Masaryk University

Faculty of Informatics

Master's Thesis

Parser Generator in C#

Richard Sirný

2013

Bibliographic Identification

Master's thesis title:	Parser Generator in C#
Author full name:	Richard Sirný
Study program:	Applied Informatics
Study field:	Applied Informatics
Supervisor:	RNDr. David Sehnal
Year of defend:	2013
Keywords:	parser, C#, CFG, DSL, parser generator, Earley parsing

Bibliografická identifikace

Téma diplomové práce:	Generátor parserů v jazyce C#
Jméno autora:	Richard Sirný
Studijní program:	Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Školitel:	RNDr. David Sehnal
Rok obhajoby:	2013
Klíčová slova:	parser, C#, CFG, DSL, generátor parserů, Earleyho parsování

Declaration

Hereby I declare, that this thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in Bibliography section.

Acknowledgement

I would like to thank to my family, friends and teachers. Very special thanks goes to David Sehnal whose support, patience, helpful attitude and leadership helped me greatly when writing this work.

Abstract

The aim of the thesis is to choose a suitable parsing algorithm and create a parser generator framework in C# language with the following features: the input grammar is specified with the C# object instances; minor mistakes in the input can be detected and corrected; suggestions on what can follow next can be generated for a given position in the input. The parser is expected to be used in web-based projects for parsing short queries in domain-specific languages.

The theoretical part of the thesis reviews different parsing techniques with regard to generality, error-handling potential, follow suggestion generation and performance. The technique chosen for the parser generator is automaton-based Earley parsing.

The framework and the used algorithm is described and tested in the practical part of the thesis. MotiveQuery language is used to demonstrate the capabilities of the framework. The error-handling mechanisms are also described there. The obtained results suggest that the framework should perform well with unambiguous grammars. On the other hand, the use with ambiguous grammars is rather inadvisable.

Abstrakt

Cílem této práce je vybrat vhodný algoritmus pro generátor parserů a implementovat jej v jazyce C#. Vstupní gramatika bude zadávána pomocí objektů v jazyce C#. Výsledný parser by měl být schopen zotavit se z drobných chyb na vstupu a umožňovat generování návrhů na další pokračování daného výrazu. Očekávané použití frameworku spočívá v parsování kratších řetězců doménově specifického jazyka ve webovém prostředí.

Teoretická část práce obsahuje přehled použitelných parsovacích technik a jejich ohodnocení s přihlédnutím k jejich obecnosti, potenciálu pro zvládání chyb, generování návrhů na pokračování daného řetězce a výkonu. Technikou, která byla vybrána pro implementaci, je varianta Earlyho parsování.

V praktické části práce je popsána architektura a detaily použitého algoritmu. Součástí této části je také ukázka použití frameworku s jazykem MotiveQuery a popis mechanismů pro zotavení z chyb. Výsledky testů ukazují, že framework je dobře použitelný s jednoznačnými gramatikami, ale použitelnost s nejednoznačnými gramatikami je horší.

Table of Contents

1 Introduction

A *domain-specific language* (DSL) is a language that is used for solving problems of some particular kind. For example, a database can be queried for data by SQL language which was designed specifically for this task. The opposite of DSL is a general-purpose language like C/C++, Java, C#, etc. that may be used, as the name suggests, to deal with virtually any kind of task. While it may be possible to solve given problem with a general-purpose language, the advantage of DSL is that it may be more user-friendly and the way the problem is solved is more apparent.

One way to create an implementation of DSL is to define it in a formalism called *grammar*. Then a special tool – a *parser generator* – is used to create a *parser* which is software that, apart from other things, checks whether a text is written in given DSL. If the text is written in DSL then the parser produces a data structure – a *parse tree* – that represents the structure of the text. This structure is then interpreted by the computer as specified by the creator of the DSL to carry out the action specified by the text. Additionally, if the input text has some errors in it, the parser may be used to signal those errors to the user.

With the advent of the Internet and cloud computing, new ways to use domain-specific languages emerged. For example, the language could be used for querying information from some database or it might manipulate some data stored in the cloud. Whatever be the case, the users would undoubtedly appreciate if they were given some help as they enter the query or command. Such help could be the aforementioned highlighting of syntax errors in their text or the ability to prompt for suggestion on what syntactical constructs may follow after given position (also know as code completion). It is also important to note that if the language is used for querying or data manipulation over the web, its parser will not likely have to deal with the input of great length (at least compared to general-purpose programming languages parsers).

The requirements of a parser discussed so far were, on the whole, those of a DSL user. Naturally, the creators of DSL require additional features. One such feature is how easily the grammar can be changed. Another one may be what they have to learn in order use the parser generator as most of the tools use their own meta-language to define the grammar.

With most of the important concepts outlined, it is now the time to look on the purpose of this thesis. As mentioned earlier, most parser generators require their users to define the grammar in a meta-language saved in format separate from the one used for the source files. While this approach enables the use across different languages and systems, it also requires the user to abandon the familiar environment of their programming language and

integrated development environment. Hence, one goal of this thesis si to create a parser generator which enables the grammar to be defined via C# language constructs. As hinted by the section that dealt with the use of DSL in the context of the Internet, the parser should offer support for error detection and suggestions on what can follow after given position. Moreover, if the error is a minor one, the parser should be able to correct it and carry on with the parsing. Additionally, it should be usable in the web-based applications. The use of the parser is to be demonstrated on MotiveQuery language created by David Sehnal which is used for description and analysis of protein motives.

A brief introduction to the theory of parsing is provided in Chapter 2 . Subsequently, various parsing techniques are reviewed in Chapter 3 ; the choice of the technique for the parser generator is also discussed there. Chapter 4 deals with the details of the chosen algorithm and the design of the parser generator. The performance of our solution is measured and discussed in Chapter 5 . Finally, the work is summarised in Chapter 6 .

2 Theory

2.1 Role of the parser

A compiler is a program that enables translation of human-readable programming language into a representation that is executable on computers. The syntax of a language is formally described by its grammar which may also impose a certain structure on its sentences which is then used in the compilation process. Probably the best way to describe how the compiler works is to think of it as of a series of consecutive phases [1].

Firstly, a text written in a given language is transformed into a sequence of tokens. The purpose of a token is to categorise different strings in the input text. For example, a string of digits could be a token that represents a number. Some characters that are not necessary for the future phases are left out. Such characters are spaces for example. This phase is called *lexical analysis*. Sometimes is is also refered to as *scanning*. The part of compiler that is responsible for this phase is then called a *lexer* or a *scanner*.

The second phase is called *syntactic analysis*, but it is also known as *parsing*. During this phase the sequence of tokens is further structured into a tree data structure called *parse tree* based on the grammar of the programming language. Before being used in the consequent phases, the parse tree is sometimes transformed into a more convenient representation: an *abstract syntax tree* (AST).

There may be consistency requirements on the entered text which are not verifiable by constructing a parse tree. Therefore, they may have to be dealt with in the next phase which is called *type checking*.

Since the rest of the process is not very pertinent to the topic of the thesis, we will not discuss the following phases in here. An interested reader may found plenty of information in [1, pp. 2-4], [2, pp. 40-42].

It is also important to note that this is not the only way to describe parsing. For example, Grune and Jacobs define [3, p. 1] it as *"the process of structuring a linear representation in accordance with a given grammar."* The linear representation can then be *"a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behaviour, in short any linear sequence in which the preceding elements in some way restrict the next element."* Hence, the importance of parsing is not restricted to the compiler construction only.

In order to be able to discuss different parsing techniques, it is necessary to introduce several concepts from formal language theory (FLT).

2.2 Language and Alphabet

An *alphabet* is [3, pp. 5-7] a finite set of characters which can be also called symbols, letters etc. It serves as the basis for defining sequences of characters called *words*. An example of an alphabet is a set $\{a, b\}$; words composed of this alphabet may then be a, aa, ab, bbb, etc. There is also a special word ε that contains no characters; it has zero length.

One way to think about languages is to consider them to be sets of admissible sentences [3, p. 5]. For example, a set of all possible English sentences could be considered the English language. Actually, this is precisely the way a language is regarded in FLT. The terminology differs slightly though: the words described in previous paragraph are the sentences. Formally, a *language* is a set of *words* over some *alphabet*. For example, with alphabet $\Sigma=\{a\}$ it is possible to consider a set $S=\{a, aa, aaa,...\}$ a language. This example also shows that languages can be infinite.

2.3 Grammar

Obviously, languages can be infinite; hence, it would be convenient to have some finite representation that could describe them. A grammar can be used for such purpose. It does so by providing means for generating every possible word of a language. Formally, a *grammar* is defined [3, p. 14] as a 4-tuple (N, Σ, P, S) where:

- N is a finite set of *non-terminal symbols* (non-terminals).
- Σ is a finite set of *terminal symbols* (terminals) and $N \cap \Sigma = \emptyset$, $V = N \cup \Sigma$.
- P is a finite set of *production rules* (productions) (α,β) where α is a word over V with at least one non-terminal and β is a (possibly empty) word over V. A production is usually written as $\alpha\rightarrow\beta$. It is possible to further shorten the notation by using only single production for productions with the same left-hand side. The right-hand side of this shortened production contains all the right-hand sides of the productions separated by symbol |. For example, if there are two productions for α with right-hand sides β and γ, the shortened production looks like $\alpha\rightarrow \beta \mid \gamma$.
- S is special non-terminal called *start symbol* or *root*.

2.3.1 Regular grammar

A grammar is said to be regular if it allows only productions in form $A\rightarrow aB$ or $A\rightarrow B$ where A, B are non-terminals and a is a terminal. This kind of grammar is indirectly used in lexical analysis: the meaningful groups of characters can be easily described by *regular expressions* which are just a convenient representation of regular grammars.

2.3.2 Context-free grammar

A grammar with productions whose left-hand side consists of a single non-terminal is called *context-free grammar* (CFG). The languages that can be described by a CFG are called *context-free languages* (CFL). There are also other classes of languages, however, they need not be discussed as they are not pertinent to the topic. Suffice to say, if more relaxed form of productions is allowed, the grammar can be used to describe greater class of languages. It should also be noted that there are some languages that cannot be described by any grammar [3, p. 8].

An example of a simple CFG grammar is shown in Figure 1. The terminals of this grammar are symbols +,- and **n**. In order to simplify the grammar, a number is represented by terminal **n**. The non-terminals are symbols *Exp* and *S* which is a start non-terminal.

$$\begin{aligned} S &\rightarrow Exp \\ Exp &\rightarrow Exp+Exp \mid Exp\text{-}Exp \mid \mathbf{n} \end{aligned}$$

Figure 1: Simple grammar for arithmetic expression with addition and subtraction operations with terminal **n** *representing a number.*

2.3.3 Conventions

There are certain conventions for denoting grammar symbols and their sequences. The following applies if not specified otherwise:

- Lower-case letters α, β, γ,... from the beginning of Greek alphabet mark a sequence of terminals and non-terminals. The sequence may also be empty or contain only one symbol.
- Upper-case letters *A, B, C*,... from the beginning of the alphabet represent a non-terminal.
- Lower-case letters *a, b, c,*... from the beginning of the alphabet denote a terminal.
- *S* is a start non-terminal.

There is also a special name for productions with empty right-hand side: they are called *epsilon productions* or *epsilon rules*, sometimes the name may be shortened to ε-productions.

2.3.4 Derivation

As was mentioned above, the production rules allow generating words that belong to the language defined by the grammar. This process is known as *derivation* [1, p. 60]. We shall restrict the description to the productions of regular and context-free grammars. One derivation step is a replacement of a non-terminal in a sequence of grammar symbols by a

right-hand side of appropriate production. The symbol used for denoting a derivation step is ⇒. For example, if there is a production $A \rightarrow \gamma$, a possible derivation step for sequence $\alpha A \beta$ is $\alpha A \beta \Rightarrow \alpha \gamma \beta$. A sequence of non-terminals and terminals derived from the start non-terminal is called a *sentential form*. If there are no non-terminals in the sentential form, it is called a *sentence* and it belongs to the language generated by the grammar. Two possible derivations of a sentence **n**+**n**-**n** of simple arithmetic expressions grammar are shown in Figure 2. The notation ⇒* means that the sentential form on the right-hand side was derived from the left-hand side in zero or more derivation steps. Furthermore, derivation in one or more steps is denoted by symbol $\Rightarrow^{+}$.

1) $S \Rightarrow Exp \Rightarrow Exp+Exp \Rightarrow Exp+Exp-Exp \Rightarrow \mathbf{n}+Exp-Exp \Rightarrow \mathbf{n}+\mathbf{n}-Exp \Rightarrow \mathbf{n}+\mathbf{n}-\mathbf{n}$
2) $S \Rightarrow Exp \Rightarrow Exp\ -Exp \Rightarrow Exp+Exp-Exp \Rightarrow \mathbf{n}+Exp-Exp \Rightarrow \mathbf{n}+\mathbf{n}-Exp \Rightarrow \mathbf{n}+\mathbf{n}-\mathbf{n}$

Figure 2: Two possible derivations of a sentence n+n-n.

There are two special cases of derivation. If the sentence is obtained by always deriving from the leftmost non-terminal in the sentential form, the derivation is said to be *leftmost*. Conversely, if a rightmost non-terminal is always used throughout derivation, it is then called a *rightmost derivation*.

If a grammar contains non-terminal A such that $A \Rightarrow^{+} A\alpha$, it is said to be *left-recursive*. Furthermore, if the recursion is a result of a presence of a production $A \rightarrow A\alpha$, it is the case of *direct left-recursion*. On the other hand, if the recurrence of a non-terminal A throughout the derivation is caused by other factors, it is referred to as *hidden left-recursion*.

If there is a non-terminal A such that $A \Rightarrow^{+} \alpha A$, it is called *right-recursive*. The concepts of *hidden right-recursion* and *direct right-recursion* are defined similarly to their counter-parts from the left-recursive case.

2.3.5 Syntax Tree

Derivation can be represented graphically by a tree structure that is called a *syntax tree* or a *parse tree*. The inner nodes of the tree are labelled with non-terminals while the leaves of the tree are labelled with terminals or empty word symbol. The tree is constructed [1, pp. 61-64] from the derivation of given sentence as follows: *"The root of the tree is the start symbol of the grammar, and whenever we rewrite a nonterminal we add as its children the symbols on the right-hand side of the production that was used."* The advantage [1, p. 64] of using the tree for representing a derivation is that it is independent of the order in which the non-terminals are rewritten. What only matters are the rules used

for the derivation. Two parse trees corresponding to the derivations from Figure 2 are shown in Figure 3.

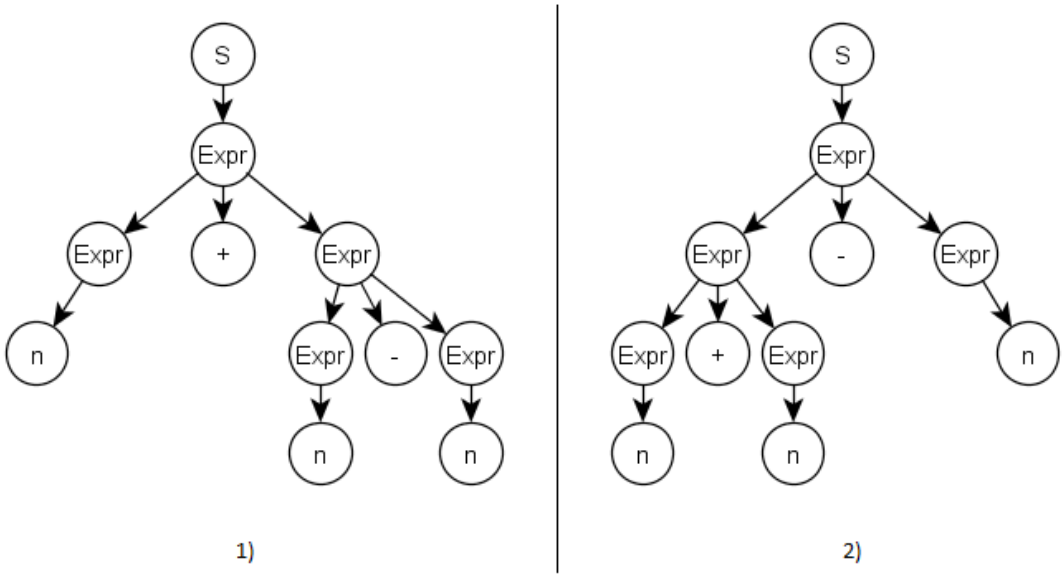

Figure 3: Two possible parse trees for expression n+n-n

The significance of a syntax tree lies in the fact that it imposes a structure on the sentence. For example, a syntax tree of an arithmetic expression could be used to evaluate it: a node with a plus sign child node could ask its two other children for their value and return the sum.

If the parse tree is to be used for evaluation or in some similar process, it may be beneficial to reduce the amount of unnecessary information. This is achieved by transforming the parse tree into an *abstract syntax tree* (AST). A result of such transformation can be seen in Figure 4.

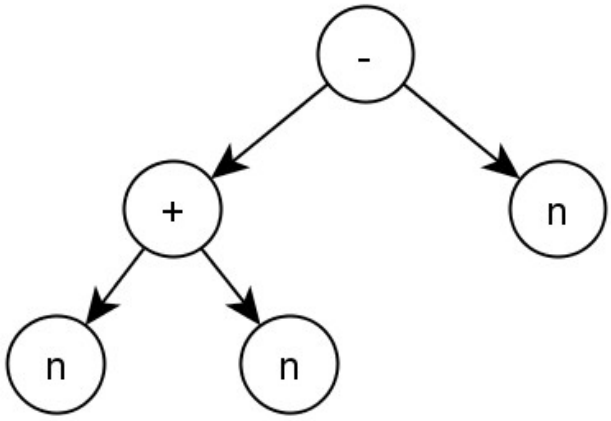

Figure 4: Abstract syntax tree for expression n+n-n

2.3.6 Ambiguity

As is apparent from Figure 3, there may be multiple syntax trees for a given sentence. If a grammar generates a sentence that has more than one parse tree, it is said to be *ambiguous*. Ambiguity in a grammar might be an issue if the parse tree is to be used for some sort of evaluation as it could produce different result than intended. For example, if we add rules *Exp→Exp*Exp* and *Exp→Exp/Exp* to the grammar from Figure 1, the arithmetic expression *2+3*4* would have two different trees with evaluation results corresponding to *(2+3)*4* and *2+(3*4)*. Obviously, the evaluation of the former yields different result than expected from arithmetic expressions.

2.4 Parsing

Context-free grammars are quite useful when it comes to generating sentences that belong to their language. However, is it also possible to determine whether a given sentence belongs to the language generated by given grammar? In theory, it might be possible to generate all sentences of the same length as the sentence in question and then check if the sentence is present. Unfortunately, this approach would be quite costly in terms of computational time and storage space. Luckily, there are more effective procedures to address the issue. All procedures that solve the problem in question are called *recognisers*.

Being able to determine whether a sentence is part of a language is quite useful, however, for some tasks it is also necessary to know how the sentence was derived – in other words its syntax tree is needed. The process of creating the tree from a sentence is

known [1, p. 55] as *parsing*. A *parser* is then a piece of software that parses sentences of some fixed grammar. Another kind of software is a *parser generator* which can create a parser for given grammar.

In FLT, the basic theoretical methods used for parsing or recognising sentences employ a device called *push-down automaton* [4, p. 109]. There are two basic ways to build a tree: either the method starts from the start non-terminal constructing the tree from the root to the leaves or subtrees are built from the leaves and joined together as the sentence is read. Unfortunately, push-down automata are non-deterministic in nature which leads to possibly high computational time complexity. As a result, these methods are not widely used in the real-world parsers [3].

Even though the pure push-down automata are not really used beyond the scope of FLT, their basic approach to building the tree is often employed as a way of categorising the parsing techniques [1–3]. Hence, the following section discusses both approaches – top-down and bottom-up.

2.4.1 Top-down parsing

One way to construct the tree is to start from the top, i.e. the start non-terminal, and guess what productions can be used based on the sentence [2, pp. 217-218], [3, p. 66]. As the tree is created, the input is read from left to right. This approach can be also viewed [2, p. 217] as *"finding a leftmost derivation for the input string"*. Figure 6 shows the order in which a parse tree for expression *n+n*n* can be constructed. The related grammar can be found in Figure 5.

$$
\begin{aligned}
S &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \varepsilon \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \varepsilon \\
F &\rightarrow (S) \mid \mathbf{n}
\end{aligned}
$$

Figure 5: Grammar for arithmetic expression with addition and multiplication operations with terminal **n** *representing a number. (source : [2, p. 217])*

The issue with this technique is that it does not handle [2, p. 212] the left-recursive grammars well. In fact, if the grammar is left-recursive, the parsing may never finish. The solution to the problem is to transform the grammar to an equivalent one without the left-

recursive productions. The algorithm can be found in [2, p. 212]. The trouble with this equivalent grammar is that it may not be as intuitive as the original – see Figure 5 for example. Moreover, if a certain way of evaluating or interpreting the parse tree is defined for the original grammar, it will not probably work with the new one.

