A Tool for Generalised LR Parsing in Haskell

Ben Medlock – St. John's College

Single Honours CS Project Report, April 2002

Word count: 17,857

Abstract

Parsing is the process of deriving structure from sentences in a given language. This structure is derived from a specification of the language defined by a formal grammar. Many different types of grammar exist, but those most often used in the field of computer science are known as context-free (CF) grammars. The LR parsing technique can be used to efficiently parse a large class of *unambiguous* CF grammars. However, many languages can only be specified using *ambiguous* grammars. These include natural language (NL) grammars, as well as a host of simpler grammars. Tomita (85) proposed an extension to LR parsing known as *generalised* LR (GLR) parsing which allows languages derived from *ambiguous* CF grammars to be parsed efficiently.

The project implements a version of Tomita's algorithm in the functional programming language Haskell and integrates it with the Haskell-based LR parser-generator tool *Happy*. The amendments to *Happy* allow it to generate a GLR parser, based on Tomita's algorithm, capable of parsing languages derived from ambiguous CF grammars. Our implementation of Tomita's algorithm is analysed both theoretically (time and space orders) and through the use of Haskell profiling tools.

Table of Contents

Table of Figures

Ch 1. Introduction

The primary objective of this chapter is to offer an introduction to two project-related topics:

- *Parsing* – deriving structure from sentences under a specified grammar
- *Haskell* – a functional programming language

A clear overview of the project objectives will then be supplied, along with the list of project deliverables. The chapter closes with the project plan and a brief guide to the rest of the report.

1.1 Introduction to Parsing

1.1.1 Language, Grammar and Syntax

Language is a fundamental and universally familiar concept, providing the foundation for communication. It is also fundamental to the field of computer science. From the earliest days of research into computational theory scientists have sought ways to express in language what can be achieved through the use of computational mechanisms. This section discusses the basic linguistic concepts of language, grammar and syntax.

Grishman [Gri 86] defines the concept of a language as *a set of sentences, where each sentence is a string of one or more symbols (words) from the vocabulary of the language*. We should further stipulate that the string comprising a sentence be of finite length. For example, consider the English language, consisting of collections of sentences made up of words separated by delimiting punctuation symbols:

My name is John. I live in England. Where do you live?

Punctuation symbols designate the end of a sentence, and in English may also convey semantic information about the sentence, i.e. that it is a question or statement.

In a computational language such as Java, we also observe this sentential structure:

```
int x = 1;
int y = 2;
int z = x + y;
```

Here, the sentences are delimited by the semicolon symbol (;). The vocabulary of the Java language is the set of all valid symbols, including keywords, identifiers and operators. Note that the semicolon exists simply to designate the end of the sentence; no further information is implied.

We could define our own language by enumerating all of its sentences:

```
L = { "aa" , "ab" , "ba" , "bb" }
```

This is a definition of the language `L`, consisting of the four specified sentences. Note that the symbols `a` and `b` make up the vocabulary of the language. Also note that `L` consists of a *finite* number of sentences, whereas both English and Java consist of an *infinite* number of sentences. We cannot define a language of infinite sentences simply by enumerating all of them; therefore we must use another method. Such languages are commonly defined using *formal grammars*. Grishman [Gri 86] defines a formal grammar as *a finite, formal specification of a set of sentences (language).*

The modern definition of a formal grammar stems mainly from work carried out in the early-to-mid 20th century, especially by an American linguist called Noam Chomsky who specified a hierarchy of formal grammars still extensively used today, the *Chomsky Hierarchy*. A formal grammar consists of a set of rules (called *productions* or *production rules*) defining how valid sentences are to be constructed. For example, we could redefine our language L using the following grammar:

```
S → T T
T → a | b
```

Grammar 1.1: Formal grammar

`S` and `T` are known as *non-terminals* – they do not actually appear in the language itself, but are used to define intermediate stages in the grammar. `a` and `b` are known as *terminals* or *tokens* – they comprise the vocabulary of the language. The second rule is shorthand for two rules: `T → a` and `T → b`. The non-terminal symbol *S* is the *start symbol* of the grammar, also known as the *sentence constituent*. The grammar states that a valid sentence `S` in the language is composed of two consecutive non-terminals – `T T`, where `T` is either of the terminals `a` or `b`.

