0]
[@8,51:51='{',< '{'>,5:8]
[@9,57:62='(bike)',<SPORT>,6:4]
[@10,64:66='1hr',<WORD>,6:11]
[@11,68:71='HRZ2',<WORD>,6:15]
[@12,72:72='.',< '.'>,6:19]
[@13,78:80='tue',<WORD>,7:4]
[@14,81:81=':',< ':'>,7:7]
[@15,83:88='(swim)',<SPORT>,7:9]
[@16,90:92='1hr',<WORD>,7:16]
[@17,94:97='HRZ1',<WORD>,7:20]
[@18,99:103='load=',<'load='>,7:25]
[@19,104:105='40',<NUM>,7:30]
[@20,107:111='note=',<'note='>,7:33]
[@21,112:122='"test note"',<WORD>,7:38]
[@22,123:123='.',< '.'>,7:49]
[@23,129:129='3',<NUM>,8:4]
[@24,131:131='*',< '*'>,8:6]
[@25,133:133='{',< '{'>,8:8]
[@26,134:138='(run)',<SPORT>,8:9]
[@27,140:144='30min',<WORD>,8:15]
[@28,145:145='}',< '}'>,8:20]
[@29,146:146='.',< '.'>,8:21]
[@30,152:154='sun',<WORD>,9:4]
[@31,155:155=':',< ':'>,9:7]
[@32,157:162='(bike)',<SPORT>,9:9]
[@33,164:166='1hr',<WORD>,9:16]
[@34,168:171='HRZ2',<WORD>,9:20]
[@35,173:173='- ',< '- '>,9:25]
[@36,175:178='HRZ3',<WORD>,9:27]
[@37,180:181='&&',< '&&'>,9:32]
```

(a) Test four tokens part 1

```
[@38,183:187='(run)',<SPORT>,9:35]
[@39,189:191='1hr',<WORD>,9:41]
[@40,193:193='<',< '<'>,9:45]
[@41,195:198='HRZ4',<WORD>,9:47]
[@42,199:199='.',< '.'>,9:51]
[@43,201:201='}',< '}'>,10:0]
[@44,204:213='"week one"',<WORD>,12:0]
[@45,215:215='{',< '{'>,12:11]
[@46,221:223='mon',<WORD>,13:4]
[@47,224:224=':',< ':'>,13:7]
[@48,226:239='[test_session]',<IMPORTED>,13:9]
[@49,240:240='.',< '.'>,13:23]
[@50,246:259='[test_session]',<IMPORTED>,14:4]
[@51,260:260='.',< '.'>,14:18]
[@52,266:268='wed',<WORD>,15:4]
[@53,269:269=':',< ':'>,15:7]
[@54,271:271='{',< '{'>,15:9]
[@55,272:276='(run)',<SPORT>,15:10]
[@56,278:283='warmup',<WORD>,15:16]
[@57,285:285='{',< '{'>,15:23]
[@58,286:288='1hr',<WORD>,15:24]
[@59,289:289='}',< '}'>,15:27]
[@60,292:295='main',<WORD>,15:30]
[@61,297:297='{',< '{'>,15:35]
[@62,298:300='1hr',<WORD>,15:36]
[@63,302:305='PWZ7',<WORD>,15:40]
[@64,306:306='}',< '}'>,15:44]
[@65,309:316='cooldown',<WORD>,15:47]
[@66,318:318='{',< '{'>,15:56]
[@67,319:323='30min',<WORD>,15:57]
[@68,325:327='PW2',<WORD>,15:63]
[@69,328:328='}',< '}'>,15:66]
[@70,329:329='}',< '}'>,15:67]
[@71,331:331='.',< '.'>,16:0]
[@72,332:331='<EOF>',<EOF>,16:1]
```

(b) Test four tokens part 2

```
PASS src/lib/HelperFunction.test.ts
Testing all helper functions
  dayNameToIndex function
    ✓ Correct days (2 ms)
    ✓ Incorrect day
  daysToWeek function
    ✓ should return a full week array when given an array of days with no missing days
    ✓ should return a full week array with missing days filled as null (1 ms)
    ✓ should handle extra days beyond Sunday by adding them to the end of the array (1 ms)
  flattenPeriods function
    ✓ should return correct flattened periods when only metadata is specified
    ✓ should return correct flattened periods when a period is given with a session
  stringToZone function
    ✓ should return the correct intensity zone for valid input strings
    ✓ should throw an error for invalid input strings (16 ms)
  validTimeString function
    ✓ should return true for valid time strings
    ✓ should return false for invalid time strings (1 ms)
  DayDataToString function
    ✓ should return an empty string when input array is empty
    ✓ should correctly convert DayData objects into string representation (1 ms)
  shortenString function
    ✓ should return the original string if its length is within the maximum length
    ✓ should shorten strings that exceed the maximum length
    ✓ should handle maximum length being equal to or less than 3

Test Suites: 1 passed, 1 total
Tests:       16 passed, 16 total
Snapshots:   0 total
Time:        1.485 s, estimated 2 s
Ran all test suites.