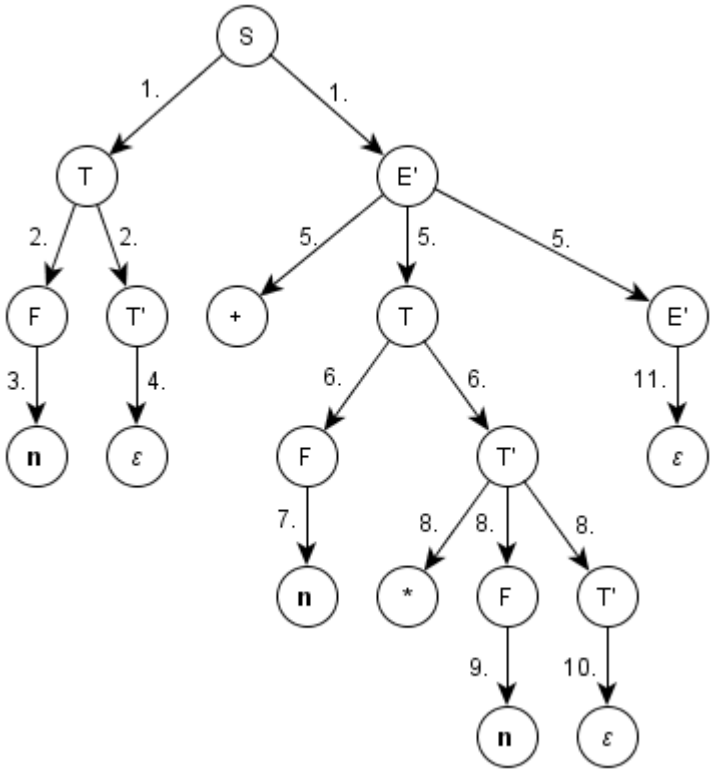

*Figure 6: A parse tree for expression n+n*n. The numbers represent in which derivation step the nodes were added.*

2.4.2 Bottom-up parsing

Unlike the top-down approach, bottom-up parsing starts building the tree from the leaves and works it's way up the tree until all the input is consumed and a tree with start non-terminal in the root is built [2, pp. 233-235]. Another way of looking at this process is to view it as a procedure that rewrites the input string back to the start non-terminal. This is done by replacing a part of the input with a left-hand side of a production whose right-hand side match the beginning of the input. As a result, the bottom-up parsing effectively reconstructs the right-most derivation. The process of replacing appropriate right-hand side

of a production, sometimes called *handle*, with the left-hand side in the input is known as a *reduction step*. An example of this process may be found in Figure 7.

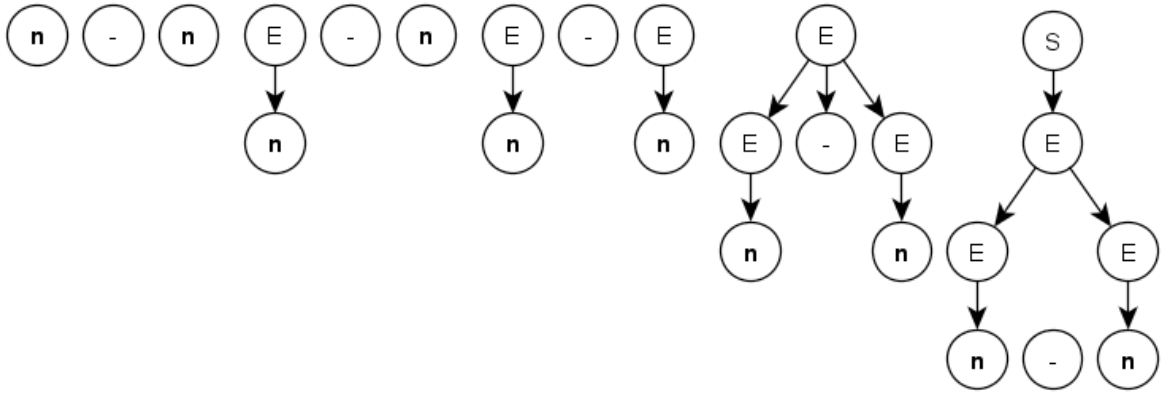

Figure 7: Reduction of expression n-n to start non-terminal. The current sentential form is always on the top.

3 Parsing Techniques

Before discussing the parsing techniques, it is necessary to determine what features they can have. These features can be in turn used as criteria for assessing the methods.

3.1 Preliminaries

While it might be more fitting to discuss the requirements on our parser in the section that deals with its implementation, it is necessary to address the issue, at least partially, now as the criteria are influenced by the desired features of the parser.

3.1.1 Parser Features

The following features are likely to have some impact on the selection of the parsing method:

- **error recovery**: the parser should be able to recover from minor syntactical errors in the input.
- **follow suggestions**: it should be possible to easily determine what token can follow next after given position in the input.
- **short queries**: the parser is expected to parse rather short texts that query for some information or manipulate some data.

There is one last issue to address: traditionally, if the parser is expected to repeatedly parse a text that is being changed between the parses, it is necessary for it to support *incremental parsing* [5]. Basically, an incremental parser retains the parse tree and changes only parts relevant to the changed input. However, web-based services scale better if they are *stateless*, i.e. they do not have to retain information as they receive requests. Moreover, the input is expected to be rather short. Therefore, it was decided that the parser need not be incremental.

3.1.2 Generality

As hinted in Section 2.4, some parsing methods cannot work with all grammars. For example, if the method is based on top-down parsing, it is quite likely that it will not handle well left-recursive grammars. Therefore, one thing to notice about a parsing technique is how general the grammar can be.

3.1.3 Ambiguity

While some languages can be defined by unambiguous grammar, there are some languages that are inherently ambiguous meaning that they can be described only by ambiguous grammars [4, p. 106]. Ambiguity is not prefered in computer languages as in its presence a problem arises: which parse tree should be used for interpretation? On the other hand, it has been shown [6] that an ambiguous grammar can simplify the parsing of C++. To further complicate the matter, it is known [7] that ambiguity problem for context-free grammars is undecidable: the corollary of which is that there is no parsing technique which would work with all unambiguous grammars and no ambiguous grammar. To conclude, the way the technique deals with ambiguity is a feature that ought to be considered.

3.1.4 Error Handling

Each parsing technique has to deal with the possibility that the input is not correct. There are various schemes [3, Chapter 16] that address the issue. The simplest reaction possible is to report an error and stop the parsing. However, this approach is not really usable as one of the required features of our parser is the ability to correct minor errors.

The ability to recover from errors may not be solely determined by the parsing technique, however, it is rational to expect that various techniques have different amenability to error handling. For example, the amount of contextual information during the parse or the complexity of the method may affect how easily an error handling can be incorporated into the technique. The technique might have what is known as *correct prefix property* [3, p. 521] which detects *"an error at the first symbol in the input that results in a prefix that cannot start a sentence of the language."* Another feature that a parser might have is *immediate error detection property* which, as suggested in [3, p. 524], *"means that an error is detected as soon as the erroneous symbol is first examined."*

It is difficult to pinpoint precise features that allow good error recovery. While the aforementioned properties provide good insight, a technique cannot be judged solely by their presence or absence. To conclude, the reason why this criterion is mentioned is rather to acknowledge that a parsing technique has to be investigated from this point of view as well.

3.1.5 Performance

It is natural to ask how fast the parsing technique is or how much resources it needs. Consequently, it has to be one of the criterions to consider. However, it has to be noted that expected use of our parser does not necessitate to regard the performance as the sole concern.

3.1.6 Follow Suggestions

Since our parser is required to suggest what token can follow after given position in the input, the parsing method has to support this feature somehow. It is relatively easy to precompute what tokens can follow after given non-terminal [1, Section 3.10] or which tokens are derived first from given non-terminal [1, Section 3.8]. However, parsing techniques may differ in how easy it is to determine which non-terminal should be used.

3.2 Recursive-descent

Recursive-descent parsing [2, Section 4.4.1] is a top-down parsing method in which parsing is done via program procedures that represent non-terminals. Basically, the procedures look similarly to the one shown in Figure 8. The parsing itself starts by calling a procedure that corresponds to the start non-terminal. Typically, the code may be implemented manually or it is possible to have the parser generator create the code for the user.

```
    void A() {
1)    Choose an A-production, A → X1 X2 ... Xk;
2)    for (i=1 to k) {
3)      if (Xi is a nonterminal)
4)        call procedure Xi();
5)      else if (Xi equals the current input symbol a)
6)        advance the input to the next symbol;
7)      else /* an error has occurred */;
      }
    }
```

Figure 8: Procedure in pseudo-code for a non-terminal A (source: [2])

It is important to note a few things about the procedure from Figure 8. Firstly, there is a possible source of non-determinism at line 1) as there may be more than one production for non-terminal *A*. Secondly, if there were more than one possible productions, one possible strategy would be to keep trying until one of them succeeds or there are no more productions to try. This strategy requires a feature known as *backtracking* which is the ability to return back in the input and rescan some part of it. Another possibility is to use *predictive parsing* [1, Section 3.7] which inspects the unread part of the input and uses it to determine which production to choose. Since predictive parsing will be discussed later as part of LL(k) parsing, the rest of the text will deal with backtracking recursive-descent. The last issue to mention is that actual implementation of fully-fledged backtracking parser has to be a bit more sophisticated. For example, the code shown in Figure 8 does not build

the parse tree. It is also necessary to determine how far the parser may backtrack and when a given production can be considered to be a part of output parse tree. For detailed discussion of the issues see [3, Section 6.6]. The description given so far, however, should enable the technique to be assessed.

3.2.1 Generality

Since this techniques is from the top-down parsing family, it is susceptible to fail with left-recursive grammars. Indeed, if there is a direct left-recursive production, the corresponding procedure would just keep calling itself. The same is the case with the indirect left-recursion as it would be possible to end up calling the corresponding procedure without consuming the input. The inability to work with left-recursive productions is rather unfortunate as they can considerably simplify the grammar.

3.2.2 Ambiguity

In theory, there is no reason why the technique should not work with ambiguous grammar. However, in practise, the resulting parse tree would be influenced by the mechanism which is used to pick the productions in the procedure. In other words, if there are ambiguities, the parser might produce just one possible parse tree without being aware that some other parse trees were possible as well. Whether, this would be an issue is questionable though as ti depends on what the user intents to do with the parse tree and whether he or she is aware of the ambiguity in the grammar.

3.2.3 Error Handling

Due to the backtracking nature of the parser, it might not be immediately clear where the error occurred. The issue is that if the parser can backtrack arbitrary far, all possible productions may be tried over the input. What it means is that the parser might backtrack before the point in the input where the error is. Hence, without modifications to the parser it does not have the correct prefix property or immediate error detection property. Luckily, there is a way [3, p. 524] to enable the correct prefix property: *"the only thing that we must remember is the furthest point in the input that the parser has reached, a kind of high-water mark. The first error is found right after this point."*

After the error is detected, there is plenty of contextual information available as the partial parse tree is at our disposal. Therefore, it should be possible to provide good error messages and error recovery facilities.

3.2.4 Follow Suggestions

Since the partial parse tree is available, it should not be too difficult to give some follow suggestion. However, this feature is again complicated a bit by the backtracking nature of the parser: it is possible that there are more than one partial parse trees possible for given position. As a result, in order to enumerate all possible follow suggestions, it might be necessary to search for all possible parse trees which could be a costly endeavour.

3.2.5 Performance

While backtracking allows for a greater degree of generality in the grammar, its drawback is that the parser might require [8], [9] time exponential to the size of the input in order to to finish the parse.

3.3 LL Parsing

LL parsing is one of the basic parsing techniques [1 Section 3.11], [2 Section 4.4.3], [3, Section 8.2]. It is a deterministic top-down parsing technique meaning that it only works with unambiguous grammars. The LL in its name suggests that the input is read from **L**eft to right and the parser simulates the **L**eftmost derivation. There are two possible ways of implementing this technique: the first is essentially the predictive variant of recursive-decent parsing; the second employs a table and a stack that conceptually work as a deterministic automaton. In both cases, the technique uses lookahead in the input to guide the parse. It is often the case that the name of the method is given as LL(k) where k determines how many tokens are used as the lookahead.

Before discussing how the method works, it is necessary to define some concepts that are crucial to the method:

- FIRST(α) is a set of terminals which start the string derived from sentential form α
- NULLABLE(α) is a function that returns true if sentential form α can derive empty string
- FOLLOW(A) is a set of terminals which may follow after non-terminal A in given grammar

The algorithm that computes the sets and the function can be found in [1, pp. 73-80]. The sets can be generalised [3, Section 8.3.1] to give not just the first terminals but first k terminals. This modification is needed if the technique is to be used with lookahead greater than one token.

The following paragraphs will deal with the possible implementations of LL(1) parsing method. The demonstration on the simplest case of the LL method is sufficient as parsing with greater lookahead works similarly.

3.3.1 Automaton-based Implementation

The first step to take is to extend the grammar with new start production $S' \rightarrow S\$$ where S is the old start non-terminal and $\$$ is a new terminal that marks the end of the parse. The next thing to do is to create a parsing table. The conceptual scheme of the table is shown in Figure 9. The terminals in the first row are all possible terminals found in the extended grammar. The non-terminals in the first column are all non-terminals found in the extended grammar. The other cells of the table may be either empty or contain a production body appropriate to the non-terminal at the beginning of the row. The mechanism that determines which production body will be in which cell works as follows: given a production $A \rightarrow \alpha$, the production body α will be put into the row that starts with non-terminal A. The body is then put into the columns labelled with terminals that appear in the FIRST(α) or in FOLLOW(A) if NULLABLE(α) returns true. Lastly, non-terminal S' is put on the top of the stack.

	terminal	terminal	...
non-terminal	production body	-	...
non-terminal	production body	production body	...
...	...	...	...

Figure 9: Parsing table template

The parsing procedure uses the table, the stack and a pointer to the current position in the input string that determines which part of the input has yet to be read by the parser. As mentioned above, the LL parsing simulates the leftmost derivation of the string. This is achieved by following operations:

- If a non-terminal is on the top of the stack, get the next symbol in the input string and use it to found the appropriate production body in the table. If there is no record in the table, report an error. If there is a production body it is the one that should be used in the next derivation step. Then pop the non-terminal form the stack and replace it with the production body so that the left-most symbol in the production body is on the top of the stack. Thus, a derivation step is simulated.

- If a terminal is on the top of the stack, determine whether it is the same as the next symbol in the input string. If it is, pop the terminal from the stack and advance the pointer. If the terminals do not match, report an error.
- Once the input is read and the stack is empty, the parsing is complete.

The operations implement only a recogniser, though. Fortunately, the parse tree should be fairly easy to construct by appropriately modifying the operation that simulates the derivation step.

There was one issue omitted in the discussion of table construction: What happens if there are two possible production bodies for given terminal and non-terminal? This situation is know as *conflict*. If a conflict arises, it is not possible to construct the deterministic parsing table for given grammar. The solution to the problem is to try to change the grammar using technique know as left factorization [1, Section 3.12.2]. However, this is not always possible according to [1, p. 83]: *"There are languages for which there exist unambiguous context-free grammars but where no grammar for the language generates a conflict-free LL(1) table."*

3.3.2 Recursive-descent Implementation

The recursive-descent version of LL parsing uses the FIRST and FOLLOW sets to choose appropriate production as shown in Figure 10. Thus, it is able to dispense with backtracking. The code for handling different production bodies would be quite similar to the one described in Section 3.2.

```
void A() {
  if NEXT_SYMBOL in FIRST(α1 FOLLOW(A))
    code for α1
  else if NEXT_SYMBOL in FIRST(α2 FOLLOW(A))
    code for α2
  else …
  …
  else if NEXT_SYMBOL in FIRST(αk FOLLOW(A))
    code for αk
  else /* an error has occurred */;
}
```

Figure 10: Procedure in pseudo-code for a non-terminal A with productions $A \rightarrow \alpha_1 | \ldots | \alpha_k$. The `FIRST(α1 FOLLOW(A))` is a shorthand version of the rule used for filling the cells of the parsing table. (source: [3])

3.3.3 Final Notes

Firstly, the presented way of using FIRST and FOLLOW sets is not the only one possible. Other versions are presented in [3, p. 251]. Basically, the presented parser may not detect some errors immediately: if the grammar contains productions with empty right-hand side, they may be tried before the error is detected. It is possible to remedy the situation at the cost of increased size of the parse table.

Recursive-descent version is usually used in handwritten parsers. It should be noted that this variant of creating parser by hand should not be taken lightly as some of the mainstream compilers like Clang [10] or GCC [11] were created in similar fashion. Since they might parse languages that are not parsable by LL parsers, they may not be considered LL parsers per se, but it is quite likely that their creators were inspired be them. Naturally, the downside of this approach is that it requires a lot of effort compared to the use of a parser generator.

It is impossible to discuss LL parsing without mentioning ANTLR Parser Generator [12], probably the most well-known implementation of the LL concept. The ANTLR does not use the typical LL(k) approach; its parsing technique is known as LL(*) [13]. Basically, it employes an unbounded look-ahead for choosing the productions. If this approach fails, it falls back to backtracking. However, it uses a technique [14] known as *memoization* which prevents the exponential behaviour by storing some intermediate results so that can be reused if they are required again during parsing. One way to think about this is that the stored intermediate results are the results of recursive procedures on given input. The cost of this approach is that it increases memory requirements. Another feature of the LL(*) is that it allows certain predicates in the productions; result of which is increased expressiveness in terms of recognised languages. Unfortunately, this technique cannot handle the left-recursive grammars.

3.3.4 Generality

Since the technique is of top-down parsing family, it cannot cope with the left-recursive grammars. Another factor that determines the expressiveness is the value of the lookahead parameter k in LL(k). According to [3], the set of LL(k) grammars is a proper subset of the set of LL(k+1) grammars for any k. Moreover, there are languages that require some minimal lookahead for the parser to recognise them. In other words, some languages are describable only by LL(k) grammars with some minimal value of k. Generally, the LL parsers are less expressive than LR parsers which are discussed later on.

3.3.5 Ambiguity

This technique in its raw form precludes the use of ambiguous grammars. However, it might be possible to allow some additional mechanism that determines which production to use if more are possible.

3.3.6 Follow Suggestions

The follow suggestions can be obtained quite easily as the next expected symbol is either on the stack or it can be decoded from the parse table. The recursive-descent version should enable this feature as well, although the procedures might have to be mildly altered to offer this capability.

3.3.7 Performance

The time complexity of the parser is linear to the size of the input [3, p. 260]. In the case of LL(*), the complexity ranges form linear to quadratic [13].

3.3.8 Error Handling

The parser has correct prefix property, however, it may not have immediate error detection property due to the issues discussed above. Nevertheless, there is plenty of contextual information so the parser knows what production it is trying to simulate which lends it great advantage. Truly, the good options for error handling are regarded [13] as a good reason to use a LL parser.

3.4 LR Parsing

LR parsing is one of the oldest parsing techniques, being created by Knuth in 1965 [15]. It is a table-based bottom-up deterministic parsing technique [3, Sections 9.4-9.9]. It cannot parse ambiguous grammars. The runtime is linear to the size of the input. The name of the method is meant to suggest that the input is read from **L**eft to right and the parser simulates the **R**ight-most derivation. The full name of the technique is LR(k) where k refers to the number of tokens used as a lookahead. If no number is given, it is assumed to be LR(1).

As a technique based on the idea of bottom-up parsing, LR parsing needs to represent the sentential form that is being rewritten and identify the correct handle to reduce. For this purpose a stack and a deterministic automaton are used. The automaton is used to identify the handles and the stack is used to store the sentential form reduced so far. As mentioned above, the technique is table-based; the reason for that is that the automaton is represented by a table. There are various versions of the technique, namely SLR, canonical LR, LALR,

which differ in the construction of the tables and the classes of languages that they can parse.

The most important part of the technique is the automaton. We will use the simplest one, the LR(0) automaton, to illustrate how the technique works. Before discussing the automaton itself, it must be noted that the grammar has to be extended with new start production $S' \rightarrow S\$$ where S is the old start non-terminal and $\$$ is a new terminal that is put after the input and marks the end of the parse. Now to the automaton: an example of it is shown in Figure 11. The automaton is represented as a graph. The nodes represent sets that hold productions annotated with the • symbol which represents what part of the handle has been recognised so far. These annotated production are called *LR(0) items*. The nodes are called states. The labels on the edges define the grammar symbols on which a transition form one symbol to another occurs. The state number 1 is an initial state, since it contains the item $S \rightarrow \bullet E\$$ which is a start production whose handle has not been read yet. The state number 5 is a final state as the item it holds is the recognised start production $S \rightarrow E\$\bullet$. Once the states 2,3,5,8 or 10 are reached, it is possible to reduce the recognised handle.

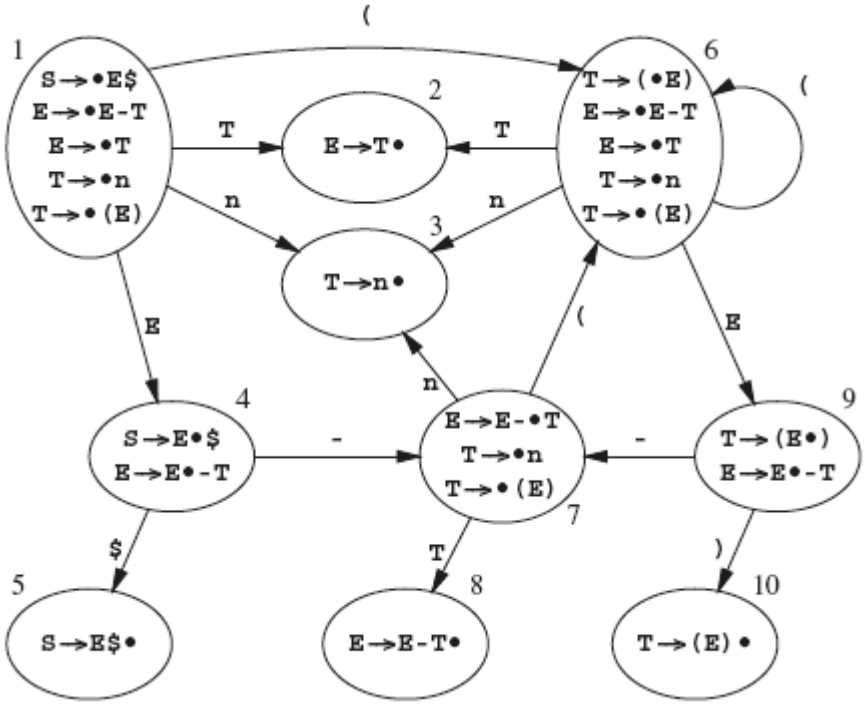

Figure 11: LR(0) automaton for simple subtraction arithmetic expression grammar represented as a graph. (source: [3])

With the automaton roughly described, it is now the time to focus on the second ingredient of the LR parsing: the stack. Firstly, when the parsing starts, the stack is empty. Secondly, there are two operation – *shift* and *reduce* – that take place on the stack. The parser uses the stack as follows: The LR(0) automaton is fed the content of the stack from the bottom to the top, i.e. the sentential form is scanned from the left to the right. The automaton starts from the initial state and changes the states based on the input. Depending on the state it ends up in, there are four scenarios possible:

1) The last state contains a single item $A \rightarrow \alpha\bullet$, i.e. the handle α has been recognised. As a result, the same number of symbols as is the length of α is popped from the stack and non-terminal is A put on the top of the stack. This is the *reduce* operation.
2) It is not possible to reduce from the last state. Therefore, another terminal has to be taken from the current position in the input and put on the top of the stack. That is, it is shifted from the input to the stack: this is the *shift* operation.
3) The last state is the final state: the parsing is finished.