The grammar described above is an instance of a *context-free* grammar. Context-free grammars are defined in the Chomsky hierarchy as those with production rules of the form:

$$A \rightarrow \alpha$$

Where A is a single non-terminal, and α is a list of non-terminals or terminals. Most grammars used in the field of computer science are context-free. More powerful classes of grammar exist, but are usually too complex to be parsed efficiently.

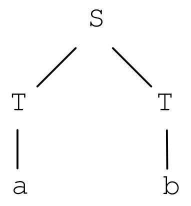
A sequence of symbols is a valid sentence in a particular language, specified by its corresponding grammar, if we can perform a *derivation* beginning with the sentence constituent (`S` in the case of language `L`) and ending with the original sequence of terminal symbols. A derivation is valid if at each stage a non-terminal is identified as the left-hand side of a production rule and replaced by the right-hand side of that rule. For example, consider the following derivation of the sentence “ab” in the language L:

$$S \rightarrow T\ T \rightarrow a\ T \rightarrow a\ b$$

This is known as a *leftmost derivation*, because at each stage the leftmost non-terminal has been replaced. A *rightmost derivation* would appear as follows:

$$S \rightarrow T\ T \rightarrow T\ b \rightarrow a\ b$$

A grammar does more than just provide a convenient means of specifying valid sentences in a particular language. It also adds *structure* to the language. This structure is known as *syntax*. The derivations shown above could also be represented in tree form. Each node represents a non-terminal in the grammar, and each leaf represents a terminal:



It is precisely this structure, or syntax, that we are interested in when we carry out *parsing*. Butler [But 92] defines parsing as *the process by which grammatical strings of words are assigned syntactic structure*.

Thus, the process of parsing is concerned with taking ‘flat’ sequences of words in the vocabulary of a given language, and converting them into richly-structured trees, illuminating the underlying syntax of the language.

Parsing techniques are divided into two main categories: *top-down* and *bottom-up*. A top-down parser begins with the start symbol of the grammar and tries various sequences of production rules until it ends up with either an error or a sequence of terminal symbols that matches the input string. A bottom-up parser scans the input string, looking for patterns of symbols that match the right-hand side of one of the production rules. Once it has found such a pattern it can replace it with the left-hand side of the production rule. The parse terminates when either an error has been detected, or the entire input string has been reduced to the start symbol of the grammar.

1.1.2 LR Parsing

We now consider a particular type of bottom-up parsing technique, known as LR parsing. LR denotes the fact that the parser scans input from (L)eft-to-right, and that a (R)ightmost derivation is produced. LR parsers will only succeed on a subclass of the context-free grammars, known as LR grammars. A given

CF grammar is LR(k) if it can be parsed by an LR parser with k *lookahead* tokens, i.e. by looking at the next k tokens in the input string. In practice k is always ≤ 1. The following is an LR(1) grammar:

```
E → E + T | T
T → i | T * i
```

Grammar 1.2: Simple LR(1) grammar

In LR parsing, a deterministic finite state automaton (DFA) is pre-computed from the input grammar, representing all states the parser can be in. The most common method of implementing LR parsing is known as *shift-reduce* parsing. It utilises a *parse stack*, containing symbols from the grammar paired with state values (representing states in the LR automaton). At each token of input the parser can take one of four actions:

- *shift* – remove the token from the input and push it onto the stack, along with a new state.
- *reduce* – pop a number of states and symbols from the stack, corresponding to the right-hand side of a production (known as a *handle*), and replace them with the left-hand side of the production.
- *accept* – the parse has been successful; terminate
- *halt* – an error has occurred; terminate

The parse stack is initialised to contain a single value representing the initial state, and parsing either terminates successfully when all the input has been consumed and the stack contains the initial and final states along with the start symbol of the grammar, or halts with an error. At each stage, the parser makes its decisions based on the pre-computed DFA, known as a *parse table*. Table 1.3 is an example parse table for Grammar 1.2:

	Action				Goto	
	i	*	+	$	E	T
0	sh4				5	3
1	sh4				2	3
2			sh6			
3		sh7	re2	re2		
4		re3	re3	re3		
5			sh6	acc		
6	sh4					9
7	sh8					
8		re4	re4	re4		
9		sh7	re1	re1		

Table 1.3: LR parse table for Grammar 1.2

As can be seen from Table 1.3, the parse table is made up of two sections: *Action* and *Goto*. The Action table specifies what must be done when considering a token (*terminal* of the grammar) from the input string. It takes a state and a terminal and returns an action.