```

Figure 24: TS-Jest test results

## A.4.5 Test Five Results

## A.4.6 Test Six Results

IM meso 3  
Written by: Dylan Barratt

April 2024

< > Current Month

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Any
01	02	03	04	05	06	07	none
08	09	10	11	12	13	14	none
15	16	17	18	19	20	21	none
22	23 🚴	24 🏊	25 🏊	26 🏊	27 🏊 🚴	28 🏊	none
29 ■	30 🚴	01 🏊 🚴	02 🏊	03 🏊	04 🏊 🚴	05 🏊	none

Figure 25: First month of the compiled training plan calendar (note the displayed version is a shrunk version of the final webpage)

IM meso 3  
Written by: Dylan Barratt

May 2024

< > Current Month

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Any
29 ■	30 🚴	01 🏊 🚴	02 🏊	03 🏊	04 🏊 🚴	05 🏊	none
06	07 🚴	08 🏊	09 🏊	10 🏊 🚴	11 🏊 🚴	12 🏊 🚴	none
13	14	15	16	17	18	19	none
20	21	22	23	24	25	26	none
27	28	29	30	31	01	02	none

Figure 26: Second month of the compiled training plan calendar (note the displayed version is a shrunk version of the final webpage)

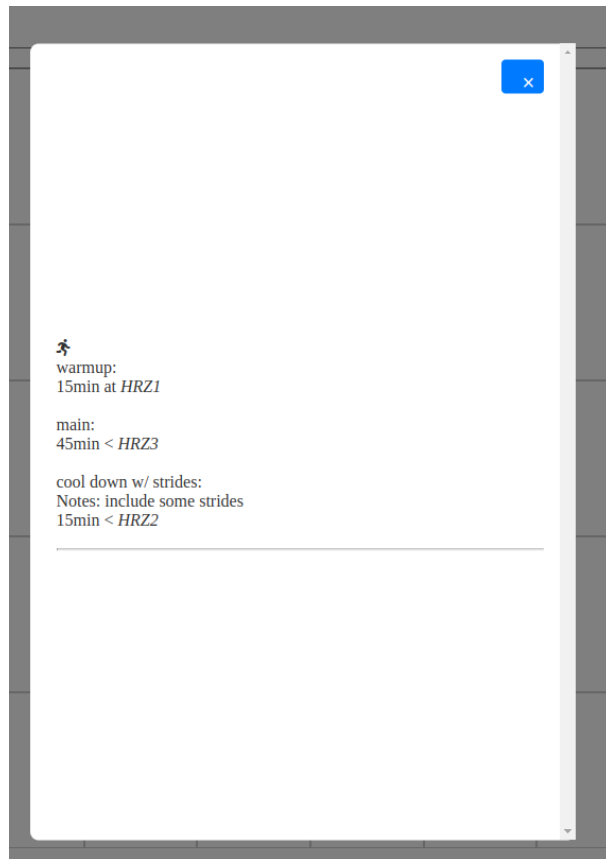


Figure 27: The details of a single day in the calendar

## A.4.7 Test Seven Results

# IM meso 3

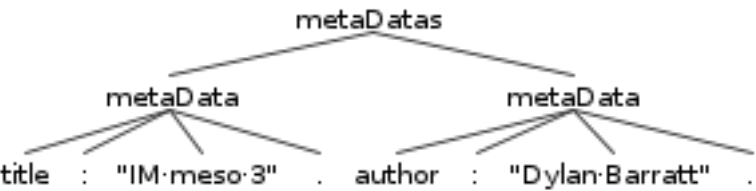
By: Dylan Barratt

mon	tue	wed	thu	fri	sat	sun	any