4) There is an invalid terminal on the stack on which no transition is possible from the current state. This means that there is an error in the input and the invalid terminal is the cause.

The parsing is thus a process of continually rescanning the stack and applying one of the four scenarios given above. However, this is not very efficient as there is a part of the stack that has not changed, yet it has to be rescanned every time. The solution is to hold the states on the stack as well and add them and pop them accordingly. An example of such approach is shown in Figure 12.

stack	*input*	*action*
$_1$	**n**-**n**$	shift **n**
$_1$**n**$_3$	-**n**$	reduce T→**n**
$_1$T$_2$	-**n**$	reduce E→T
$_1$E$_4$	-**n**$	shift -
$_1$E$_4$-$_7$	**n**$	shift **n**
$_1$E$_4$-$_7$**n**$_3$	$	reduce T→**n**
$_1$E$_4$-$_7$T$_8$	$	reduce E→E-T
$_1$E$_4$	$	shift $
$_1$E$_4$$$_5$		accept

Figure 12: Example of parsing n-n with the stack. The top of the stack is on the right. The small numbers represent the state of the automaton from Figure 11.

So far, only the inner working of the parser has been discussed. Now, it is the time to investigate the LR(0) automaton. Since it is the only part of the parser that depends on the grammar, it is the task of parser generator to create it. The algorithms for doing so are not that important; they can be found in [2, Section 4.6.2]. The important feature of the way the automaton is used is that it is deterministic: its always possible to determine whether we have to reduce or shift in given state. Unfortunately, a grammar might give rise to an LR(0) automaton without this nice feature. If such situation occurs, it is termed as a *conflict*. There are two kinds of conflicts known as *shift/reduce* and *reduce/reduce* conflicts. The former refers to the fact that in given state there is an item whose handle has been recognised and an item whose handle is still being recognised. As a result, it is impossible to tell whether to shift or to reduce; a conflict arises. The reduce/reduce conflict refers to similar situation: in this case there are two possible handles recognised; the result

of which is that it is impossible to determine which one is to be reduced. It should be noted that there are further issues if empty productions are involved. Finally, it must be mentioned that the automaton and its related shifting and reducing capabilities are usually defined by two tables called ACTION and GOTO. The first one indicates when to shift and when to reduce. The second one defines the transitions between individual states.

In order to get rid of conflicts in the automaton, it is possible to extend the items with lookahead information which in turn leads to refinement of the states in the automaton. This approach is used by canonical LR(1) parser. The number in the brackets refers to the fact that the automaton uses one token lookahead to distinguish the items. Unfortunately, the number of states may be too great. In order to address this issue a LALR version of this parsing technique was created. The automaton of this technique combines some states in a way that does not produce conflicts. Grune and Jacobs provide more detailed discussion in [3, Section 9.6-9.7].

As mentioned above, there are three main versions of the LR parsing: SLR, LALR and canonical LR. The order in which they are named represents their strength in terms of the size of the class of languages parsable by them. That is, there are languages that can be only parsed by canonical LR parsing, but not by LALR and SLR, etc. In practice, the parser generators, such as Bison [16], tend to use the LALR version due to the size of the parsing tables.

LR parsers have some interesting properties. For example, it is possible to have LR(k) with k>1; however, they are not considered practical as their tables can have enormous sizes [3, p. 299]. Another interesting property of LR parsing is that any language parsable by LR(k) with k>1 can have its grammar transformed to the one that can be parsed by LR(1) parser [3, Section 9.6.3]. The advantage of the technique over LL parsing is [1, p. 88] that *"... there are also languages for which there exist LR-acceptable grammars but no LL(1) grammars."*

The discussion of the features will be restricted only to the LALR and LR versions.

3.4.1 Generality

As mentioned above, the technique allows grammars that are more general than those of LL parsing techniques. Specifically, the left-recursive grammars are not precluded by this technique. The trouble is that if the grammar is not trivial, it might be difficult to pinpoint what productions cause the conflicts in the states of the automaton.

3.4.2 Ambiguity

Since the technique is deterministic, it does not allow ambiguous grammars. The ambiguities in the grammar lead to conflicts in the automaton. It might be possible, though,

to use some conflict-resolving techniques [1, Section 3.16.3], [3, Section 9.9] to allow ambiguity in the grammar. For example, it might be possible to declare precedence and associativity for the operators of the grammar.

3.4.3 Follow Suggestions

It is possible to extract the next expected tokens from the automaton by investigating outgoing transitions from the current state. The downside is that the only contextual information available are the handles recognised so far.

3.4.4 Performance

It is an efficient parsing technique with time complexity linear to the size of the input. In practise, the LALR version is faster than canonical LR due to the lesser size of its tables.

3.4.5 Error Handling

When it comes to error detection, the following holds [2, p. 283]: *"A canonical LR parser will not make even a single reduction before announcing an error. SLR and LALR parsers may make several reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack."* Consequently, the canonical LR parser has both immediate error detection property and correct prefix property. LALR parsers, on the other hand, have only correct prefix property. Generating helpful error messages might be an issue as if an error occurs it is because there was no transition on the input symbol. Furthermore, the state may contain more than one production, therefore it may be impossible to determine which one to use in the error report. As a result, it may be only possible to state what symbols were expected.

3.5 Generalised LR

Despite the fact that LR parsing works with no small number of grammars, there are still some unambiguous grammar it cannot parse. Furthermore, as noted before in Section 3.1.3, ambiguity may be beneficial. In order to allow these features, a generalised version of LR parsing (GLR parsing) was created: its detailed description can be found in [3, Section 11.1].

The automatons of LR techniques cannot be used if they contain states with the reduce/reduce and shift/reduce conflicts. When such things happens, the choice of appropriate action is non-deterministic. One way to circumvent this restriction is to perform all the actions possible on the copies of the stack. If an error occurs, the stack is ignored in consequent parsing. The trouble with this approach is that it may lead to

exponential explosion of the copies of stack. There is a way to prevent this, though: Some stack prefixes and suffixes may be shared among the copies as they are the same. A structure called *graph-structured stack* is used for this purpose. It is a directed acyclic graph whose nodes represent states and whose edges are labelled with grammar symbols that form the sentential form stored on the stack. An edge point to the node from which their node of origin was inferred. An example of such this structure is shown in Figure 13. As it is intended for demonstrative purposes, it is not important what the grammar or the automaton looks like.

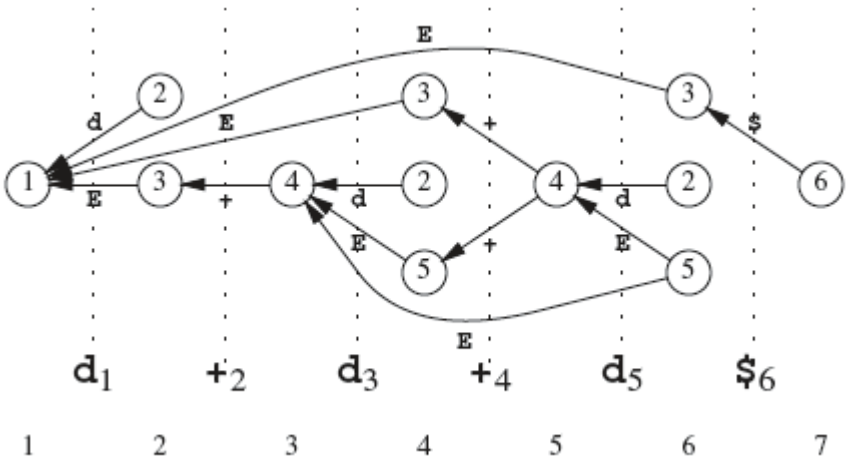

Figure 13: A graph-structured stack example. The parsed expression is d+d+d$. (source: [3])

Even though the technique is rather easy to describe, there are various issues involved. For example, left-recursive rules might complicate the parsing [3, p. 387] as well as the presence of empty productions. Another topic is the generation of the parse tree: in its simplest form, it might be possible only after the whole input was read [3, p. 386].

Nevertheless, there are parser generators that use this technique. For example, it is supported by the aforementioned Bison parser generator [16]. Another real-world implementation is Elkhound parser generator [6], [17].

3.5.1 Generality

As mentioned before, the GLR parsers can parse any CFG grammar.

3.5.2 Ambiguities

Since all CFG should be parsable by the technique, ambiguous grammars can be parsed easily. The trouble might be obtaining a compact representation of parse trees if the grammar is highly ambiguous. A technique called *shared packed parse forest* [18, Section 1.3], which works similarly to the graph-structured stack: the same parts of the parse trees are grouped together, is used to solve the issue.

3.5.3 Follow Suggestion

All it takes to suggest the next possible terminal is to inspect all "tops of the stacks" in the graph-structured stack.

3.5.4 Performance

The technique can be made to run with $O(n^3)$ time complexity [3, p. 390].

3.5.5 Error Handling

The issues with error handling in GLR parsing are quite similar to those of LR parsing. An error is detected when the parsing cannot continue from any state on the top of the stacks. Hence, the parser has the correct prefix property. The immediate error detection property depends on the automaton used.

3.6 PEG and Packrat Parsing

All the parsing techniques discussed so far relied on the formalism of context-free grammars. However, this is not the only road to take. The acronym PEG from the headline stands for *parsing expression grammar* [19]. The form of these grammars is quite similar to the CFG grammars: there are still terminals, non-terminals; the productions are referred to as rules and the semantics differ slightly. Unlike the productions, the rules, rather than being used for generating sentences, are used for recognition which is also expressed by using an arrow ← in the definition of the rules. In other words, given a rule A←B C, the meaning of the rule is [19, p. 21]: *"To read an instance of nonterminal A from a string, look for an instance of nonterminal B followed by an instance of nonterminal C, possibly followed by additional, unrecognized input. If both B and C are found, then succeed and consume the corresponding portion of the input string. Otherwise, fail and do not consume anything."* Moreover, the right-hand side of the rule is not composed only from terminals and non-terminals; it may also contain symbols that define repetition, choice, sequence or optionality of a group of symbols or what can or cannot follow after given symbol, etc.

These constructs with terminals and non-terminals then form what is known as a *parsing expression* – for more details on these see [19, p. 21]. Parsing with PEGs can also dispense with the traditional lexical analysis phase: the tokens are described explicitly in rules by parsing expressions. Another important feature is that the body of the rule is matched greedily, i.e. it tries to match as much of the input as possible. Furthermore, the alternates of the rule are tried in the same order as they are defined; what it means is that when trying to recognise nonterminal A with rule $A \leftarrow B\ C \mid D\ E$, the first alternate $B\ C$ is always tried first and the second one is tried only if $B\ C$ was not matched. As a result, with PEGs, there is only one parse tree possible. Consequently, the grammar cannot be ambiguous.

Parsing with PEGs is basically a top-down approach: we start from the top rule and try to match its body. If matching fails, another alternative has to be tried instead. This is, basically, a top-down recursive-descent parsing with backtracking. Consequently, the rules for matching the non-terminal can be sees as a function that either succeeds or fails. The trouble is that backtracking can lead to exponential runtime. Left-recursion is a problem as well. Although a technique for dealing with left-recursion was presented in [20], later investigation [21] showed that it may contain some flaws.

To alleviate the performance problem, a *packrat parser* is used. This parser uses the memoization technique to speed up the execution at the cost of increased space requirement. As mentioned above, the rules of PEGs can be considered functions. The memoization is used as follows [8]: *"The basic reason the backtracking parser can take super-linear time is because of redundant calls to the same parse function on the same input substring, and these redundant calls can be eliminated through memoization."* As a result, it is able [8], [19], [22] to parse the input with linear time complexity. A possible downside of memoization is that the parse functions are presumed not to have any side-effects. In other words, as stated by Ford [8]: *"...the parsing function for each nonterminal depends only on the input string, and not on any other information accumulated during the parsing process."*

There are various implementations of packrat parsers. The memoization technique used in packrat parsers can [8] be expressed implicitly in functional languages without strict evaluation: *"A non-strict functional programming language such as Haskell provides an ideal implementation platform for a packrat parser."* There are also implementations in imperative languages such as Rats! [22] or Parboiled [23] for Java.

3.6.1 Generality

The PEGs are different formalism than CFG. Furthermore, their relationship is not clear-cut [24]. There are some languages with no CFG that can be parsed by PEGs. However,

whether there are context-free languages that are not parsable by PEGs is unclear. Despite the work on the issue, left-recursive rules are not supported [21].

3.6.2 Ambiguities

PEGs does not allow for ambiguities due to the semantics of its rules.

3.6.3 Follow Suggestions

It should not be difficult to extract what is expected next. However, due to the fact that the packrat parsers typically do not use lexers, it might not be completely easy. For example, if a number is defined through parsing expression as a repetition of digits, the follow suggestion might end up being a digit. Similarly, if a keyword is expected, only its first letter might appear as a suggestion. These issues are probably amendable; they had to be noted, though.

3.6.4 Performance

As noted above, the worst-case time complexity of PEGs is exponential; however, the optimization techniques of packrat parser can lower it to linear time complexity at the cost of increased space complexity. Despite the increased space complexity, the packrat parsers are deemed [22] usable on modern hardware.

3.6.5 Error Handling

Due to the similarity with the top-down backtracking parsers, the issues are virtually the same – see Section 3.2.3. Error handling should be possible as it is implemented in the real-world parsers (e.g. Parboiled [23] which also supports error recovery). The difficulty or ease of implementing error handling is unclear, though: Parr [13, p. 426] even argues that packrat parsers *"...cannot recover from syntax errors because they cannot detect errors until they have seen the entire input."*

3.7 Generalised LL

Generalised LL (GLL) parsing is the youngest of the presented techniques, being first described by Scott and Johnstone [25] in 2010. Interestingly, they note that this technique has been inspired by their work on variants of GLR parsing.

Essentially, troubles with recursive-descent LL(1) parsers arise if the choice of the next production to use, based on the FIRST or FOLLOW sets, is not deterministic or if the left-recursion in grammar leads to an infinite loop of procedure calls. The first issue can be solved by pursuing all the choices simultaneously. However, this is difficult to achieve with

implicit call stack which is used for handling execution of recursive functions and procedures. To address the issue, the call stack is in GLL handled explicitly. Simultaneous execution demands that each pursued execution path has its individual call stacks which, in turn, leads to issues with performance if the grammar is ambiguous. However, all stacks may share some common parts much like the stacks in GLR method. Consequently, GLL uses graph-structured stack to represent the call stacks. The procedures are dispatched using labels and **goto** statements.

The GLL parsing can be made to run with cubic worst-case time complexity if appropriate data structures are used – for more details on the implementation see [26].

3.7.1 Generality

GLL parser should be able to parse any CFG grammar.

3.7.2 Ambiguity

Ambiguous grammars are not an issue as the parser handles all CFGs.

3.7.3 Follow Suggestions

As well as in the case of traditional recursive-descent parsing, the procedures have to be prepared for providing the suggestions. This should be achievable, however.

3.7.4 Performance

As mentioned above, the parser, if proper data structures are used, has the worst-case $O(n^3)$ time complexity. There is an implementation of the GLL parser called GLL combinators [27]. The notes of its author suggest [27] that the performance of the implementation might not be optimal: *"At the moment, performance is basically non-existent. The GLL algorithm itself is O(n^3) even in the worst case, but there is a high constant factor which is introduced by the framework which makes this quite a bit slower than it sounds."* Although he later remarks that the situation might not be as bad for grammars without high ambiguity.

3.7.5 Error Handling

Error detection should be rather straight forward: if no procedure on the top of the graph-structured stack succeeds, there is an error. Consequently, GLL is likely to have both immediate error detection property and correct prefix property. Due to these properties and the top-down approach of the method, it should provide good error messages. The quality of error correction is rather uncertain as its availability depends on how easily it can be

incorporated into the special-purpose data structures used by the technique to ensure its cubic runtime.

3.8 Earley Parsing

Earley parsing is a general parsing method invented by Jay Earley [28]. It can be described [3, p. 206] as a *"breadth-first top-down parser with bottom-up recognition"* or a *"bottom-up method with a top-down component"*.

For an input of length n, the parser constructs a sequence of $n+1$ sets, known as *Earley sets*, that contain information about what productions were recognised and to what part of input they match [29]. This information is represented as *Earley items* which consist of two parts: the first one describes what part of a production has been seen so far (very much like LR(0) items in LR(0) automaton) and a pointer to the set where their recognition started. The items are often represented as $[A \rightarrow \alpha \bullet \beta, i]$ where the first part has the same meaning as LR(0) items and i is the aforementioned pointer.

There are three operations that create the sets: *scanner*, *predictor* and *completer*. Traditionally, these three steps are applied to the items of last created set until no more items can be added; during these steps, the scanner operation initialises items for the next set. However, the technique might be easier to understand, if the last set is considered completed. With this slight and admissible modification, the scanner operation takes a next token/terminal a from the input and finds all items in the last set which expect this terminal to appear next; i.e. they are in the form of $[A \rightarrow \alpha \bullet a\beta, i]$. All such items are then modified appropriately (the dot is advanced over a) and put into the new set. Now, there are two things to do with the new incomplete set. Firstly, we have to determine what productions may be recognised at this position and update the set: this is the duty of the predictor operation. Hence, for each item $[A \rightarrow \alpha \bullet B\beta, i]$ we add a new item $[B \rightarrow \bullet\gamma, k]$ where k is a pointer to the current set. Secondly, we have to check if a recognition of some production has been completed by the scanner operation. If so, there is an item $[B \rightarrow \alpha \bullet, j]$ in the new set. This item was added as a result of a predictor operation on item $[A \rightarrow \alpha \bullet B\beta, i]$ in some previous set. Consequently, it is necessary add $[A \rightarrow \alpha B \bullet \beta, i]$ to the current set in order to keep the information on what has been recognised so far up-to-date. Naturally, the addition of a new item may necessitate another completer or prediction operation on the added item. As a result, the completer and predictor operations are executed on the set until no new item is added. Once this happens, the set is finished and a new set may be created by the scanner. The last thing that has to be mentioned is that the grammar is extended[1] with a new start production $S' \rightarrow S$. The production is used for the initialisation of the first set:

1 Although it is not necessary: it is done for convenience.

Earley item [$S' \rightarrow \bullet S,\ 0$] is added to the set and then the predictor and completer operations are applied. This step is the aforementioned top-down component and it ensures that the only productions derivable from the root are considered for recognition at given position in the input. The parsing is successful if the last set contains item [$S' \rightarrow S\bullet,\ 0$].

With the construction of sets and items covered, it is necessary to investigate their meaning. Earley set can be viewed as a collection of expectations for the next token in the input based on what has been seen so far. The first set is created before any of the input is read: the fact that all the items are obtained via prediction from the start non-terminal signifies that the next terminal has to be one of the firsts derivable from the start non-terminal. The significance of items and sets can be seen from alternative representation shown in Figure 14 which contains a completed parse of a string $a+a$ with following grammar:

$$S \rightarrow E$$
$$E \rightarrow E+E|E-E|V$$
$$V \rightarrow a|b|c$$

Basically, the nodes represent the sets while edges labelled with terminals specify the terminal used in scanner operation to obtain one set from another. An edge labelled with dotted production represents a recognition of a part of input: the starting point of the edge determines where the recognition started and the end point specifies how far it got. For example, an edge labelled with $E \rightarrow V\bullet$ starting at set *0* and going to set *1* means that it matches a part of input a. If translated back to our representation, it is the same as item [$E \rightarrow V\bullet,\ 0$] in set *1*.

What has not been specified yet is how the tree can be obtained from finished parse. In order to achieve this, some additional information has to be stored. Aycock and Borsotti suggest [30] to store two following types of information:

- *predecessor link* from I to J: if an item J was added as a result of advancing a dot over a symbol in item I.
- *causal link* from I to J: it is additional information to predecessor link which indicates that completion of J was the cause why I was added.

Each item has pairs of these links associated with it which allows to recreate the right-most derivation from parsed sequence of items – see [30, p. 551].

There is one issue with the recognition algorithm as presented: it has troubles with empty productions. There are different approaches to address the issue [3, Section 7.2.3]. Very elegant solution [29] is to modify the predictor step so that if the dot in item is before nullable non-terminal, an item with the advanced dot is also added to the set.