The actions are encoded as follows:

- `sh`*N* (shift) – shift the input token onto the stack and move into state *N*.
- `re`*N* (reduce) – reduce by production rule *N*.
- `acc` (accept) – a successful parse has been found
- no entry (halt) – a parse error has occurred

The *Goto* table tells the parser what state it should be in after it has reduced by a specific production. It takes a state and a non-terminal (left-hand side of the reduction rule) and returns a new state.

Notice that the $ symbol, standing for EOF (end of file), appears in the parse table but is not present in the original grammar (Grammar 1.2). In LR parsing it is customary to augment the original grammar by adding a rule of the form:

```
S′ → S $
```

where `S′` is an unused grammar symbol, `S` is the start symbol of the original grammar and `$` is the EOF symbol. In Grammar 1.2, the new rule might be:

```
E′ → E $
```

All input strings to be parsed are then suffixed with the EOF symbol ($ in this case).
Consider the shift-reduce parse of the input string "`i + i * i`" using the parse table in Table 1.3 (current input symbol in bold):

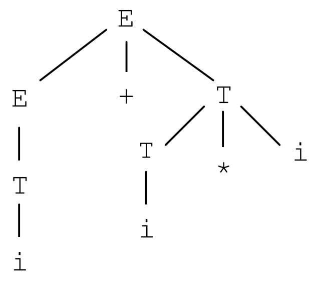
Stack	Input	Action
0	**i** + i * i $	shift (4)
0 i 4	**+** i * i $	reduce (T → i) (goto 3)
0 T 3	**+** i * i $	reduce (E → T) (goto 5)
0 E 5	**+** i * i $	shift (6)
0 E 5 + 6	**i** * i $	shift (4)
0 E 5 + 6 i 4	***** i $	reduce (T → i)
0 E 5 + 6 T 9	***** i $	shift (7)
0 E 5 + 6 T 9 * 7	**i** $	shift (8)
0 E 5 + 6 T 9 * 7 i 8	**$**	reduce (T → T * i) (goto 9)
0 E 5 + 6 T 9	**$**	reduce (E → E + T) (goto 5)
0 E 5	**$**	accept

Figure 1.4: Shift-reduce parse of input string '`i + i * i`'

Note that the combination of the stack and input (ignoring the state information) on each reduction line in the parse is a stage in a rightmost derivation of the input. The derivation can be produced by concatenating these results in reverse:

```
E → E + T → E + T * i → E + i * i → T + i * i → i + i * i
```

The syntax tree associated with the input sentence can be constructed from the derivation:



An issue that has been deliberately avoided thus far is that of constructing parse tables. It is possible to construct an LR parse table from any context-free grammar. There are a number of popular techniques, the four main ones being LR(0), SLR(1), LALR(1) and LR(1) (ascending order of strength). A thorough analysis can be found in [Aho 86].

Sometimes, using a given parse table generation technique will produce a DFA in which one of the states contains more than one action. If this is the case, we say that the parse table contains a *conflict*. There are two types of conflict:

- *shift/reduce* – either a shift or a reduce action can be taken
- *reduce/reduce* – one of two different reduce actions can be taken

Note that a *shift/shift* would correspond to a non-deterministic finite automaton (NDFA) and cannot occur in practice. See [Aho 86] for further details.

If a parse table that is conflict-free can be generated for a particular grammar using an LR(0) table generation technique, then we say that the grammar is LR(0). Often stronger techniques can eliminate conflicts in tables, so that for instance, if an LR(0) technique failed to produce a conflict-free table, an LALR(1) technique could be tried. If it was successful, then the grammar would be LALR(1) but not LR(0). The weaker classes of grammars are proper subsets of the stronger classes; for example a grammar that is SLR(1) is also LR(1).