	Cycle: "long ride road" 4hr<HRZ3	Swim: load 20 "swim lesson" 30min	Run: warmup: 15min HRZ1  main: 45min<HRZ 3  cool down w/ strides: "include some strides" 15min<HRZ 2	Swim: load 70 "long swim" 1hr30min	Swim: load 40 "tech swim" 1hr  Cycle: 1hr30min HRZ2	Run: "long run" 2hr30min<H RZ3	
Notes: "Slight deload week with varsity. Could cycle to watch cycling varsity?"	Cycle: 3hr HRZ2	Tri: load 60 "Varsity Race!" 60min	Run: warmup: 15min HRZ1  main: 45min<HRZ 3  cool down w/ strides: "include some strides" 15min<HRZ 2	Swim: load 70 "long swim" 1hr30min	Swim: load 40 "tech swim" 1hr  Cycle: 1hr HRZ2	Run: "long run" 2hr30min<H RZ3	
	Cycle: "long ride road" 4hr<HRZ3	Swim: load 40 "tech work" 30min	Run: warmup: 15min HRZ1  main: 45min<HRZ 3  cool down w/ strides: "include some strides" 15min<HRZ 2	Swim: load 50 "long swim" 1hr  Cycle: 1hr HRZ2	Swim: load 40 "tech swim" 1hr  Cycle: 1hr30min HRZ2	Tri: load 60 "bucs sprint tri" 1hr30min HRZ2	

Figure 28: The compiled PDF



The following images are parts of the parse tree in figure 19 for test three, displayed separately for image clarity.

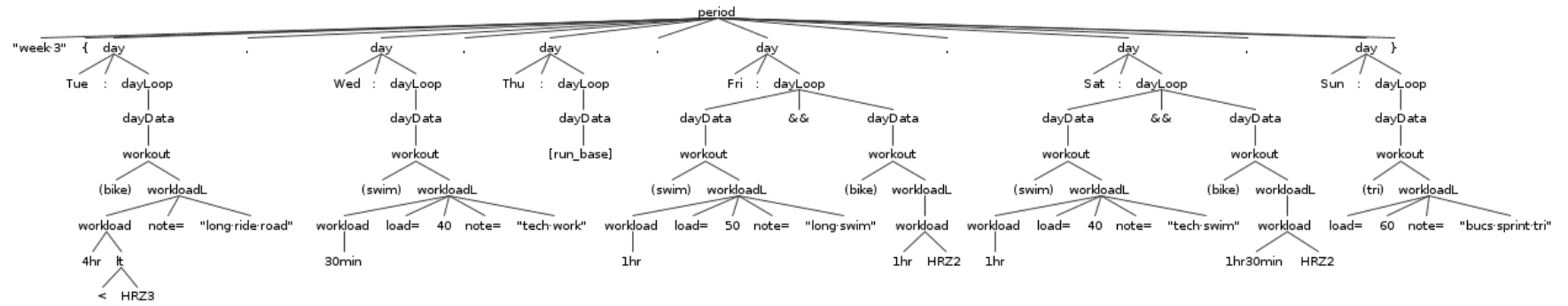


Mesocycle three metadata



Mesocycle three imports





Mesocycle three period three