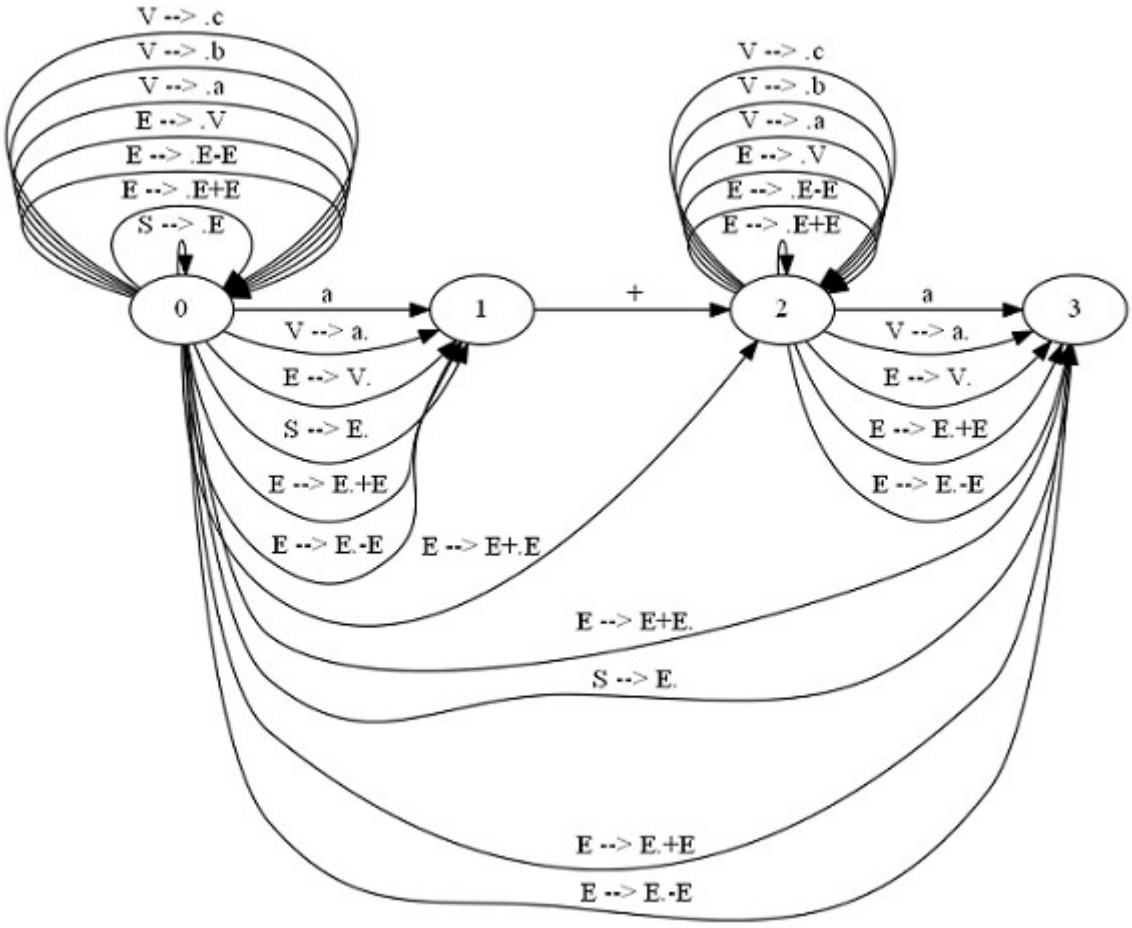

Figure 14: A representation of the parse run for input a+a.

3.8.1 Generality

The method parses all CFGs.

3.8.2 Ambiguity

Ambiguous grammars can be parsed as well.

3.8.3 Follow Suggestions

It is easy to obtain the suggestion from the Earley set as it is a question of going through all the items and noting all the terminals (or even non-terminals) that are expected at given point.

3.8.4 Performance

As other general parsing methods, Earley parsing has worst-case cubic time complexity. Although in practice, it can be expected to behave linearly on practical grammars and it has $O(n^2)$ time complexity for unambiguous grammars [3, p. 212]. The downside is that it may have greater memory requirements; theoretically $O(n^2)$, although it should be much better in practise.

3.8.5 Error Handling

Parsing fails if the scanner operation yields an empty set. As a result, the parser can be considered to have both *immediate detection property* and *correct prefix property*. Furthermore, since the parsing is represented via a sequence of sets, error recovery should not be an issue – a part of sequence can be discarded or expected artificial terminals may be fed to it in order to recover.

3.9 Technique Choice

The previous sections reviewed the candidate parsing techniques for our parser generator implementation. Now it is time to choose one. Firstly, it has to be noted that the choice is not easy. Every parsing technique has some trade-offs as is often the case with problems and techniques in computer science.

Possibly with the exception of backtracking-based techniques, every discussed method may provide the required features – error correction and follow suggestions. The difference lies in how easy it is to implement and what grammars are acceptable by the technique. Despite the amenability to error handling, the lack of support for left-recursive grammars in the most prevalent top-down parsing techniques incurs too much of a burden on the user. Left-recursion is necessary to allow such basic features as left-associative operators. While deterministic LR-based methods from the realm of bottom-up techniques allow for greater expressivity in the grammar than their LL counterparts, it might be equally difficult to pinpoint what part of the grammar causes the non-determinism in the parsing table. What is left are the general parsing techniques: GLR, GLL and Earley. Interestingly, GLR and Earley may be considered [6] the same with the former being optimised form of the latter.

Since the idea GLL was conceived from a variant of GLR, all these techniques are related in certain sense. Regarding the performance, all techniques share the worst-case $O(n^3)$ time complexity. It is not that much of an issue though as the parsed strings are expected to be rather short. Our technique of choice is Earley parsing as it is quite amenable to error correction as argued in Section 3.8.5.

4 Implementation

The topic of implementation can be divided into two areas: the used Earley algorithm and the design of the parser generator.

4.1 Algorithm

The basic overview of Earley algorithm is given in Section 3.8. Despite the fact that the performance is not the sole concern, a more efficient implementation by Aycock and Horspool described in [29] was chosen. This version of Earley parsing uses a modified version of LR(0) automaton to speed up its execution. Another important feature is that the input grammar is slightly altered in order to deal with the presence of epsilon productions.

4.1.1 Basic Operations

The Section 3.8 gives a broad description of the scanner, predictor and completer operations. In order to give proper formal definition of the technique, we will formalise the three operations. Suppose the input has a form of $x_1 \ldots x_n$ and the resulting sequence of Earley sets is $S_0 \ldots S_n$. The Earley set S_{i+1} is initialised from completed Earley set S_i by following operation:

- *Scanner*: For each $[A \rightarrow \alpha \bullet x_{i+1} \beta, i]$ in S_i, add $[A \rightarrow \alpha x_{i+1} \bullet \beta, i]$ to S_{i+1}.

The creation of the set S_{i+1} is then finished by applying the following two operations until no new item is added:

- *Predictor*: If item $[A \rightarrow \alpha \bullet B \beta, j]$ is in S_{i+1}, add item $[B \rightarrow \bullet \gamma, i+1]$ to the set. Furthermore, if B is nullable, add item $[A \rightarrow \alpha B \bullet \beta, j]$ as well.
- *Completer*: If item $[B \rightarrow \alpha \bullet, j]$ is in S_{i+1}, add $[A \rightarrow \alpha B \bullet \beta, k]$ for all items $[A \rightarrow \alpha \bullet B \beta, k]$ in S_j.

As discussed previously, grammar is extended with new start production $S' \rightarrow S$, the item $[S' \rightarrow \bullet S, 0]$ is added to the initial set and operations corresponding to predictor and completer are performed on it. The final set has to contain $[S' \rightarrow S \bullet, 0]$ for the parsing to finish successfully. This is how the basic Earley recogniser operates. Note that the predictor step is extended, as described by Aycock and Horspool, to deal with epsilon productions.

4.1.2 Grammar Modification

The extension of the predictor step is not enough to deal with epsilon productions if the parse tree is to be created. The method suggested by Aycock and Horspool transforms the

input grammar into *nihilist normal form* (NNF). The purpose of NNF is to enable distinguishing whether a dot was moved over a non-terminal in Earley item by predictor step due to the non-terminal being nullable or due to non-epsilon production being recognised. This is achieved by following steps:

- All epsilon productions $A \rightarrow \varepsilon$ are replaced by $A_\varepsilon \rightarrow \varepsilon$.
- Add a new explicitly-nullable non-terminal A_ε for each nullable non-terminal A.
- For each production that contains nullable non-terminal A, add a new one with A_ε. If a production contains multiple occurences of A, all possible combinations must be enumerated.
- If some production has only explicitly-nullable non-terminals on its right-hand side, the non-terminal on its left-hand side is replaced by its explicitly-nullable counterpart.
- It might happen that some non-terminal becomes redundant as only its explicitly-nullable form is present in the productions.

An example of such transformation is shown in Figure 15. The result of the transformation is that only non-terminals subscripted with ε can derive an empty terminal. Another corollary of the transformation is that there may be two root non-terminals: one that is nullable and another one which is not nullable. Since the grammar is intended for parsing, this is not an issue: if there is a nullable root, the empty input is automatically accepted. Moreover, it is rather unlikely that the user would like to perform any operations on a tree for empty input.

The transformation might raise the question: what is its impact on the size of the grammar? Naturally, this depends on the number of epsilon productions in the grammar. In order to approximate the impact on a real-world grammar, Aycock and Horspoolperformed a test [29, Section 8] in which they transformed ten publicly availabile programming language grammars into NNF. The languages included C, C++, Pascal, Delphi, Ada, Java 1.1 and others. The result was rather positive; according to their report: *"The conversion to an NNF grammar did not even double the number of grammar rules; most increases were quite modest."*

With the grammar transformed into NNF, the steps described in the previous section have to be adjusted to accommodate the explicit nature of the nullable non-terminals. Basically, whenever an item is created or the dot is moved in an item, the position of the dot has to be adjusted so that it is automatically moved over nullable non-terminals.

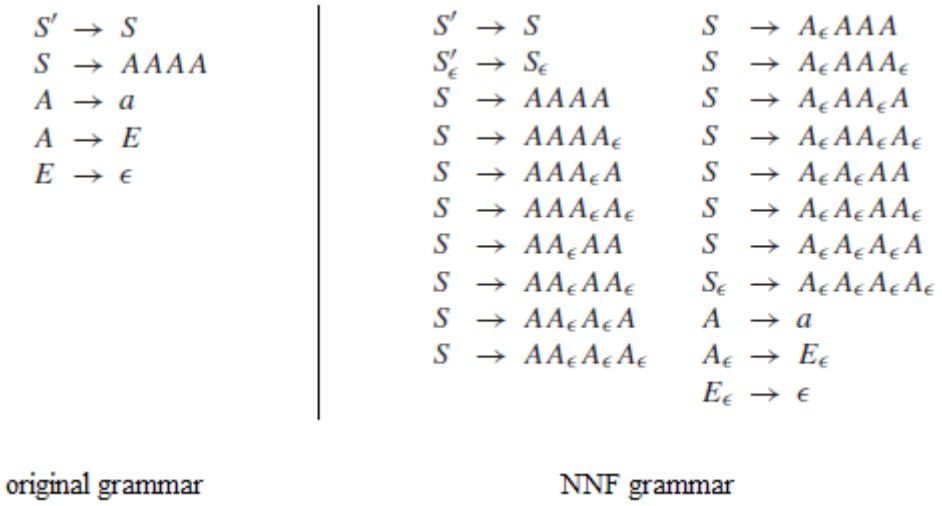

Figure 15: A sample ambiguous grammar transformed to NNF grammar. (source : [29])

4.1.3 Automaton

The reason for using an automaton in Earley parsing is quite simple: it is basically a form of pre-computation of groups of Earley items that appear together. Thus, a possibly large group of Earley items can be added to an Earley set in one step. In other words, the form of an Earley item becomes [*automaton state*, *pointer to parent set*]. Aycock and Horspool show [29] a modified version of LR(0) automaton that can be used in Earley parser. The use of the automaton yields performance approximately two times better than that of typical Earley parser.

One the main differences between typical LR(0) and Aycock-Horspool (AH) automaton is in the handling of dot movement behaviour: the nullable non-terminals described in the previous section are treated differently; when creating an AH item, the dot is automatically moved over adjacent nullable non-terminals. The same condition also applies if the dot is advanced. For example, corresponding AH item for the production $S \rightarrow A_\varepsilon A_\varepsilon AA$ is $[S \rightarrow A_\varepsilon A_\varepsilon \bullet AA]$ not the $[S \rightarrow \bullet A_\varepsilon A_\varepsilon AA]$. Similarly, if the dot is to be advanced over the second non-terminal A in $[S \rightarrow A \bullet AA_\varepsilon A]$, the resulting item is $[S \rightarrow AAA_\varepsilon \bullet A]$. The last thing to note is that the AH items with nullable non-terminals on their left-hand side are not included in the states.

Another feature of AH automaton is somewhat similar to the concept from the regular LR(0) automaton. The items of the traditional LR(0) automaton may be viewed [2, p. 246] as:

- *kernel items*: The root item $S' \rightarrow \bullet S$ and all items without the dot on the left end.
- *non-kernel items*: All items with the dot on the left end except the root item.

The terms *kernel* and *non-kernel* refer to the fact that all LR(0) items in given set may be derived only from the kernel items. Indeed, it has been suggested [2, p. 245] that: *"we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that they could be regenerated by the closure process."* The AH automaton items may be divided into these two categories as well. However, the definition of the items is altered in order to reflect the presence of nullable non-terminals:

- *non-kernel items*: All items with the dot on the left end and those derived from adding the productions with nullable non-terminals in the beginning to the state. The exception to this rule is the root item $S' \rightarrow \bullet S$.
- *kernel items*: All other items.

Moreover, this division is used to split the states into the *kernel* and *non-kernel* states. The reason for doing is related to the steps of Earley algorithm: presuming that I and J are LR(0) items in a non-splitted state s, the non-kernel item I can only be present in set S_k due the predictor step which means that the related Earley item would have the form of $[I, k]$. On the other hand, the kernel item J can be added to S_k by the scanner or completer steps which results in the corresponding Earley item $[J, i]$ where i may not be equal to k. As mentioned before, the reason for using an automaton is to group Earley items that appear in sets together into a single entity. Unfortunately, as discussed above, the kernel and non-kernel items may originate from different Earley sets. As a result, it is not possible to have an Earley item in the form of $[s, m]$ with the state s that is not splitted. This form is, however, possible when the states of the automaton are splitted into the kernel and non-kernel states. The corollary of splitting the states is that each kernel state has up to one corresponding non-kernel state while a non-kernel state is accessible from at least one kernel state. When a state is splitted, an epsilon transition is added from the kernel state to the non-kernel state. An example of the AH automaton can be seen in Figure 16 where the states 2 and 4 are non-kernel states.

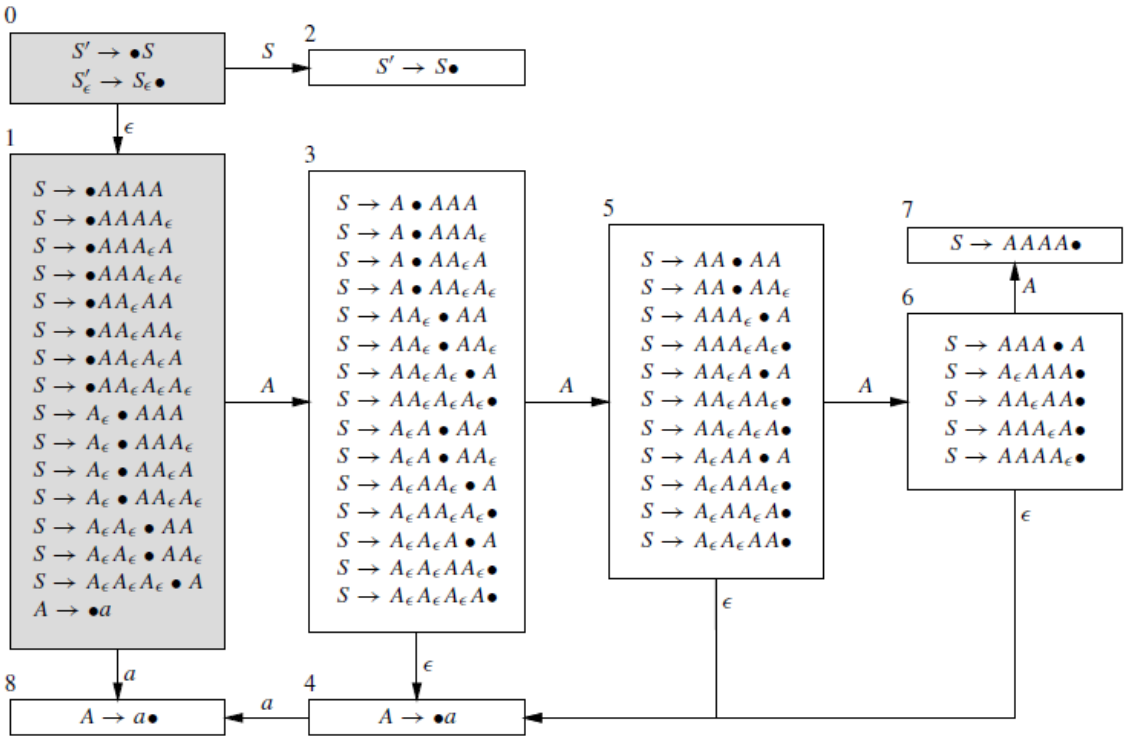

Figure 16: Aycock-Horspool automaton for NNF grammar from Figure 15. The reason for the states 0 and 1 being shaded is that state 0 is the initial kernel state while state 1 is a non-kernel state reachable from the initial kernel state. Thus, they can be viewed as the original non-splitted initial state. (source: [29]*)*

While algorithm for the computation of LR(0) states is rather straightforward and well-known [2, Section 4.6], the case of AH automaton is a bit more complicated due to its state-splitting nature. The core parts of the algorithm can be seen in Figure 17; the full algorithm is provided in Appendix A. Basically, the algorithm uses a queue to hold states which need their GOTO[2] table to be initialized. This is done as part of the procedure `ProcessState()`. When the GOTO records are generated, any state that has not been seen so far is added to the queue. Subsequently, if the state that is being processed has a corresponding non-kernel, a transition on empty symbol is added. The last thing to note is that if the non-kernel state has not been encountered before, it is added to the queue in the procedure `ComputeNonKernelState()`.

2 The GOTO can be also understood as a function which takes a state and a symbol as its arguments and returns a state which is reached from the input state on the given symbol.

```
InitializeAutomaton(){
  initialState = {[S'→•S]}
  if S'ε→ Sε is in the NNF grammar, add it to the initial state;
  add initialState to KERNEL_STATES and enqueue it;
  while UNPROCESSED_STATES_QUEUE is not empty {
    ProcessState(Dequeue(UNPROCESSED_STATES_QUEUE));
  }
  mark state with item [S'→S•] as final;
}
ProcessState(originalState){
  transitionSymbols = ExtractTransitionSymbols(originalState);
  foreach(symbol in transitionSymbols) {
    newState=GetItemsWithDotAdvancedOverSymbol(
                                      originalState,symbol);
    if newState is not empty {
      AddGoto(originalState, symbol, newState);
      if newState is not in KERNEL_STATES {
        Enqueue(UNPROCESSED_STATES_QUEUE, newState);
        Add(KERNEL_STATES, newState);
      } else {
        newState = newState from KERNEL_STATES;
      }
      AddGoto(originalState, symbol, newState);
    }
  }

  if originalState is not in NON_KERNEL_STATES {
    nonKernelState = ComputeNonKernelState(originalState);
    if nonKernelState is not empty {
      AddGoto(originalState,EMPTY_SYMBOL, nonKernelState);
    }
  }
}
```

Figure 17: The core parts of algorithm for AH automaton creation in pseudocode.

At this point, it makes sense to investigate how the automaton is used to create the Earley sets. Since the final form of algorithm will need to keep some additional information to facilitate parse tree construction, only an altered version of Earley

operations will be described. Suppose the input has a form of $x_1 \ldots x_n$ and the resulting sequence of Earley sets is $S_0 \ldots S_n$. Earley set S_{i+1} is initialised from completed Earley set S_i by following operation:

- *Scanner*: For each [*s, i*] in S_i where *s* has a transition to state *p* on x_{i+1}, add [*p, i*] to S_{i+1}.

The creation of the set S_{i+1} is then finished by applying the following two operations until no new item is added:

- *Predictor*: If item [*s, j*] is in S_{i+1}, and *s* has epsilon transition to state *p*, add item [*p, i+1*] to the set.
- *Completer*: Let [*s, j*] be an Earley item in S_{i+1}, If an AH item [$B \rightarrow \alpha\bullet$] is completed in the state *s*, add [*p, k*] for all items [*q, k*] in S_j where *p* is a state reached from state *q* on non-terminal *B*.

The actual pseudocode that implements these steps can be found in [29].

4.1.4 Parse Tree Construction

As mentioned in Section 3.8, the parser needs to retain some additional information about the progress of the parse in order to be able to efficiently reconstruct the parse tree. The approach discussed in the aforementioned section involved *predecessor* and *causal* links to record [30, p. 550] *"why an item was added to an Earley set, and which item (if any) it was begotten from."*

Since the modified version of algorithm uses the states of the AH automaton instead of the dotted productions, new definition [29] of the links is necessary:

- *Predecessor link*: Let *s* and s_{next} be states with transition from *s* to s_{next}. If an item [s_{next}, *p*] is added to Earley set due to item [*s*, *p*] from some previous Earley set, then a predecessor link from [s_{next}, *p*] to [*s*, *p*] is added to [s_{next}, *p*]. These links track origin of an item across the Earley sets. Consequently, they are created whenever the scanner or completer operations take place.
- *Causal link*: These links provide additional context to the origin of an item which has a predecessor link attached to it. Rather than tracking the origin across the Earley sets, this type of link focuses on the origin within an Earley set. Formally, if a predecessor link from [s_{next}, *p*] to [*s*, *p*] is added due to transition on a nonterminal symbol which is completed in state *q* of item [*q*, *k*], a causal link from [s_{next}, *p*] to [*q*, *k*] is attached to the predecessor link. If the transition occurs on terminal symbol, the causal link is recorded being absent (Λ).