1.1.3 Ambiguous Grammars

A large class of context-free grammars cannot be parsed deterministically using LR parsing techniques; i.e. whichever table generation technique is chosen will still result in a parse table with conflicts. Such grammars are *ambiguous*. [Gru 00] defines an ambiguous grammar as one *that can produce two different production trees with the same leaves in the same order*.

To illustrate this concept let us consider the following ambiguous grammar along with its LR parse table:

```
E → E + E | i
```

Grammar 1.5:
Simple ambiguous grammar

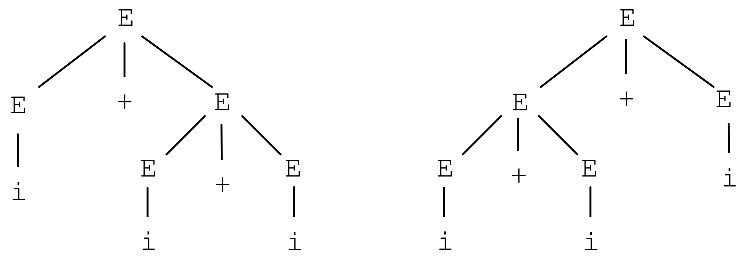
	Action			Goto
	i	+	$	E
0	sh3			4
1	sh3			2
2		sh5		
3		re2	re2	
4		sh5	acc	
5	sh3			6
6		sh5/re1	re1	

Table 1.6: LR parse table for Grammar 1.5

The table contains a shift/reduce conflict (highlighted) in state 6. This occurs because the grammar does not make clear the associativity of the '+' operator. Thus, for the simple sentence '`i+i+i`' there are two ways of associating:

```
i+(i+i) and (i+i)+i
```

and therefore two possible syntax trees:



It is clear that the leaves of the two trees are identical and in the same order, thus demonstrating the ambiguity of Grammar 1.5 by the definition above.

In this case it is easy to disambiguate the grammar by adding an *associativity rule* for the '+' operator. *Precedence* and *associativity rules* are often used to govern the interpretation of sentences containing operators such as those found in standard arithmetic or various logical languages. For example, a grammar might be embellished with the following rules:

```
precedence left *,/
precedence left +,-
```

denoting that the '*' and '/' operators have higher precedence than the '+' and '–' operators, and that they are all left associative. These rules are then used to disambiguate such sentences as the one above, '`i+i+i`', in that reductions take priority over shifts for rules defining use of the '+' operator. Thus '`i+i+i`' is parsed into the single syntax tree representing the association, '`(i+i)+i`'. By the use of precedence, the parser could also disambiguate sentences such as '`i+i*i`' under a grammar that defined both '+' and '*' ambiguously. '*' has higher precedence than '+', so the sentence is parsed into the syntax tree representing '`i+(i*i)`'.

Precedence and associativity rules can be used on simple expression grammars, but are inadequate when dealing with more complex ambiguous grammars.

1.1.4 Generalised LR (GLR) Parsing

Standard LR parsing techniques cannot handle ambiguous grammars; however, an extension to the basic LR technique can be used to deal with ambiguity. This technique is known as *Generalised* LR (GLR) parsing. A GLR parser handles a conflict in the parse table by performing *both* actions, conceptually in parallel. The parse stack is replaced with a data structure that can handle these multiple actions. When a parse is completed successfully, rather than returning a single syntax tree as in the case of a standard LR parser, a GLR parser will return a *forest* of valid syntax trees. GLR parsing is a major topic covered in the remainder of the report, so further details are left until then.

1.1.5 Tomita's Algorithm

Tomita's algorithm is a GLR parsing technique developed in the 1980's by Masaru Tomita [Tom 85]. He replaces the parse stack found in standard LR parsing with a Graph Structured Stack (GSS). As its name suggests, the GSS is a graph structure, capable of representing multiple parses of the input. Shift and reduce operations are performed on the GSS in a similar way to standard LR parsing. The result of a successful parse is a *packed forest* representing all valid parses of the input. Tomita's algorithm is discussed in detail in 3.1, so further explanation is left until then.