Investigation of Figure 18, which shows Earley sets and links created when parsing input *ab*, will provide some useful insights about the reconstruction technique. It does not

matter that the Earley items in the figure do not employ the automaton state since the technique and the semantics of links is quite similar in both cases. First thing to note is that the reconstruction starts from the final item [*S′→S•*, *0*]. What this item suggests is that the root of the tree will hold non-terminal *S′* and its rightmost child node will hold non-terminal *S*. Unfortunately, it is not possible to learn more about the tree without considering the item which led to the dot being advanced over the non-terminal *S*. Luckily, there is a casual link attached to item [*S′→S•*, *0*] which points to [*S→ab•*, *0*]. The next symbol to deal with is the terminal *a*. What it means is that a leaf node has been reached, i.e. it is not possible to go further down to reconstruct the tree. What can be done now is to use the predecessor link to identify item from which the current item [*S→ab•*, *0*] originated. Following the link, item [*S→a•b*, *0*] is reached. Since the next symbol is a non-terminal again, another leaf node is created and the predecessor link is followed again to item [*S→•ab*, *0*]. This item has the dot at the beginning which means that the subtree *S* has been fully reconstructed. Now it is possible to return back to the item [*S′→S•*, *0*] and follow its predecessor link to [*S′→•S*, *0*]. Obviously, there is nothing more to be done here and the tree is reconstructed. What may not be immediately clear is that the technique tries to reconstruct the tree by deriving the right-most derivation of the item it gets as input. Another lesson to be learnt is that causal links serve as a mean for going down the tree during the construction and predecessor links move the construction in "horizontal" direction.

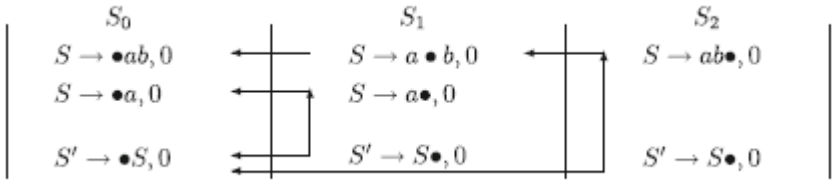

*Figure 18: Earley sets with links generated on input **ab**. The vertical arrows represent causal links while the horizontal arrows mark predecessor links. The grammar is quite simple as is evident from the content of the first set. (source: [30])*

The actual pseudocode for the automaton-based version is shown in Figure 19. The algorithm takes a non-terminal and an Earley item as input and conceptually follows the process described above. Since the NNF grammar is used, there is a different handling for nullable non-terminals: when such non-terminal is encountered, its corresponding subtree is constructed by a special method that does not use the predecessor or causal link

information. Another minor difference is that the use of states necessitates to extract the actual dotted production by combining the nonterminal given as input and the list of productions completed in given state.

What has not been discussed yet is the situation in which there are more parse trees possible for given input. In that case, there may be more completed productions and predecessor and causal links to choose from. Therefore, the algorithm may need some disambiguation rules implemented to pick the completed production or next predecessor/causal link pair.

```
def rparse(A, item):
    (state, parent) ← item
    choose A → X1X2...Xn in completed(state)
    output A → X1X2...Xn
    for sym in Xn, Xn-1, ..., X1:
        if sym is ϵ-non-terminal:
            derive-ϵ(sym)
        else if sym is terminal:
            item ← predecessor(item, Λ)
        else:
            item_why ← causal(item)
            rparse(sym, item_why)
            item ← predecessor(item, item_why)
```

Figure 19: Parse tree reconstruction algorithm. If the grammar is ambiguous, the greyed out parts locate the places where it is possible to place disambiguation mechanism. (source: [29])

4.1.5 Pseudocode for Earley Set Creation

With all other aspects of Earley parsing covered, it is now possible to discuss the actual algorithm for Earley set construction. The pseudocode can be seen in Figure 20. The first loop in `GetNextSet()` procedure acts as the scanner operation; it merely initialises the new set with items from the previous set which have a transition for terminal on the input. After that comes the completer operation. The operation from the theoretical description that seems to be missing is the predictor. Closer inspection reveals that this operation is present, but it has been integrated into the parts that handle the other two operations. Every time an item is added to the new set, its state is inspected for epsilon transition. If the transition is present, the corresponding predicted Earley item is created in the new set.

The next thing to discuss are the links. Each item may have associated collection of tuples which represents pairs of causal and predecessor links. The tuple holds the items that are pointed to by the links. The first element of the tuple is the item the predecessor links

points to while the second element is the target of the causal link. When the causal link is not present, it is marked with symbol *Λ*. Lastly, the tuples are always associated with the item where the links start so that it is not necessary to record this information explicitly in the tuple.

```
GetNextSet(originalSet, token) {
  newSet = empty Earley set;
  foreach([s, p] in originalSet) {
    if (state s has a transition on given token to state q) {
      add [q, p] to newSet;
      add link ([s, p], Λ) to [q, p];
      if (state s has a transition to non-kernel state nk) {
        add [nk, newSet] to newSet;
      }
    }
  }
  Complete(newSet);
  return newSet;
}

Complete(newSet) {
  foreach([s, parrentSet] in newSet) {
    //predicted items do not contain completed productions
    //so it is not necessary to consider them
    if (parrentSet == newSet) continue;

    foreach(nonterminal B completed in state s) {
      foreach([q, r] in parrentSet) {
        if (state q has a transition on B to state t) {
          add [t, r] to newSet;
          add link ([q, r], [s, parrentSet]) to [t, r];
          if (nonkernel state nk is reachable from state q) {
            add [nk, r] to newSet;
          }
        }
      }
    }
  }
}
```

Figure 20: Pseudocode for Earley set construction.

What is also important to note is the use of the for loops throughout the procedures. The theoretical description states that the set is created once no new item is added by the last application of the operations. Obviously, the for loop in the `GetNextSet()` procedure merely initialises the new set with suitable items from the previous set and with items predicted from the aforementioned items. The use of for loop in the `Complete()` procedure, on the other hand, is somewhat evasive. The Earley set used in the pseudocode can be viewed as a hybrid between a list and a set: the items are ordered and whenever a new item is added, it is appended behind the last item provided that the new item is not in the set already. Therefore, the for loop basically iterates through a list whose number of items may be increased during the iteration. What it also indicates is that an item is fully processed during a single iteration. Why is this possible? Firstly, it should be noted that predicted items are used only in the subsequent sets: once they are added to the current set, it is impossible to use the predictor operation on them as the state-based nature of items ensures that they already contain all productions predicted from the source item. Moreover, the only way for a predicted item to be used in completer operation is to hold an AH item with nullable left-hand side A_ε of a production. This is not possible since these items are excluded from the AH automaton states. Secondly, once an item is added into the current set, if its corresponding predicted item exists, it is added as well. As a result, an item from the current set can only contribute to the set creation if its state contains completed productions that can be used to add more items into the set.

4.2 Design

The aim of this section is to provide a description of the project implemented as part of the thesis. The name that was chosen for the project is *Earley Parser Framework (EPF)*.

Conceptually, the software implementation of the parsing process can be broken into several parts. As discussed before, there may be a phase of parsing called lexical analysis that transforms the source text into a sequence of tokens. This functionality is implemented by a lexer or a scanner. Naturally, there is also the parser which does the actual parsing. In order to specify what language is expected to be parsed, a part that corresponds to the grammar has to be present as well. All these parts as well as some others can be found in high-level architecture overview of the project shown in Figure 21. The rectangular objects represent modules, while the full arrows represent input or output information. Finally, the dotted arrow represents that the target uses the source object in some way.

From the high-level perspective, the source text is transformed with lexer into a sequence of tokens. This step is executed at once; i.e. the whole source text is transformed

before the actual parsing starts. The parser uses external information from the grammar and error handler module to carry out the parsing. If it succeeds, a parse tree is created.

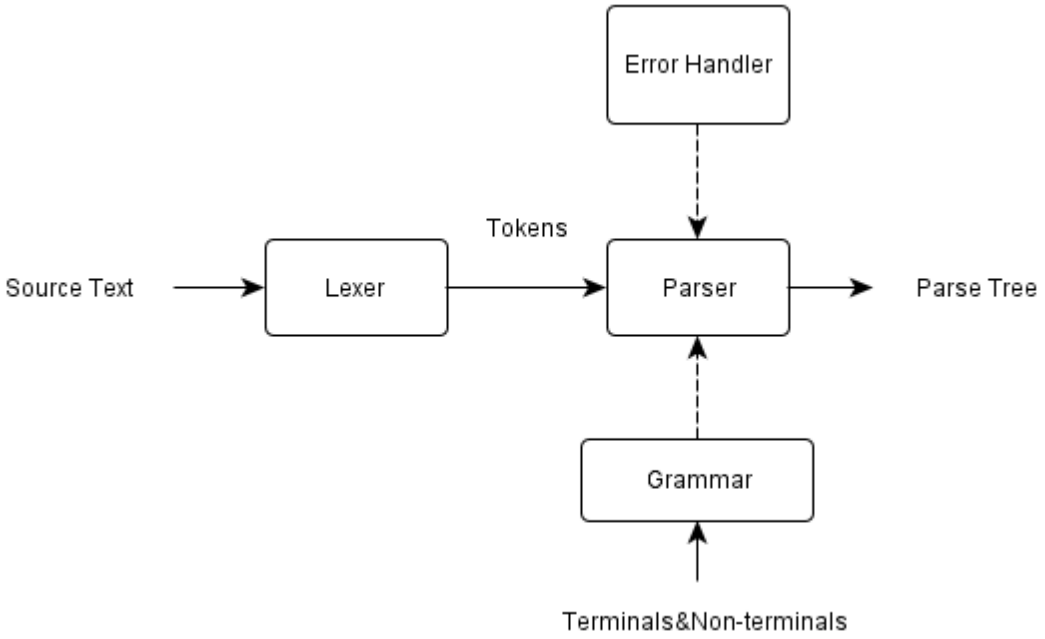

Figure 21: High-level scheme of the project.

It should be noted that the overall design of the program has been greatly inspired by project *Irony - .NET Language Implementation Kit* [31] of Roman Ivantsov. In fact, we considered forking the project to use its matured support infrastructure. Unfortunately, Irony uses LALR(1) algorithm and its use affects parts of the program which do not directly implement the algorithm such as lexer and grammar. It is our opinion that it would be rather difficult to embed the Earley algorithm into the existing architecture. Therefore, we have just used some elements of the Irony design.

4.2.1 Grammar

The way the grammar is entered is expected to use C # language constructs. The approach used by EPF is shown in Figure 22. The usage is quite obvious: a non-terminal is used to represent itself (in a body of a production) as well as to define all productions which have it on the left-hand side. The terminals can be defined either explicitly, such as the terminal `num`, or it can be defined implicitly as a string in the definition of a production. The important thing to note is that the implicit use is impossible if there are more consecutive

implicit terminals at the beginning of an alternate[3]. In such case, the C# operator + would be applied a concatenation of two strings. Therefore, the best practice is to use always use explicit terminal as a first symbol of an alternate. The last thing to note is that this approach was taken from the Irony parser.

```
class ExpressionGrammar : Grammar
{
  public ExpressionGrammar()
  {
    NonTerminal F = new NonTerminal("F");
    NonTerminal E = new NonTerminal("E");
    NonTerminal T = new NonTerminal("T");

    RegexTerminal num = new RegexTerminal("number",
                                          "[0-9]+",
                                          TerminalPriority.Nominal);

    E.Rule = E + "+" + T |
             E + "-" + T |
             T;
    T.Rule = T + "*" + F |
             T + "/" + F |
             F;
    F.Rule = new Keyword("(") + E + ")" | num;
    this.Root = E;
  }
}
```

Figure 22: A simple expression grammar.

How is the behaviour described above achieved? Further insights about the design of grammar input part may be gained from Figure 23. The abstract class `GrammarSymbol` defines + and | operators in a way which ensures that that if a string is encountered, it is automatically transformed into a terminal. An example of this definition is the following method:

3 When a production is represented as $A \rightarrow \beta \mid \gamma$, the β and γ can be called alternates.

```
public static GrammarTerm operator |(GrammarSymbol symbol1,
                                      string symbol2)
{
  return AlternateOperation(symbol1, new Keyword(symbol2));
}
```

As can be guessed from the definition, class `GrammarTerm` is also responsible for the functionality that allows definition of the productions. This class, basically, acts as a list of alternates. An alternate itself can be viewed as a list of grammar symbols so we are dealing with a list of lists of grammar symbols. The implementation of operators in `GrammarSymbol` class uses this feature in the way that when the whole expression is evaluated, the final `GrammarTerm` object holds a representation of all alternates encountered during the evaluation. The aforementioned object is then assigned to the `Rule` property of some nonterminal.

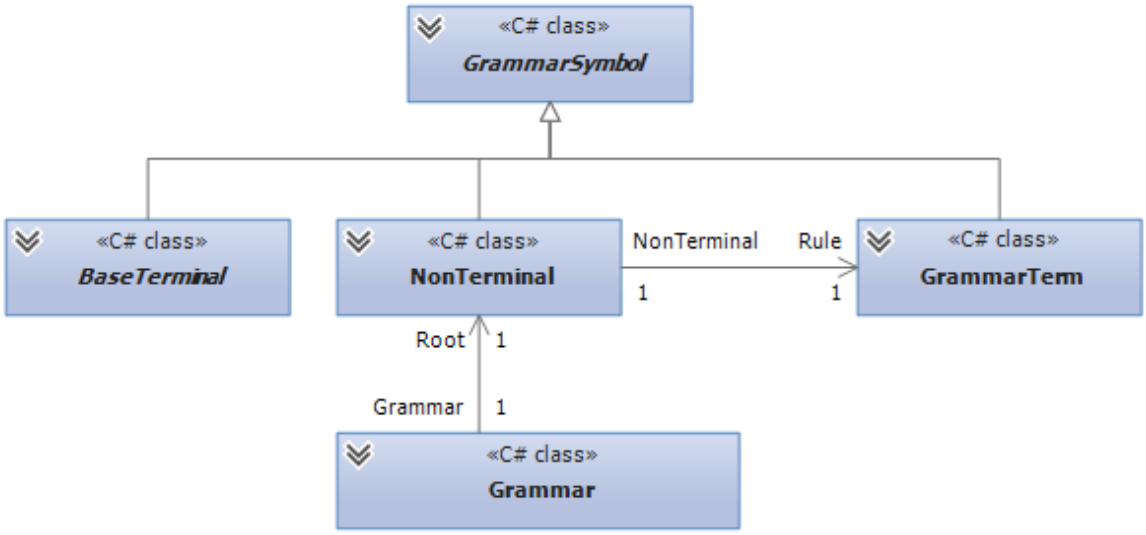

Figure 23: UML diagram of classes related to grammar definition.

There are several other issues that has to be mentioned. Firstly, before the grammar can be used, it has to be initialized by the `Init()` method. Secondly, there is a property called `IsWhitespace` which can be used to set what characters should be considered whitespaces, i.e. the characters that should be skipped by the lexer. Thirdly, if the grammar is to be used in a multi-threaded environment, the best practice is to create a single instance of it and use it across the threads. The reason for doing so is that the initialisation may be rather costly operation: as part of the process, the AH automaton is created and attached to the grammar. While the space and time complexity of the construction may not be that problematic on modern machines, it is unnecessary waste of computing resources: especially since the grammar and the automaton ought to be thread-safe. What is also important to note at this point is that the transformation into NNF creates copies of nullable non-terminals. Such copies have the original name prefixed with string "___NULLABLE_" and their `IsNullable` property is set to `true`. Lastly, the static property `Empty` holds an instance of terminal class `EmptyTerminal` which serves as the epsilon symbol in standard terminal notation.

4.2.2 Lexer

Lexer has two modes of operations: it can either transform the whole source text into a sequence of tokens or the transformation may be limited only up to some position in the text. The reason why it is possible to specify the position is that when the parser is

expected to provide suggestion of what is expected to appear next, it is not necessary to transform the whole text.

The inner working of the lexer is rather simple. Until the stop condition is met, it tries to read a token or skip whitespace. If no token can be reconised at the current position in the source text, the characters in the text are skipped until recognition is possible.

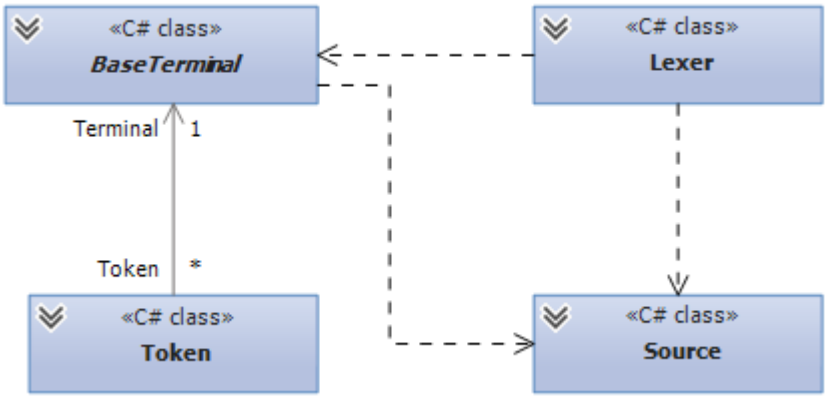

Figure 24: Conceptual diagram of the lexer.

The conceptual overview of lexer infrastructure can be seen in Figure 24. Class `Source` provides an abstraction over the source text: among other things, it holds current position in the source text. One of the classes crucial to the parsing is the abstract class `BaseTerminal` whose details are shown in Figure 25. When the `Lexer` class attempts to read a token, it tries to match all known terminals to the text at the current position in the source using their `Match()` methods[4]. The order in which the terminals are matched is determined by the value of the `Priority` property. Another part to mention is the delegate `CharacterCannotFollow` which can be used to define what characters cannot follow behind a terminal. For example, if the DSL contains an operator +, it is likely that it can be followed by almost anything: a digit, a number, etc. On the other hand, if it contains the word *while*, it may not make sense to recognise the token on string *whilec* as it may be poorly named identifier. The last property to mention is the `OnTokenRecognised` property which allows the user to attach a method to a terminal which is executed if the corresponding token is recognised. This behaviour does not have to be turned on explicitly; as soon as there is a terminal with the action set, the lexer executes it automatically when the token is recognised. The two undiscussed properties `ErrorValue` and `CanBeIncomplete` will be covered in section that deals with error recovery and suggestion generation.

4 This is approach was inspired by the one the Irony project uses.

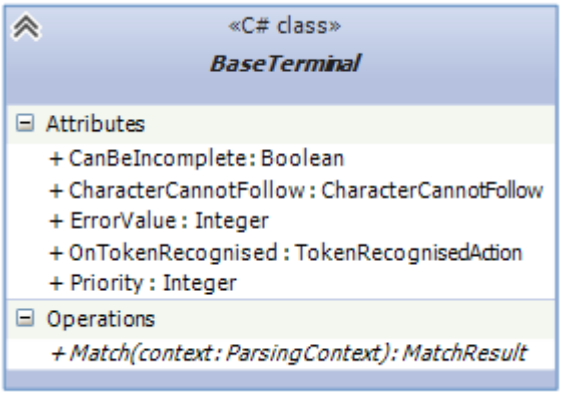

Figure 25: Details of terminal implementation.

There are two subclasses of the base terminal class called `Keyword` and `RegexTerminal`. The former may be used to represent a keyword or an operator; basically, it is a terminal with fixed number of characters. This class is used as a default choice for strings entered as terminals in the grammar definition. By default, it can be followed by any character. The other class uses a regular expression to define the terminal. Use of both types can be seen in Figure 22.

4.2.3 Parsing Context

As can be seen in Figure 25, the method `Match()` takes an instance of class `ParsingContext` as its argument. This class serve as a repository for important information about the parsing process; this design concept has been inspired by the Irony project. The members of the class are shown in Figure 26. There are several notable members. Firstly, there are members that correspond to the output and input token sequence. The error recovery process might alter the input token sequence so the purpose of the output token sequence is to be able to discern what was the input and what was the output token sequence. Secondly, it holds the list of all Errors encountered during a single parse. This is especially important if the user decides to implement custom error recovery strategy. Another important property to mention is the `CustomData` property which allows to user to store important additional information that can be used in custom error recovery. Lastly, the context also holds the Earley sets created so far in the property `Parse`.

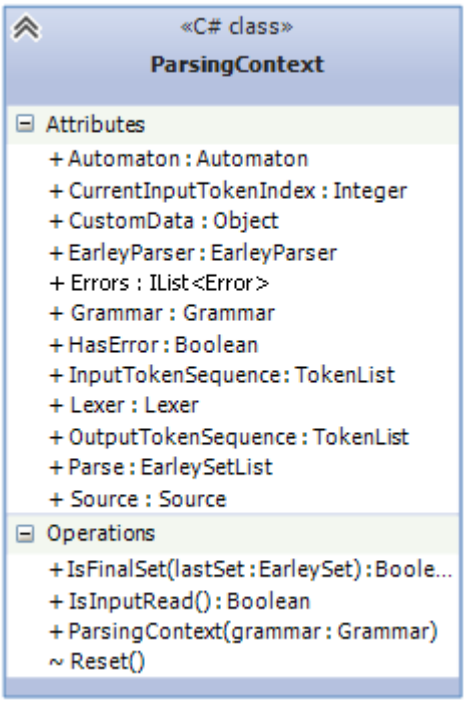

Figure 26: Details of class ParsingContext.

4.2.4 Earley Parser

There are three main features expected from the parser: firstly, it has to be able to parse a given string. Secondly, it should provide information that would allow to suggest what is expected to appear next in the test. Lastly, it has to be able to recover from minor errors encountered when parsing or generating suggestion information; this functionality is discussed in separate section. The first two uses cases are shown in Figure 28. Most of the code is quite obvious; the only exception might be the use of `MakeSuggestion()` method. If it is not supplied the position where the suggested text should start, the suggestion is generated for the position at the end of the string. Figure 27 shows how the value of position for the suggestion can be determined.

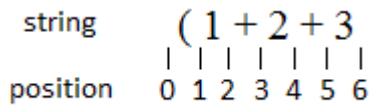

Figure 27: An example of position index.

The details of what information the suggestion and parsing methods return can be seen in Figure 29. Members of both classes are rather self-explanatory. The only one property to discuss at this point is the `IncompleteToken` property found in `SuggestionInformation` class as it is closely related to one important feature of the suggestion process. If the token can be incomplete[5] and it borders with the position where the suggestion should be made, the parsing process does not consider this token when creating the Earley sets. It is, however, included in the result of the suggestion.

```
var grammar = new ExpressionGrammar();
grammar.Init();

var parser = new EarleyParser(grammar);
ParseResult result = parser.Parse("1 + 2");
SuggestionInformation suggestion1 = parser.MakeSuggestion("1+3+");
var suggestion2 = parser.MakeSuggestion("1+23+", 2);
```

Figure 28: Basic use of the Earley parser. When the second suggestion is generated, only string "1+" is considered.

Each non-terminal may have an associated action which is executed whenever the non-terminal is recognised during the parsing or suggestion-making process. The property of non-terminal class that is responsible for this functionality is `NonterminalRecognizedAction` which holds the reference to the action that should be executed. This functionality has to be enabled explicitly in the property `NonterminalActionsEnabled` of the parser. The last thing to note is that the execution of the actions may be influenced by the error recovery process. For example, if the recovery were to the change some of the tokens that have been already parsed, it might be necessary to revoke the effect of actions that were executed due to these tokens.

5 This is determined by the value of `CanBeIncomplete` property of its corresponding terminal.

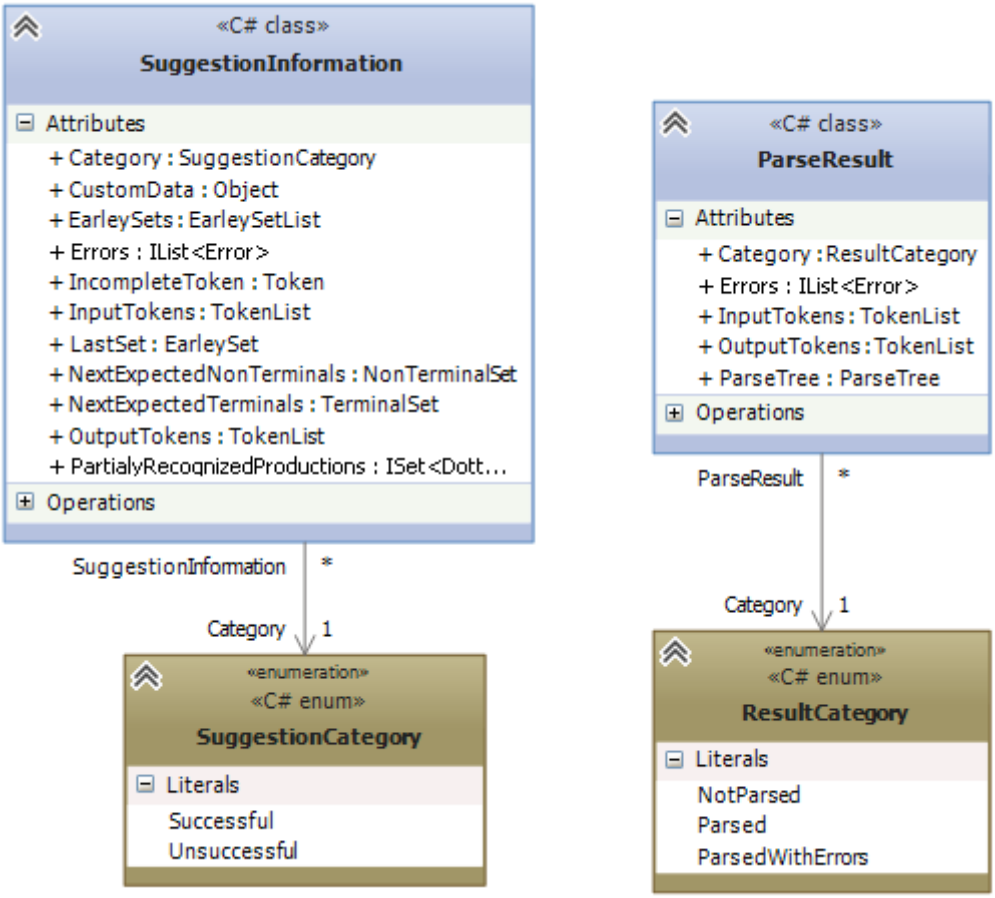

Figure 29: Classes that represent a result of the parsing and suggestion.

The parse tree that is created by the parser is a typical tree structure. Class `ParseTree` is not very interesting as its single purpose is to hold the root of the tree. We will focus on the class `ParseTreeNode` shown in Figure 30 as it is more interesting than the actual class `ParseTree` whose single purpose is to hold the root node of the tree. Basically, a parse tree node holds either a token or a non-terminal which depends on whether it is a leaf node or not. The node implements the Visitor/Visitable pattern as is evident from the method `AcceptVisitor()`. Probably the most important method is the static `CreateSubtree()` method which implements the tree construction algorithm from the Section 4.1.4.

The inner working of the `Parse()` and `MakeSuggestion()` methods relies heavily on the `EarleySet` class. The `GetNextSet()` method of this class implements the algorithm shown in Figure 20. The instance on which the method is called acts as the input

set from the original algorithm. This allows the parsing methods to keep calling the `GetNextSet()` method with the current token as the input. If the returned set is empty, it means there is an error and the parser has to react to it.

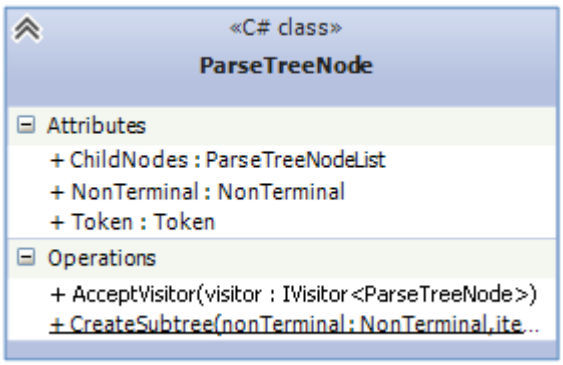

Figure 30: Details of parse tree node implementation.

4.2.5 Error Handling

There are two aspects of error handling: error recovery and error reporting. While there are many strategies for both to handle the both [3, Chapter 16], it is rather difficult to pick one that would be the best to use in every single scenario. As a result, the EPF allows the users to supply their own recovery and reporting strategies. There are also two recovery strategies provided by default which are based on the FMQ [3, Section 16.6.4] and Burke-Fisher [32] error recovery techniques.

There are three possible behaviours of the parser for the error recovery and reporting; all of them are described in Table 1. The parser uses by default the modified version of Burke-Fisher recovery technique. The default error reporting scheme is quite simple: if the parsing fails because of unexpected token, it is reported that an unexpected token has been encountered. If all the tokens are processed, but the last set does not contain an item with final state of the AH automaton, an unexpected end of file is reported.

Table 1: Possible behaviours of error reporting and recovery in the parser.

Behaviour	**Way to achieve it**
no recovery with default error reporting	Create the `EarleyParser` object with `noRecovery` argument set to true.
default recovery and error reporting	Create the `EarleyParser` object.

custom recovery and reporting	Create the `EarleyParser` object with the constructor that takes an instance of class `ErrorHandler` as argument.

Figure 31 provides an insight into the error handling mechanism of the parser. Class `ErrorHandler` contains a single error reporting strategy and may hold multiple error correcting strategies. What should be pointed out is that it may make sense for the error recovery method to do the error reporting as well based on the found recovery. The way the general error-handling mechanism works is following:

1. Start trying error recovery strategies in order in which they appear in the list.
2. If a strategy succeeds, check whether it also reports errors.
    a. If the errors are reported as part of the strategy, the error handling is finished.
    b. If the errors are not reported, try to use the reporting strategy defined in the error handler to report an error.
        i. If the reporting strategy succeeds, the error handling is finished.
        ii. If the strategy fails, use the default error reporting strategy.
3. If no recovery strategy succeeds, try to use the error reporting strategy: the approach is the same as discussed in 2.b..

There is one more feature connected to error recovery in general. Each terminal has a property `ErrorValue` which can be used in error recovery strategy to hold some integer value. Moreover, each grammar symbol may have a tag attached to it. The tag consists of a name and an object. This can be used in error recovery as well.

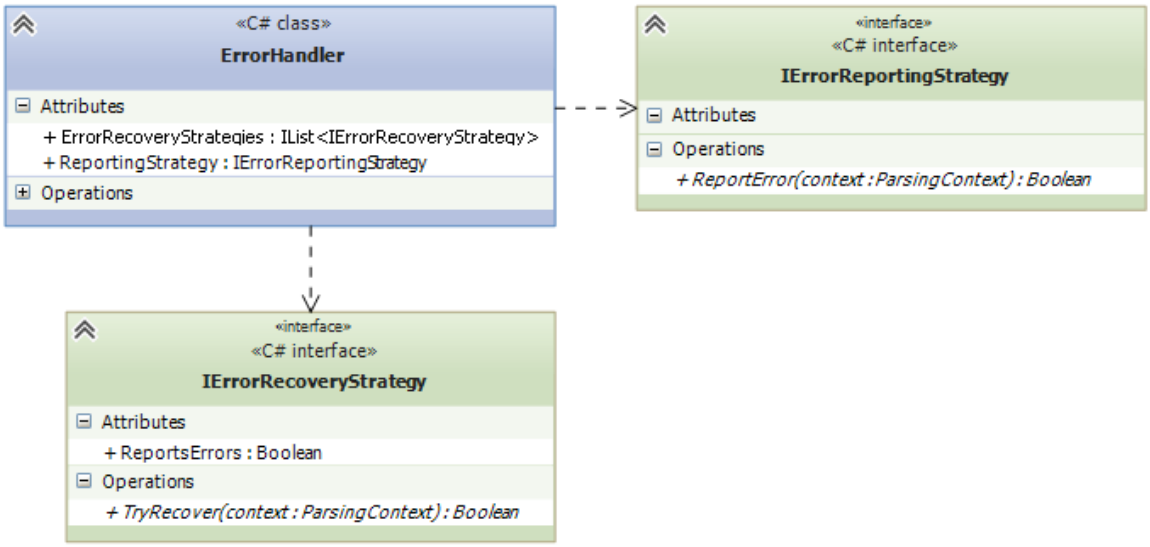

Figure 31: ErrorHandler class details.

4.2.5.1 Insertion-Only Error Recovery

This technique is based on the FMQ error recovery which, basically, inserts one or more artificial tokens before the token on which the parsing stops so that the parsing can continue. Each token has some sort of cost attached to it so it is possible to compute and pick the cheapest correction. The natural question to ask is whether every language can be corrected this way. Unfortunately, there are languages [3, p. 538] which cannot be corrected this way.

The method provided by the EFP is basically a form of weighted graph search on the space of all possible insertions. Each node in this space has associated Earley set, a weight and a reference to the previous set. The weight is a sum of all error values on the inserted terminals. Thus, each node represents a correction candidate and its cost. Naturally, the correction has to be valid, i.e. the token on which the error occurred has to be accepted if the correction is inserted before it. Basically, when some node is reached it can be either valid or invalid. If the node is valid, all other nodes derived from the current one will have their cost higher than the current one, so it is not necessary to consider them. If the node is invalid it can be used to generate more nodes. The way to achieve that the node with lowest cost is picked is to add all the generated nodes into a priority queue. We will always pick the node with the lowest cost from the queue and check if it is valid. When it is valid, we have a valid correction. If it is not, then the node is used to generate more nodes which are subsequently added into the queue.

There are two parameters that affect the method. As mentioned above, some languages may not be correctable by this procedure. Therefore, there is a limit for maximal length of

the inserted token sequence. To further confirm the validity of the inserted tokens, the recovery process may try to use a given number of input tokens after the error position to check that the recovery was successful. The code for setting insertion-only recovery method is shown in Figure 32.

```
var handler = new ErrorHandler(
                    InsertionOnlyErrorRecovery
                      .Create(maxRepairSteps:3,
                              validationSteps: 1));

var parser = new EarleyParser(grammar, handler);
```

Figure 32: Use of insertion-only recovery.

4.2.5.2 Modified Burke-Fisher Error Recovery

Sometimes when parsing cannot continue it is not due to the token on which it failed. The cause can be also a token that has been successfully parsed before. To be able to address the issue, Burke-Fisher algorithm [32] keeps a window of the last successfully parsed *k* tokens. When an error is encountered, the recovery procedure tries every possible single token insertion, deletion and substitution in the window. The correction is validated if the parsing can continue *m* tokens past the point where the error was encountered. The original version of the algorithm associates the operations and tokens with some cost and after all possible valid corrections are considered, it picks the one with the lowest cost.

The version of the technique that the EPF uses does not consider all the possible corrections. It starts the operations from the position where the error occurred and continues to the other side of the window until it finds a valid correction. This correction is then used for the recovery. The last thing to note is that the terminals in insertion and substitution operations use the error value of terminal as a priority: the higher priority, the sooner the terminal is tried.

Users can affect the technique in two ways: firstly, they can set the size of the window. Secondly, they can specify how many successful steps are needed to consider a correction valid.

4.2.6 Abstract Syntax Tree

The final part of the EFP to discuss is its facility to transform the parse tree into the abstract syntax tree. There is a general mechanism that should allow this functionality.

Each grammar symbol has a property `CreateAstNodeMethod`. It is a delegate with the following signature:

```
object CreateAstNodeMethod(ParseTreeNode ptNode,
                           object context = null).
```

Thus, each parse tree node can call the method with itself as an argument. This is a very generic approach so the users can customize the AST creation to match their needs. However, it is rather inconvenient to have to create the whole new infrastructure even for very simple languages. To address the issue, a class `AstNode` and some supporting methods are provided. The class is shown in Figure 33.

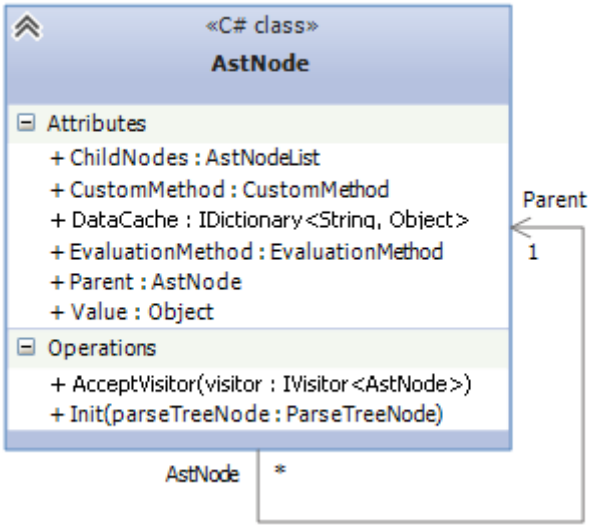

Figure 33: Details of AstNode class.

There are different delegates that are used by the class. Probably the most important is the `EvaluationMethod` delegate. As the name suggests, it allows the user to supply the method that is used for the evaluation of the AST node. The purpose of `CustomMethod` delegate is quite similar; it can be used for some additional functionality that for some reason cannot be a part of evaluation. For example, it could be used for type-checking.

On their own, these delegates do not provide the functionality to create the AST. The mechanism that allows AST creation is implemented by the methods `MakeCreate()` and `MakeCreate<T>()` from the `AstUtilLibrary` class which have the following signatures:

```
public static CreateAstNodeMethod MakeCreate(
                EvaluationMethod evaluationMethod,
                CustomInitialization customInit = null,
                CustomMethod customFunc = null)

public static CreateAstNodeMethod MakeCreate<T>()
                                    where T : AstNode, new()
```

The delegate returned by the first method creates an AST node with the delegates set to those specified as the method's parameters. The delegate `CustomInitialization` can be used to customize the node when it is created. The second method is for the situations in which the AST functionality is defined as part of a subclass of `AstNode` class. It should be noted that in this case the `Init()` method is automatically called when the node is created.

4.3 MotiveQuery Grammar

MotiveQuery is *"a language for describing and analyzing protein motives"* created by David Sehnal. The problem domain of the language will not be discussed as it is outside of scope of this thesis. Detailed description of the MotiveQuery can be found in [33]. Syntactically, the language uses a restricted subset of Python programming language features. As can be seen from the grammar definition in Figure 34, the features include function calls, class members calls, lambda definitions, arithmetic, logical and relational expressions, lists etc. The reason why there is no functionality for class or function definitions is that these are predefined by the MotiveQuery language. It should be clear from the definition that the language is expected to be type-checked, although the identifiers are predefined and new types cannot be entered by the users as they cannot cannot create new classes or functions.

When a language is type-checked, the grammar does not specify all possible names of functions or classes. It rather provides a general definition of a function by stating that it is a sequence of characters followed by brackets between which a list of arguments can be included. This complicates the suggestion generation as it is no more simply the matter of stating what token can appear next. In this situation, it is necessary to have at least a part of parse tree on which the type-check could be made. Obtaining a partial parse tree is rather difficult due to the way the Earley parsing works. However, this does not exclude possibility of a successful suggestion generation.

In order to determine what information is necessary for the successful creation of a suggestion, it is necessary to consider what suggestions are expected in given situation.

The first thing to note is that if we know that some type is expected at some point in the input, it is not enough to list only functions that have the same return type. The reason for that is that member method calls on return types which are not related to the expected type may result in the type that is expected. For example, this is completely legal call of a hypothetical method that computes a square root of given number:

```
SquareRoot(GetSomeList().Length()).
```

Obviously, if a suggestion is required for the argument of the method, the `GetSomeList()` method should not be excluded from the suggestions as it may yield the expected type later on through its member methods calls. On the other hand, it is sensible to require that the methods that return a number are listed before the `GetSomeList()` method in the suggestion. Table 2 the describes various situations and what behaviour is expected from the suggestions.

```
expr.Rule = orExpr | lambdaDef;
lambdaDef.Rule = lambdaHead + expr;
lambdaHead.Rule = LAMBDA + lambdaParams + ":" | LAMBDA + ":";
lambdaParams.Rule = param + "," + lambdaParams | param;
param.Rule = ID + "=" + expr | ID;
orExpr.Rule = orExpr + "|" + andExpr | andExpr;
andExpr.Rule = andExpr + "&" + notExpr | notExpr;
notExpr.Rule = "!" + notExpr | comparisonExpr;
comparisonExpr.Rule = comparisonExpr + compOp + addExpr | addExpr;
compOp.Rule = new Keyword("<=") | "<" | ">=" | ">" | "==" | "!=";
addExpr.Rule = addExpr + addOp + multExpr | multExpr;
addOp.Rule = new Keyword("+") | "-";
multExpr.Rule = multExpr + multOp + unaryExpr | unaryExpr;
multOp.Rule = new Keyword("*") | "/";
unaryExpr.Rule = addOp + unaryExpr | powerExpr;
powerExpr.Rule = memberAccess + "^" + unaryExpr | memberAccess;
memberAccess.Rule= memberAccess + subscriptStart+subscript+subscriptEnd|
                   memberAccess + "." + func |
                   memberAccess + "." + ID |
                   atom;
subscriptStart.Rule =  new Keyword("[");
subscriptEnd.Rule = new Keyword("]");
subscript.Rule = expr | optExpr + ":" + optExpr;
optExpr.Rule = Grammar.Empty | expr;
atom.Rule = INT | REAL | ID | TRUE | FALSE | STRING | CHAR |
            func | "(" + expr + ")" | listStart + listBody + listEnd;
listBody.Rule = expr + "," + listBody | expr;
listStart.Rule = new Keyword("[");
listEnd.Rule = new Keyword("]");
func.Rule = funcBegin + argList + funcEnd | funcBegin + funcEnd;
funcBegin.Rule = ID + "(";
funcEnd.Rule = new Keyword(")");
argList.Rule = arg + argSeparator + argList | arg;
argSeparator.Rule = new Keyword(",");
arg.Rule = namedArg | expr;
namedArg.Rule = namedArgHead + expr;
namedArgHead.Rule = ID + "=";
```

Figure 34: Specification of the MotiveQuery grammar. The root non-terminal is expr. Identifiers with name in upper case represent terminals.

Table 2: Expected behaviour of suggestions for MotiveQuery language constructs. Symbol | represents the position at which the suggestion is requested.

Situation	Expectation
`foo(|`	The type of the argument should be considered when making the suggestion.
`foo(...,...,|`	The previous arguments are of little importance to the suggestion which is expected at the point. The behaviour is the same as in previous situation.
`X.|`	The X can be a function, a list, etc. so its type have to be determined in order to be able to generate the suggestion.
`foo(...,...- bar(...,|`	The current suggestion is only affected by the signature of `bar` function; function `foo` is not taken into consideration.
`["aaa",|`	The MotiveQuery language uses lists to hold names of atoms, molecules, etc. which are represented as a string. As a result it would be expected that the names are included in the suggestion.
`lambda x, y: 1+|`	Lambda parameters x and y should be included in the suggestions.
`foo()[|`	The suggestions should reflect that lists are expected to have an integer in their index.
`fo|`	All applicable symbols starting with `fo` should be suggested.
arithmetic_operator|	With the exception of multiplication operator, the arithmetic operators are expected to work with numerical values. The suggestions should reflect this fact.
logical_operator|	Logical operators work with boolean type.
relational_operator|	Relational operators work primarily with numerical values.

4.3.1 Language Construct Context

The suggestion requires to have a certain context which can help to determine what symbols are admissible. There are several contexts that might be relevant to suggestion generation; they are all listed in Table 3.

Table 3: Possible contexts and their purposes.

Context	Purpose
Function	The expected type of arguments can be determined from this context.
Argument	This context is always connected to function context. It identifies the argument of the related function that should be entered at this point.
Lambda body	When a suggestion within a body of a lambda definition is needed, the parameters of the lambda may be part of it.
List	The lists are often used in MotiveQuery to hold names of atoms, molecules etc. so it makes sense to be aware that at the moment a list is being defined.
Indexer	Since the expected use of an indexer is on a list, it might make sense to suggest integer returning functions.