1.2 Introduction to Haskell

We now introduce Haskell, a pure functional programming language. It is the implementation language used in the project. This section introduces some of its basic features.

1.2.1 Functional Programming

The online encyclopaedia *Wikipedia* defines functional programming as:

> A style of programming that emphasises the evaluation of functional expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. [Wik 01]

A functional program is made up of functional expressions. As such, it contains no variables, no assignments and no iterative constructs. The program is executed by evaluating these expressions. The *lambda calculus*[1] provides a sound mathematical basis for functional programming theory. In a *pure* functional programming language, all computations are performed via function application.

1.2.2 Functions

A function in Haskell takes a number of arguments and returns a result:

```
                         Arguments
fn x y =  x * y
\  x v -> x * v
                        Body
```

Both of the above represent the same function. Some other simple examples are:

```
plus x y   = x + y          (Addition of two numbers)
\x y       -> x == y        (Equality comparison)
swap (x,y) = (y,x)          (Pair swapping)

fac 0 = 1
fac x = x * fac (x - 1)     (Recursive calculation of factorial value)
```

1.2.3 Types

Haskell is *strongly typed*, meaning an expression and its evaluation is a member of a determinable *type*. The '`::`' symbol in Haskell means 'of type'. For example:

```
42       :: Int
"hello"  :: String
True     :: Bool
("A",1)  :: (String,Int)
[1,2,3]  :: [Int]
```

Note that `(String,Int)` denotes a pair of values, where the first is of type `String` and the second is of type `Int`. Also note that `[Int]` represents a list of values of type `Int`.

Functions also have types. The symbol `'->'` denotes a function type:

```
plus :: Int -> Int -> Int
```

`plus` takes two arguments of type `Int` and returns an `Int`. New types can be introduced by the use of *type constructors*. See [Tho 99] for more information.

[1] see http://www.mactech.com/articles/mactech/Vol.07/07.05/LambdaCalculus/ for Lambda Calculus introduction

An important feature of Haskell is *polymorphic typing*. This makes it possible to define operations that can be carried out on arbitrarily-typed objects. Consider the following function:

```
swap (x,y) = (y,x)
```

It takes a pair of objects and swaps them. It is unimportant what the types of the two objects are – the operation should work on objects of any type. Thus we declare the function type as:

```
swap :: (a,b) -> (b,a)
```

This states that `swap` takes a pair of values of arbitrary type, and returns a pair of values where the type of the first element in the new pair is the type of the second element in the original pair, and vice-versa. Polymorphism is a useful programming tool, and can be used to greatly enhance code reusability.

Strong typing creates a safer programming environment, where many of the errors common in other languages (such as core dumps or cast exceptions) are necessarily eliminated.

1.2.4 Higher Order Functions

In Haskell, functions are *first-class citizens* of the language. Among other things, this means we can pass functions as arguments to other functions. A function that takes a function as an argument or returns a function is known as a *higher order function* (HOF). Consider the following function:

```
twice :: (a -> a) -> a -> a
twice fn x = fn (fn x)
```

The use of brackets in the function type denotes that a function is expected as the first argument. The function passed in will be of type '`a -> a`' (it will take a value of type '`a`', and return a value of type '`a`'). `twice` applies the given function to its second argument to return a value of type '`a`' upon which it applies the given function again to yield the result.

Higher order functions are a powerful programming abstraction, allowing significant improvements to the structure and clarity of programs.

1.2.5 Comments

Many features of Haskell have been omitted from this introduction. The inquisitive reader should note that the *Haskell Report* [Pey 99] provides a complete coverage of the language, while [Tho 99] gives a more practical guide to Haskell programming. In Chapter 2 we will consider some more advanced Haskell-related topics relevant to the project.

1.3 Project objectives

1.3.1 Implementation of Tomita's Algorithm

The first major objective of the project is to implement Tomita's algorithm in Haskell. This requires a thorough understanding of the algorithm itself, as well as the various related issues, such as standard LR parsing and the use of Haskell.

1.3.2 Integration with *Happy*

A further development of the project is the integration of the implementation of Tomita's algorithm with the Haskell parser-generator tool *Happy* (see [Mar 00]). *Happy* currently provides facility for the generation of LR parsers from unambiguous CF grammars. The project will extend it to allow handling of ambiguous CF grammars using the implementation of Tomita's algorithm.