The most important thing to note about the contexts is that it is necessary to store information about all contexts that have been encountered so far and have not been finished yet. Figure 35 may help understand the issue. The symbol | represents how far the parse progressed and that a suggestion could be expected at that point. In the first case, the context that should be used is that of the `bar()` function. Context for `foo()` has to be stored as well as it will be necessary later on. In the second case, the function `bar()` is already finished and it cannot affect the suggestion at the current position in any way. The way to store all the opened contexts is to push a new context on stack as soon as it is encountered and pop it out when the context is no longer relevant. Thus, the current context is always on the top of the stack and all unfinished contexts are kept there as well.

```
foo(bar(...,|      →    foo(bar(...,...),|
```

Figure 35: Example of multiple contexts.

A valid question to ask is how this behaviour can be achieved with the EPF. The answer is rather simple; the main issue is to detect that a new context has been encountered. This feature required minor redefinition of the grammar. For example, when a non-terminal `funcBegin` is recognised, we know that a new context has to be pushed onto the stack which is stored in the `CustomData` property of the `ParsingContext` class.

5 Results

With the implementation described, the next step is to test its performance. Several quantities were measured that should allow to demonstrate the capabilities of the project. This chapter also tries to deal with what the implications of these tests are. Detailed results can be found in Appendix C. The number of measurements was 100 for each case. All the queries used in this chapter are described in Appendix E. All tests were performed a machine with the following parameters:

- OS: Windows 7 64bit
- CPU: Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz
- RAM: DDR3-SDRAM (1333 MHz)
- HDD: 465.8 GB, 7200 RPM, 16 MB

5.1 Parsing Performance

Probably the first question to ask is how fast the basic functionality of the parser is. In order to answer this question three grammars were tested:

- *MotiveQuery grammar*: The grammar was described in previous section
- *Expression grammar*: Grammar for a simple language of arithmetic expressions which may refer to predefined mathematical functions.
- *Ambiguous expression grammar*: This grammar describes the same language as the previous one, but unline its predecessor, it is ambiguous.

Definitions of both grammars for arithmetic expression can be found in Appendix D. The first two grammars could be considered the typical use cases for the parser. The purpose of the third grammar is to show how the parser behaves with ambiguous grammar. The tests were conducted on source texts with different number of tokens. As discussed before, the EPF is expected to work on rather short input sequences which is the reason why the maximal size of the input was 400 tokens.

Figure 36 shows how long it took to parse a token sequence of given size. The parser's speed of execution shows approximately linear dependency on the size of the input. This result confirms the theoretical expectation about Earley algorithm's behaviour on unambiguous grammars which was discussed in Section 3.8.4. The obtained results are

rather favourable as they are well beneath the limit of 100-200 ms which is considered [34] to be a response time which does not restrict the work of a user.

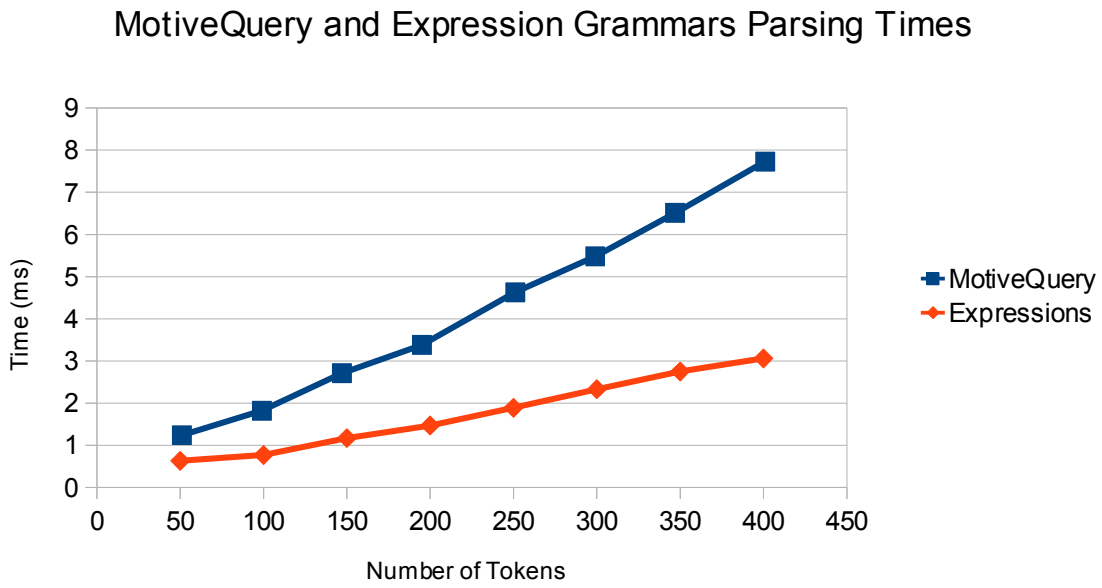

Figure 36: Measured times of MotiveQuery grammar and arithmetic expressions grammar for input of different size.

As can be seen in Figure 37, the performance on ambiguous grammars is not likely to be as good. In theory, the algorithm has cubic time complexity on ambiguous grammars. The execution time is more than 30 times slower on the largest input. The reason for that is that all information about possible parse trees are stored in the Earley sets. In order to investigate this matter, the average size of an Earley set was measured during parsing of the input with 400 tokens. Average size of set for the ambiguous grammar case was 40.4. This number is in stark contrast with the figure for unambiguous grammar which is 4.16. To conclude, it seems rather inadvisable to use ambiguous grammars.

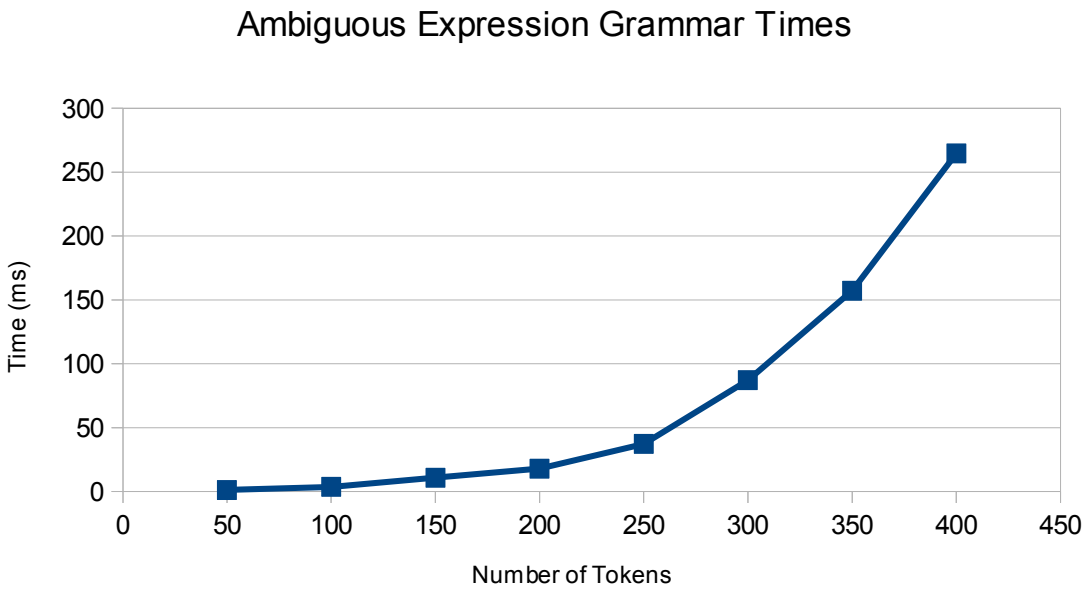

Figure 37: Parse times for ambiguous grammar for arithmetic expressions.

5.2 Performance With Error Recovery

Another batch of tests was carried out in order to determine how the error recovery methods affect the performance. Simplified Burke-Fisher recovery method was used for arithmetic expression grammar with the repair window size set to 2; the same value was used for the number of steps that conform recovery. Insertion-only technique was employed for MotiveQuery grammar with the maximum number of insertions set to 2 which was also the number of steps necessary to validate the correction. The size of the input was 400 tokens in both cases.

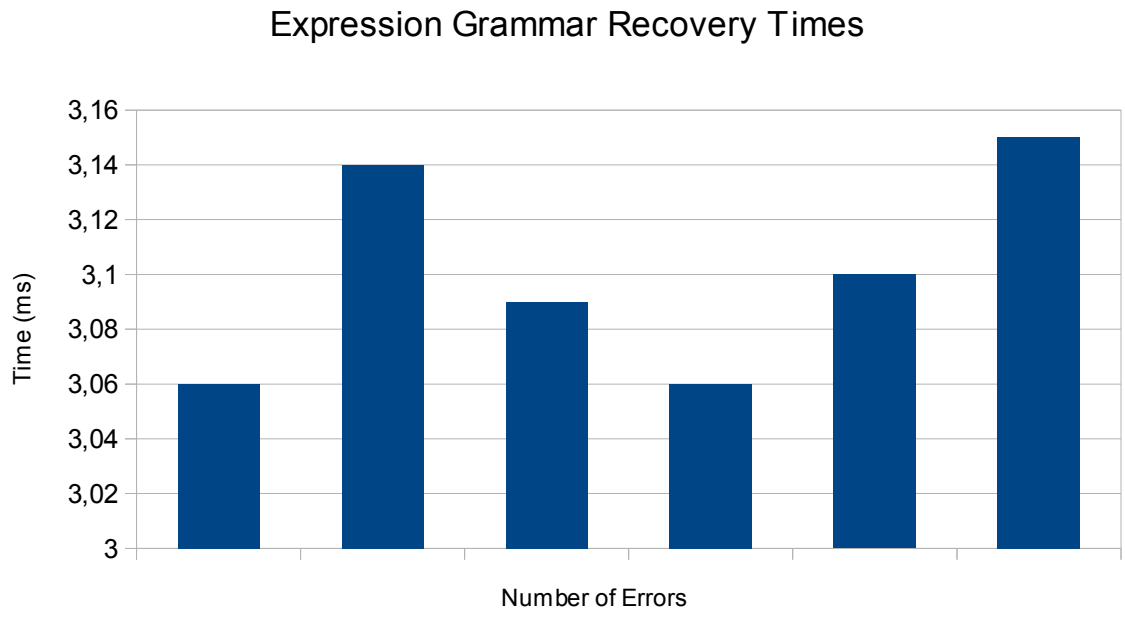

Figure 38: Parsing times of arithmetic expression input of size 400 tokens with variable number of errors.

The test results are shown in Figures 38 and 39. The data do not show any obvious trends; indeed, it seems that the results tend not to be affected by the number of errors in the input. The effect of the error recovery seems to be rather marginal.

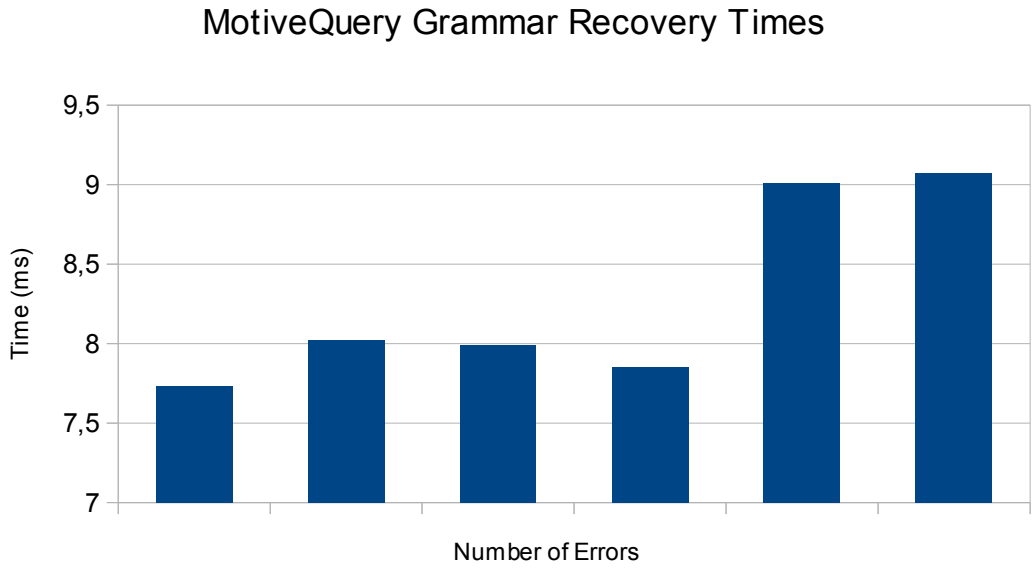

Figure 39: Parsing times of MotiveQuery expression input of size 400 tokens with variable number of errors.

5.3 Suggestion Generation Performance

A suggestion generator was created for MotiveQuery language based on the description from Section 4.3. There is one minor issue with the implementation: in situation where a suggestion is expected after a dot (`X.|`), the type-check of X is not performed. This issue should be correctable as soon as a new version of type system in which types can have member methods is available. At the moment, the situation is solved by suggesting all applicable functions in the type system. Since all situation only use the static symbol table to lookup relevant symbols for suggestions, the test was performed only for `X.|` use case.

The obtained results are shown in Figure 40 which compares the times needed to parse an expression in MotiveQuery and to generate suggestions for it. The likely reason for the normal parsing being slower than suggestion generation is that a parse tree is created during regular parsing.

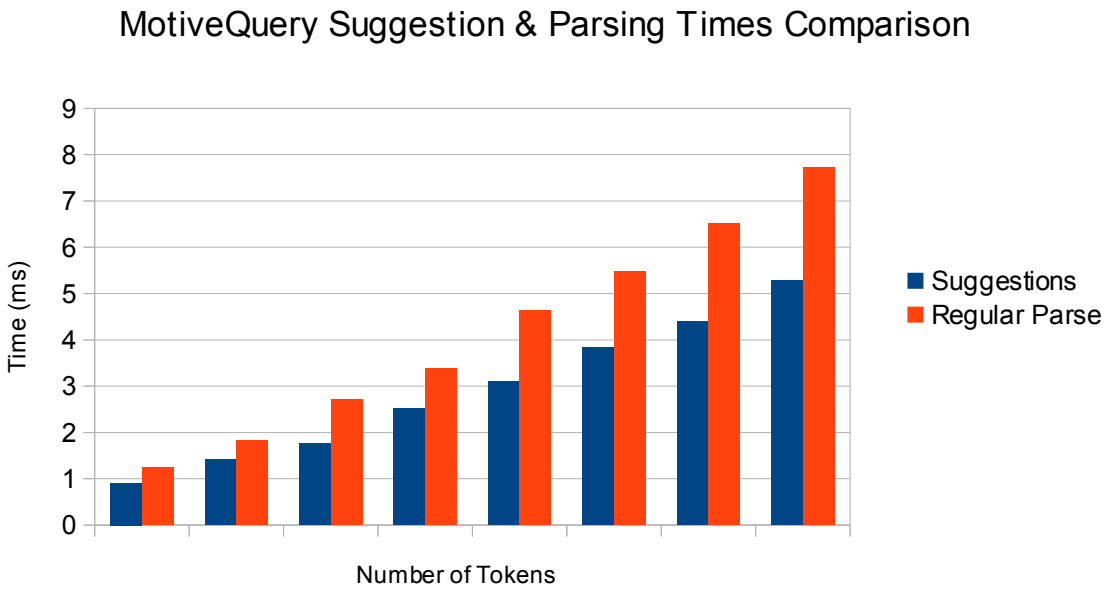

Figure 40: Parsing and suggestion generation times for MotiveQuery expressions.

5.4 Discussion

The tests have shown that even longer queries are parsable within roughly 10 ms even with provided error recovery turned on. The same applies to suggestion generation. What should be also noted is that these times were measured on a machine that is hardly comparable with modern servers. Therefore, it is probable that Earley Parser Framework will perform even better in such environment. Another thing that was shown by the tests is that it is rather inadvisable to use ambiguous grammars.

One of the requirements was the use in multi-threaded environment. This is easily achievable: the instances of classes responsible for implementation of the grammar and the AH automaton can be accessed by instances of Earley parser from multiple threads. Thus, all users can be served by their own parser and the grammar and the automaton do not have to be constructed individually for each user.

It would seem desirable to test somehow the quality of error recovery techniques. Unfortunately, this issue is not as simple as it seems. While there are techniques [35] that allow to perform this, the trouble with those techniques is that they usually involve a database of known mistakes for some programming languages. Since the EPF is rather focused on working with specific DSL and parsing short queries written in them, this approach is not very usable. Moreover, the error recovery techniques often require a fine-

tuning before they can be used so what is applicable in one case may not be very usable in other. This is also one of the reasons why the EPF provides a general way of specifying error recovery strategy.

5.4.1 Future Work

There are several ways the EFP could be improved:

- *Stateful parsing*: The EFP was ment to work statelessly. The original thought was that the web environment tends to be stateless so it is not necessary to implement some sort of optimisation when a modified version of previous query is entered. On the other hand, the way Earley parser works is quite amenable to reuse of a modified query. To be more specific, if there is a prefix that two queries share, there is a sequence of corresponding Earley sets that do not have to be created again.
- *Link disambiguation*: The current version of EPF does not provide a mechanism for dealing with situation when an Earley item has multiple causal and predecessor links attached to it.
- *Nullable non-terminal derivation*: Currently, when the parse tree is constructed and a nullable non-terminal is encountered, a parse tree node with the terminal corresponding to epsilon is added. A technique which would allow to add a derive the actual nullable subtree could be beneficial.
- *Shared packed parse forest (SPPF)*: SPPF [36] is a data structure that can represent multiple parse trees. It could be used for ambiguous grammars.
- *Minor redesign*: It would be convenient for the users to be able to supply their own lexer. The overall performance might be increased with better inner implementation of Earley set.
- *Type-system support*: One of the things that might be looked into is some sort of meta-support for type systems.

6 Conclusion

The aim of the thesis was to implement a parser generator framework for context-free grammars in C# language with the following features:

- *Input*: The input grammar should be defined using the objects of C# language.
- *Error recovery*: The parser has to be able to recover from minor errors in the parsed string.
- *Suggestion generation*: Given a point in the input sequence, the parser should offer possible continuations.
- *Web environment*: The expected use is parsing queries entered on a web page.

The theoretical part of this thesis reviews possible candidates for the parsing algorithm with the focus on how they perform in error-handling, follow suggestions, performance, generality and ability to handle ambiguous grammars. The reviewed approaches include parsing expression grammars and LL, LR, GLR, GLL and Earley parsing. Almost all of the techniques were found to be able to provide the desired features. In the end, an automaton-based version of Earley algorithm was chosen due to its amenability to error recovery and simplicity of the error-handling functionality implementation.

The practical part describes the created Earley Parser Framework and provides thorough review of the parsing algorithm which employes a slight modification that simplifies its implementation in object-oriented language such as C#.

The framework allows the users to implement their own error recovery technique. There are also two techniques provided by default: modified Burke-Fisher error recovery and Insertion-only error recovery. Another error-handling feature of the parser is that the way the errors are reported can be customized.

The use of the framework was demonstrated on MotiveQuery language for which a mechanism for suggestion generation was created.

The way the framework is implemented ensures that some components can be reused across multiple parsers. The components in question are the grammar and the automaton used in the Earley algorithm. As a result, the computing resources are not wasted on repeated creation of these components. Obviously, this feature is desirable for the use in web-based projects.

Performance tests of the parser showed that it is able to create a parse tree, generate a suggestion or recover from error with the use of default techniques within 10 ms on the input with length up to 400 tokens. This is rather positive as an application is considered well-responsive if it responds within 100-200 ms to user's request. What this implies is that the framework can be used for suggestion generation. The tests also confirm expected

theoretical performance of the Earley algorithm; the times of unambiguous grammars indicate linear or quadratic dependency of execution time on the size of the input. The only performance limitation of the framework seems to be use with ambiguous grammars; the results on input of length 400 tokens were more than 30 times slower on ambiguous grammars than on its unambiguous counter-parts.

Bibliography

[1] T. Æ. Mogensen, *Basics of Compiler Design*. Torben Ægidius Mogensen, 2009.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006.

[3] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide (Monographs in Computer Science)*. Springer, 2007, p. 662.

[4] M. Sipser, *Introduction to the Theory of Computation*. Course Technology, 2005, p. 456.

[5] T. A. Wagner and S. L. Graham, "Efficient and Flexible Incremental Parsing," *ACM Transactions on Programming Languages and Systems*, vol. 20, 1996.

[6] S. G. McPeak, "Elkhound: A Fast, Practical GLR Parser Generator," Berkeley, CA, USA, Jan. 2003.

[7] C. Brabrand, R. Giegerich, and A. Møller, "Analyzing ambiguity of context-free grammars," in *Proceedings of the 12th international conference on Implementation and application of automata*, 2007, pp. 214–225.

[8] B. Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl," in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming - ICFP '02*, 2002, vol. 37, no. 9, pp. 36–47.

[9] R. R. Redziejowski, "Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking," *Fundamenta Informaticae*, vol. 79, no. 3–4, pp. 513–524, Aug. 2007.

[10] "Clang - Features and Goals." [Online]. Available: http://clang.llvm.org/features.html. [Accessed: 30-Dec-2012].

[11] "New_C_Parser - GCC Wiki." [Online]. Available: http://gcc.gnu.org/wiki/New_C_Parser. [Accessed: 30-Dec-2012].

[12] “ANTLR Parser Generator.” [Online]. Available: http://www.antlr.org/. [Accessed: 31-Dec-2012].

[13] T. Parr and K. Fisher, “LL(*),” *ACM SIGPLAN Notices*, vol. 46, no. 6, p. 425, Jun. 2011.

[14] P. Norvig, “Techniques for automatic memoization with applications to context-free parsing,” *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, Mar. 1991.

[15] D. Knuth, “On the Translation of Languages from Left to Right,” *Information and Control*, vol. 8, pp. 607 – 639, 1965.

[16] Free Software Foundation, Inc., “Bison - GNU parser generator,” 2012. [Online]. Available: http://www.gnu.org/software/bison/. [Accessed: 02-Jan-2013].

[17] “Elkhound and Elsa.” [Online]. Available: http://scottmcpeak.com/elkhound/. [Accessed: 02-Jan-2013].

[18] M. Tomita, *Generalized Lr Parsing (Google eBook)*. Springer, 1991, p. 166.

[19] B. Ford and M. F. Kaashoek, “Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking,” Massachusetts Institute of Technology, 2002.

[20] A. Warth, J. R. Douglass, and T. Millstein, “Packrat parsers can support left recursion,” in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM ’08*, 2008, p. 103.

[21] L. Tratt, “Direct Left-Recursive Parsing Expressing Grammars,” Technical report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, 2010.

[22] R. Grimm, “Practical Packrat Parsing.” Technical Report TR2004854, Dept. of Computer Science, New York University.

[23] “Home · sirthias/parboiled Wiki · GitHub.” [Online]. Available: https://github.com/sirthias/parboiled/wiki. [Accessed: 03-Jan-2013].

[24] B. Ford, “Parsing expression grammars,” *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 111–122, Jan. 2004.

[25] E. Scott and A. Johnstone, “GLL Parsing,” *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 177–189, Sep. 2010.

[26] A. Johnstone and E. Scott, “Modelling GLL parser implementations,” pp. 42–61, Oct. 2010.

[27] “djspiewak/gll-combinators · GitHub.” [Online]. Available: https://github.com/djspiewak/gll-combinators. [Accessed: 04-Jan-2013].

[28] J. Earley, “An efficient context-free parsing algorithm,” *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970.

[29] J. Aycock and N. Horspool, “Practical Earley Parsing,” 2002. [online] Available: http://courses.engr.illinois.edu/cs421/sp2012/project/PracticalEarleyParsing.pdf. [Accessed: 05-Jan-2013]

[30] J. Aycock and A. Borsotti, “Early action in an Earley parser,” *Acta Informatica*, vol. 46, no. 8, pp. 549–559, Oct. 2009.

[31] “Irony - .NET Language Implementation Kit. - Home.” [Online]. Available: http://irony.codeplex.com/. [Accessed: 05-Jan-2013].

[32] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 2004, p. 556.

[33] D. Sehnal, “MotiveQuery Language Reference - WebChem Wiki.” [Online]. Available: http://webchem.ncbr.muni.cz/Wiki/index.php?title=MotiveQuery_Language_Reference. [Accessed: 23-May-2013].

[34] J. Nielsen, “Response Time Limits: Article by Jakob Nielsen.” [Online]. Available: http://www.nngroup.com/articles/response-times-3-important-limits/. [Accessed: 25-May-2013].

[35] P. Degano and C. Priami, “Comparison of Syntactic Error Handling in LR Parsers,” 1995.

[36] E. Scott, “SPPF-Style Parsing From Earley Recognisers,” *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 2, pp. 53–67, Apr. 2008.

Appendix A

Aycock-Horspool Automaton Creation

```
InitializeAutomaton(){
  initialState = {[S'→•S]}
  if S'e->Se belongs to input NNF grammar {
    add S'e->Se to initialState;
  }

  Add(KERNEL_STATES, initialState);
  Enqueue(UNPROCESSED_STATES_QUEUE, initialState);

  while UNPROCESSED_STATES_QUEUE is not empty {
    ProcessState(Dequeue(UNPROCESSED_STATES_QUEUE));
  }

  mark state with item [S'→S•] as final;
}

ProcessState(originalState){
  transitionSymbols=ExtractTransitionSymbols(originalState);
  foreach(symbol in transitionSymbols) {
    newState=GetItemsWithDotAdvancedOverSymbol(
                                originalState,
                                symbol);
    if newState is not empty {
      AddGoto(originalState, symbol, newState);
      if newState is not in KERNEL_STATES {
        Enqueue(UNPROCESSED_STATES_QUEUE, newState);
        Add(KERNEL_STATES, newState);
      } else {
        newState = newState from KERNEL_STATES;
      }
      AddGoto(originalState, symbol, newState);
    }
```

```
  }

  if originalState is not in NON_KERNEL_STATES {
    nonKernelState=ComputeNonKernelState(originalState);
    if nonKernelState is not empty {
      AddGoto(originalState, EMPTY_SYMBOL, nonKernelState);
    }
  }
}

ComputeNonKernelState(kernelState) {
  newState = ComputeInitialNonKernelState(kernelState);

  repeat
    foreach(A→α•Bβ in newState){
      foreach(B→X1...Xk, k > 0 in input NNF grammar) {
        newItem = CreateItem(B→X1...Xk);
        add newItem to newState;
      }
  until no new item is added;

  if newState is not empty {
    if newState in NON_KERNEL_STATES {
      return newState from NON_KERNEL_STATES;
    } else {
      Enqueue(UNPROCESSED_STATES_QUEUE, newState);
    }
  }
  return newState;
}

ComputeInitialNonKernelState(kernelState) {
  initialState = {};
  foreach(A→α•Bβ in kernelState){
    foreach(B→X1...Xk, k > 0 in input NNF grammar) {
      newItem = CreateItem(B→X1...Xk);
      add newItem to initialState;
```

```
    }
  }
  return initialState;
}

CreateItem(B→X1...Xk) {
  if X1 is not nullable {
    return B→•X1...Xk;
  } else {
    let X1...Xj be nullable nonterminals, j>0;
    return B→X1..Xj•Xj+1...Xk;
  }
}
AdvanceDot(B→α•β) {
  if β is empty { return B→α•β;}
  if (|β| == 1) { return B→αβ•;}

  let β = YX1...Xk, k > 0;
  if X1 is not nullable {
    return B→αY•X1...Xk;
  } else {
    let X1...Xj be nullable nonterminals, j>0;
    return B→αYX1...Xj•Xj+1...Xk;
  }
}
```

Appendix B

MotiveQueryGrammar Class

```
public class MotiveQueryGrammar : Grammar
{
    public static class MqNonTerminals
    {
        public static readonly NonTerminal Expr = new NonTerminal("expr");
        public static readonly NonTerminal LambdaDef = new
NonTerminal("lambda_def");
        public static readonly NonTerminal LambdaParams = new
NonTerminal("lambda_params");
        public static readonly NonTerminal LambdaHead = new
NonTerminal("lambda_head");
        public static readonly NonTerminal Param = new NonTerminal("param");
        public static readonly NonTerminal OrExpr = new
NonTerminal("or_expr");
        public static readonly NonTerminal AndExpr = new
NonTerminal("and_expr");
        public static readonly NonTerminal NotExpr = new
NonTerminal("not_expr");
        public static readonly NonTerminal ComparisonExpr = new
NonTerminal("comparison_expr");
        public static readonly NonTerminal CompOp = new
NonTerminal("comp_op");
        public static readonly NonTerminal AddExpr = new
NonTerminal("add_expr");
        public static readonly NonTerminal AddOp = new
NonTerminal("add_op");
        public static readonly NonTerminal MultExpr = new
NonTerminal("mult_expr");
        public static readonly NonTerminal MultOp = new
NonTerminal("mult_op");
        public static readonly NonTerminal UnaryExpr =  new
NonTerminal("unary_expr");
        public static readonly NonTerminal PowerExpr = new
NonTerminal("power_expr");
        public static readonly NonTerminal MemberAccess = new
NonTerminal("member_access");
        public static readonly NonTerminal SubscriptStart = new
NonTerminal("subscript_start");
        public static readonly NonTerminal SubscriptEnd = new
```

```
NonTerminal("subscript_end");
        public static readonly NonTerminal Atom = new NonTerminal("atom");
        public static readonly NonTerminal Subscript = new
NonTerminal("subscript");
        public static readonly NonTerminal OptExpr = new
NonTerminal("opt_expr");
        public static readonly NonTerminal Func = new NonTerminal("func");
        public static readonly NonTerminal FuncBegin = new
NonTerminal("func_begin");
        public static readonly NonTerminal FuncEnd = new
NonTerminal("func_end");
        public static readonly NonTerminal ArgList = new
NonTerminal("arg_list");
        public static readonly NonTerminal Arg = new NonTerminal("arg");
        public static readonly NonTerminal ArgSeparator = new
NonTerminal("arg_separator");
        public static readonly NonTerminal NamedArg = new
NonTerminal("named_arg");
        public static readonly NonTerminal ListBody = new
NonTerminal("list_body");
        public static readonly NonTerminal ListStart = new
NonTerminal("list_start");
        public static readonly NonTerminal ListEnd = new
NonTerminal("list_end");
        public static readonly NonTerminal NamedArgHead = new
NonTerminal("named_arg_head");

        static MqNonTerminals()
        {
            FuncBegin.NonterminalRecognizedAction =
NonTerminalActionsLibrary.FunctionHeadRecognized;
            FuncEnd.NonterminalRecognizedAction =
NonTerminalActionsLibrary.FunctionEndRecognized;
            ArgSeparator.NonterminalRecognizedAction =
NonTerminalActionsLibrary.ArgumentSeparatorRecognized;
            NamedArgHead.NonterminalRecognizedAction =
NonTerminalActionsLibrary.NamedArgHeadRecognized;
            LambdaHead.NonterminalRecognizedAction =
NonTerminalActionsLibrary.LambdaHeadRecognized;
            ListStart.NonterminalRecognizedAction =
NonTerminalActionsLibrary.ListBeginRecognized;
            ListEnd.NonterminalRecognizedAction =
NonTerminalActionsLibrary.ListEndRecognized;
            SubscriptStart.NonterminalRecognizedAction =
NonTerminalActionsLibrary.SubscriptBeginRecognized;
            SubscriptEnd.NonterminalRecognizedAction =
```

```
NonTerminalActionsLibrary.SubscriptEndRecognized;
        }
    }

    public MotiveQueryGrammar()
    {

        #region terminals
        RegexTerminal INT = new RegexTerminal("INT",
                                              "[0-
9]+",TerminalPriority.Nominal);
        RegexTerminal REAL = new RegexTerminal("REAL", "[0-9]+\.[0-9]+",
                                               TerminalPriority.Nominal+1);
        RegexTerminal ID = new RegexTerminal("ID", "[a-z|A-Z|_][a-z|A-Z|_|0-
9]*",
                                             TerminalPriority.Nominal);
        RegexTerminal STRING = new RegexTerminal("STRING", "\".*\"",
                                                 TerminalPriority.Nominal);
        RegexTerminal CHAR = new RegexTerminal("CHAR", "\'.\'",
                                               TerminalPriority.Nominal);
        Keyword TRUE = new Keyword("True");
        Keyword FALSE = new Keyword("False");
        Keyword LAMBDA = new Keyword("lambda");
        ID.CanBeIncomplete = true;
        #endregion

        #region nonterminals
        NonTerminal expr = MqNonTerminals.Expr;
        NonTerminal lambdaDef = MqNonTerminals.LambdaDef;
        NonTerminal lambdaParams = MqNonTerminals.LambdaParams;
        NonTerminal lambdaHead = MqNonTerminals.LambdaHead;
        NonTerminal param = MqNonTerminals.Param;
        NonTerminal orExpr = MqNonTerminals.OrExpr;
        NonTerminal andExpr = MqNonTerminals.AndExpr;
        NonTerminal notExpr = MqNonTerminals.NotExpr;
        NonTerminal comparisonExpr = MqNonTerminals.ComparisonExpr;
        NonTerminal compOp = MqNonTerminals.CompOp;
        NonTerminal addExpr = MqNonTerminals.AddExpr;
        NonTerminal addOp = MqNonTerminals.AddOp;
        NonTerminal multExpr = MqNonTerminals.MultExpr;
        NonTerminal multOp = MqNonTerminals.MultOp;
        NonTerminal unaryExpr =  MqNonTerminals.UnaryExpr;
        NonTerminal powerExpr = MqNonTerminals.PowerExpr;
        NonTerminal memberAccess = MqNonTerminals.MemberAccess;
        NonTerminal subscriptStart = MqNonTerminals.SubscriptStart;
        NonTerminal subscriptEnd = MqNonTerminals.SubscriptEnd;
```

```
        NonTerminal atom = MqNonTerminals.Atom;
        NonTerminal subscript = MqNonTerminals.Subscript;
        NonTerminal optExpr = MqNonTerminals.OptExpr;
        NonTerminal func = MqNonTerminals.Func;
        NonTerminal funcBegin = MqNonTerminals.FuncBegin;
        NonTerminal funcEnd = MqNonTerminals.FuncEnd;
        NonTerminal argList = MqNonTerminals.ArgList;
        NonTerminal arg = MqNonTerminals.Arg;
        NonTerminal argSeparator = MqNonTerminals.ArgSeparator;
        NonTerminal namedArg = MqNonTerminals.NamedArg;
        NonTerminal namedArgHead = MqNonTerminals.NamedArgHead;
        NonTerminal listBody = MqNonTerminals.ListBody;
        NonTerminal listStart = MqNonTerminals.ListStart;
        NonTerminal listEnd = MqNonTerminals.ListEnd;
        #endregion

        #region rules
        expr.Rule = orExpr | lambdaDef;
        lambdaDef.Rule = lambdaHead + expr;
        lambdaHead.Rule = LAMBDA + lambdaParams + ":" | LAMBDA + ":";
        lambdaParams.Rule = param + "," + lambdaParams | param;
        param.Rule = ID + "=" + expr | ID;
        orExpr.Rule = orExpr + "|" + andExpr | andExpr;
        andExpr.Rule = andExpr + "&" + notExpr | notExpr;
        notExpr.Rule = "!" + notExpr | comparisonExpr;
        comparisonExpr.Rule = comparisonExpr + compOp + addExpr | addExpr;
        compOp.Rule = new Keyword("<=") | "<" | ">=" | ">" | "==" | "!=";
        addExpr.Rule = addExpr + addOp + multExpr | multExpr;
        addOp.Rule = new Keyword("+") | "-";
        multExpr.Rule = multExpr + multOp + unaryExpr | unaryExpr;
        multOp.Rule = new Keyword("*") | "/";
        unaryExpr.Rule = addOp + unaryExpr | powerExpr;
        powerExpr.Rule = memberAccess + "^" + unaryExpr | memberAccess;
        memberAccess.Rule= memberAccess + subscriptStart + subscript +
subscriptEnd|
                           memberAccess + "." + func |
                           memberAccess + "." + ID |
                           atom;
        subscriptStart.Rule =  new Keyword("[");
        subscriptEnd.Rule = new Keyword("]");
        subscript.Rule = expr | optExpr + ":" + optExpr;
        optExpr.Rule = Grammar.Empty | expr;
        atom.Rule = INT | REAL | ID | TRUE | FALSE | STRING | CHAR |
                    func | "(" + expr + ")" | listStart + listBody +
listEnd;
        listBody.Rule = expr + "," + listBody | expr;
```

```
        listStart.Rule = new Keyword("[");
        listEnd.Rule = new Keyword("]");
        func.Rule = funcBegin + argList + funcEnd | funcBegin + funcEnd;
        funcBegin.Rule = ID + "(";
        funcEnd.Rule = new Keyword(")");
        argList.Rule = arg + argSeparator + argList | arg;
        argSeparator.Rule = new Keyword(",");
        arg.Rule = namedArg | expr;
        namedArg.Rule = namedArgHead + expr;
        namedArgHead.Rule = ID + "=";
        #endregion

        MarkBrackets("(",")");
        MarkBrackets("[", "]");
        SetTerminalsErrorValue(30, "(",")","[","]",",",".");

        Root = expr;
    }
}
```

Appendix C

Measurements

Table 4: Measured parsing times for MotiveQuery, expression and ambiguous expression grammars.

MotiveQuery Grammar Parse Times								
Number of Tokens	51	99	147	195	251	299	347	401
Time (ms)	1.24	1.82	2.71	3.38	4.63	5.48	6.51	7.73
Expression Grammar Parse Times								
Number of Tokens	50	100	150	200	250	300	350	400
Time (ms)	0.63	0.77	1.17	1.47	1.89	2.33	2.75	3.06
Ambiguous Expression Grammar Parse Times								
Number of Tokens	50	100	150	200	250	300	350	400
Time (ms)	1.23	3.59	10.8	17.85	37.24	87.12	157.25	264.7

Table 5: Measured times of parsing with error recovery enabled. Modified Burke-Fisher error repair was used for expression grammar. Insertion-only recovery was used for MotiveQuery grammar.

MotiveQuery Recovered Parsing Times						
Number of Errors	0	1	2	4	6	8
Time (ms)	7.73	8.02	7.99	7.85	9.01	9.07
Simple Expression Recovered Parsing Times						
Number of Tokens	0	1	2	4	6	8
Time (ms)	3.06	3.14	3.09	3.06	3.1	3.15

Table 6: Measured time of suggestion generation for MotiveQuery.

Appendix D

Expression Grammar

```
class ExpressionGrammar : Grammar
    {
        public ExpressionGrammar()
        {
            NonTerminal atom = new NonTerminal("atom");
            NonTerminal addExpr = new NonTerminal("addExpr");
            NonTerminal multExpr = new NonTerminal("mult");
            NonTerminal unExpr = new NonTerminal("unary");
            NonTerminal func = new NonTerminal("func");
            NonTerminal argList = new NonTerminal("argList");

            RegexTerminal INTEGER = new RegexTerminal("integer", "[0-9]+",
                                                TerminalPriority.Nominal);
            RegexTerminal ID = new RegexTerminal("ID", "[a-z|A-Z]+",
                                            TerminalPriority.Nominal);

            addExpr.Rule = addExpr + "+" + multExpr |
                     addExpr + "-" + multExpr |
                     multExpr;
            multExpr.Rule = multExpr + "*" + unExpr |
                     multExpr + "/" + unExpr |
                     unExpr;
            unExpr.Rule = atom | "-" + atom;
            atom.Rule = new Keyword("(") + addExpr + ")" | INTEGER | func;
            func.Rule = ID + "(" + argList + ")";
            argList.Rule = addExpr | addExpr + "," + argList;

            Root = addExpr;

            SetTerminalsErrorValue(600,",");
            SetTerminalsErrorValue(650, ")", "(");
        }
    }
```

Ambiguous Expression Grammar

```
    class AmbigousExpressionGrammar : Grammar
    {
```

```
        public AmbigousExpressionGrammar()
        {
            NonTerminal atom = new NonTerminal("atom");
            NonTerminal addExpr = new NonTerminal("addExpr");
            NonTerminal multExpr = new NonTerminal("mult");
            NonTerminal unExpr = new NonTerminal("unary");
            NonTerminal func = new NonTerminal("func");
            NonTerminal argList = new NonTerminal("argList");

            RegexTerminal INTEGER = new RegexTerminal("integer", "[0-9]+",
TerminalPriority.Nominal);
            RegexTerminal ID = new RegexTerminal("ID", "[a-z|A-Z]+",
TerminalPriority.Nominal);

            addExpr.Rule = addExpr + "+" + addExpr |
                     addExpr + "-" + addExpr |
                     multExpr;
            multExpr.Rule = multExpr + "*" + multExpr |
                     multExpr + "/" + multExpr |
                     unExpr;
            unExpr.Rule = atom | "-" + atom;
            atom.Rule = new Keyword("(") + addExpr + ")" | INTEGER | func;
            func.Rule = ID + "(" + argList + ")";
            argList.Rule = addExpr | addExpr + "," + argList;

            Root = addExpr;

            SetTerminalsErrorValue(600, ",");
            SetTerminalsErrorValue(650, ")", "(");
        }
    }
```

Appendix E

Test Queries

Table 7: Queries used for the expression grammar parsing time tests.

Expression Grammar Queries	
Number of Tokens	Query
50	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555-666666+3*Min(4,6)+1*2
100	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6-3*555- 666666+3*Min(4,6)+1*2
150	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6-3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6-3*555- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2
200	10000+2+3*4-2+Max(Abs(-3),-1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)
250	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)
300	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2/10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2
350	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555-

	666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2/10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6-- 666666+3*Min(4,6)+2
400	10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2/10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+-3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6-- 666666+3*Min(4,6)

Table 8: Queries used for the MotiveQuery grammar parsing time tests. These queries were also used for suggestion parsing. In that case, symbol "." was appended to them.

MotiveQuery Queries	
Number of Tokens	Query
51	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))
99	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))
147	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))
195	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r:

	r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))
251	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\")))
299	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\")))
347	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r:

	r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\")))
401	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))

Table 9: Queries with the maximum number of errors tested. All other test queries were devised from these by correcting the errors.

Error Recovery Queries	
MotiveQuery Grammar (400 tokens, 8 errors)	AtomSimilarity(Current().Find(NotAtoms(\"N\")).ToMotive(), Motive(\"model\").Find(NotAtoms(\"N\")).ToMotive()).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\" \"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\" \"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\" \"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\" \"Ca\",\"Ca\",\"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\",\"Ca\",\"Ca\" \"Ca\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\" \"Cb\",\"Cc\"))).Filter(lambda r: r.IsNotConnectedTo(Atoms(\"Ca\")))
Expression Grammar (400 tokens, 8 Errors)	10000+2+3*4-2+Max(Abs(-3) 1)*(8+3)*30/6--3*555-666666+3*Min(4 6)+1*2+10000+2+3*4-2+Max(Abs(-3),1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3) 1)*(8+3)*30/6-- 3*555-666666+3*Min(4,6)+1*2+10000+2+3*4-2+Max(Abs(- 3),1)*(8+3)*30/6--3*555-666666+3*Min(4,6)+1*2-10000+2+3*4- 2+Max(Abs(-3),1)*(8+3)*30/6--3*555-666666+3*Min(4 6)+1*2/10000+2+3*4-2+Max(Abs(-3) 1)*(8+3)*30/6--3*555- 666666+3*Min(4,6)+1*2-10000+2+-3*4-2+Max(Abs(-3) 1)*(8+3)*30/6-- 3*555-666666+3*Min(4,6)+1*2*10000+2+3*4-2+Max(Abs(-3) 1)*(8+3)*30/6--666666+3*Min(4 6)

Appendix F

Contents of the Attached CD

The attached CD contains MS Visual Studio 2012 MS project EarleyParserFramework which is expected to be placed into folder C:\Users\Richard\Documents\Visual Studio 11\Projects\. The project references libraries created by David Sehnal that provide type-system support for the MotiveQuery language. The libraries are included in the project.