1.3.3 Analysis and Evaluation

Analysis will be carried out on both theoretical and practical levels, aided by the use of Haskell profiling tools. The implementation will be evaluated for correctness and efficiency.

1.3.4 Other Possibilities

Possibilities include consideration of how one might efficiently process *parse forests*. This may include a brief look at some of the issues related to semantics arising from the processing of natural language grammars. Another possibility is to consider how to identify and deal with erroneous ambiguities arising from malformed input, such as that often found in web-pages based on HTML.

1.3.5 Overall Objectives

The overall success of the project relies on achievement in the following areas:

- A concise and maintainable implementation of Tomita's algorithm in Haskell that is both correct and efficient.
- Proof by analysis that these properties are indeed true of the implementation.
- A transparent integration of the implementation with the tool *Happy*, extending its capabilities to include reliable GLR parsing.

1.4 Project deliverables

The project objectives are officially met in the form of *deliverables*, broken up into three progressive stages:

1.4.1 Basic

- Study Tomita's algorithm and become familiar with its use.
- Design and implement a prototype of Tomita's algorithm and associated data structures in Haskell.

1.4.2 Intermediate

- Implement a technique for decoding a parse forest into a list of individual trees.
- Study the implementation of *Happy* and provide a summary of the results.
- Design and implement the changes to *Happy*.
- Develop a small but accurate natural language grammar for testing.

1.4.3 Advanced

Possibilities include:

- Investigate the efficiency of the implementation, using theoretical methods (time and space orders) and Haskell profiling tools to eliminate bottlenecks.
- Study and implement techniques for processing GLR parse results.

1.5 Project plan

1.5.1 Activity graph

The following activity graph indicates dependencies between tasks in the project:

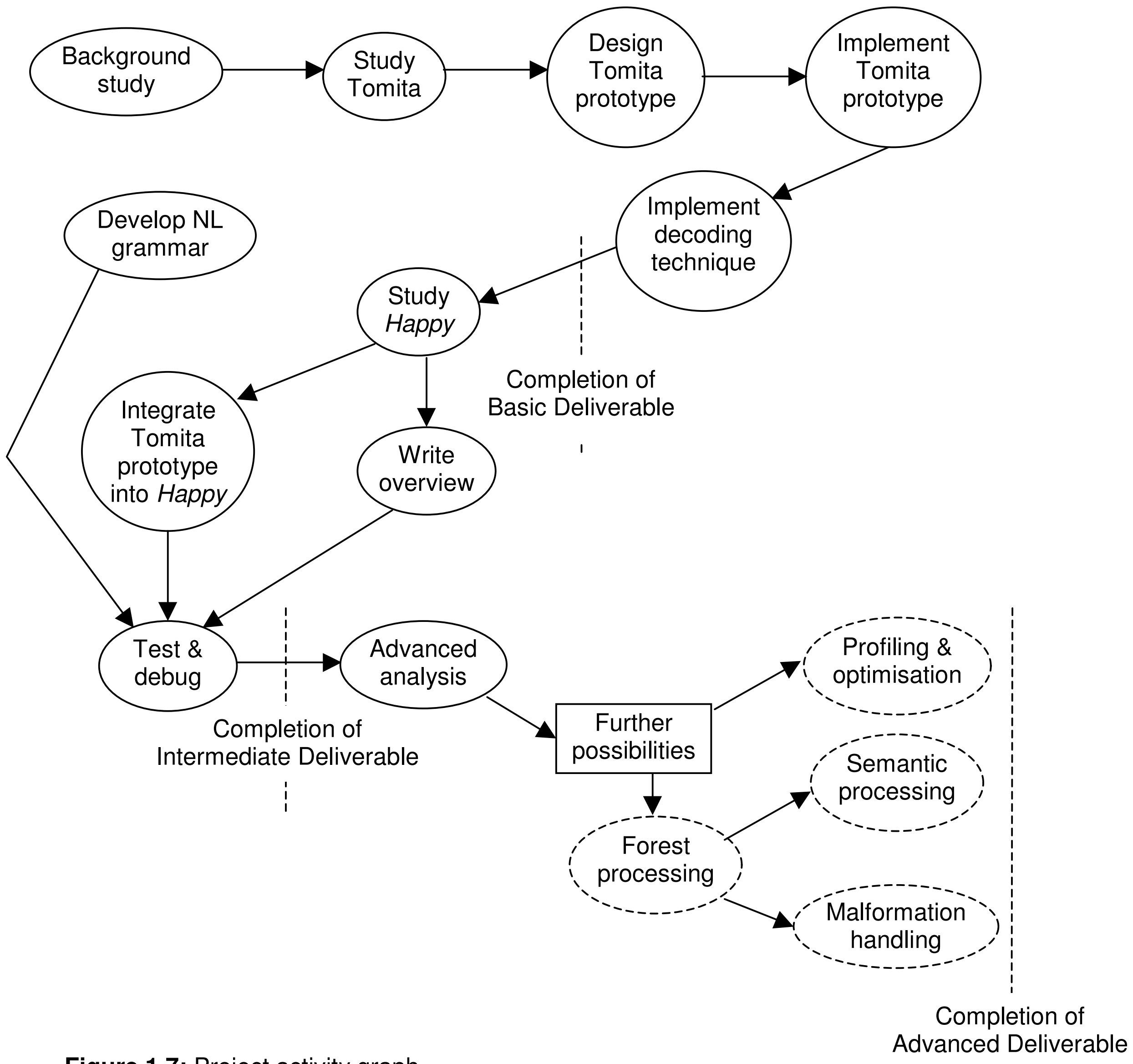


Figure 1.7: Project activity graph

1.5.2 Time Management Plan

The following table gives an outline of the projected time schedule for achievement of project tasks.

Task		Academic weeks
Background study		Summer vacation
Study Tomita		Summer vacation
Write Report	Implement Tomita prototype	1-3
	Implement decoding technique	3
	Study *Happy* & write overview	4-5
	Integrate prototype into *Happy*	6-7
	Develop NL grammar	8
	Test and debug	9-10
	Analysis	11-14
	Advanced possibilities	15-22

Table 1.8: Time management plan

1.6 Report

The remainder of the report is arranged in the following manner:

- *Chapter 2. Background and Literature Survey*: a guide to existing work in the subject area and related issues.
- *Chapter 3. Design*: initial design choices and issues raised.
- *Chapter 4. Implementation*: details of delivered material and related issues.
- *Chapter 5. Results and Evaluation*: analysis results and reflection on achievements vs objectives.
- *Chapter 6. Conclusion*: consideration of the positive and negative aspects of the project.

Ch 2. Background and Literature Survey

This chapter offers background information and a brief survey of existing material related to the areas of study associated with the project. The following topics are covered:

- LR Parsing
- Ambiguous Grammars, GLR Parsing and Tomita's algorithm
- NL Parsing
- Forest Processing
- Haskell

2.1 LR Parsing

Although the project relates specifically to the parsing of languages defined by ambiguous grammars, a number of the basic techniques employed are similar to those used in standard LR parsing. Thus we begin by looking at material relating to these techniques.

LR parsing of CF grammars is well understood and documented. Most sources approach the topic from either a mathematician's or programmer's point-of-view.

[Aho 86] provides thorough mathematical coverage of the LR parsing process. It looks at such key issues as *handles*, *lookahead symbols*, *stack-based shift-reduce parsing*, and dealing with *conflicts*. All terms and processes are formally defined without appeal to any particular implementation method.

The process of LR parsing is described less formally in [Hun 81]. Good use is made of diagrams to illustrate the concepts being discussed. The book provides step-by-step examples to illustrate the process of shift-reduce parsing, demonstrating how lookahead symbols are used to determine when a handle has been identified and reduction can take place. This source is based on work carried out at the University of Strathclyde on compiler design and implementation.

On the implementation side, [Hol 90] is a guide to writing compiler code in C. It covers all of the issues relating to LR parsing from a programmer's point-of-view.

LR parsing techniques have improved over the years and [Gru 00] provides up-to-date coverage of modern techniques, as well as an excellent section on error recovery.

2.2 Ambiguous Grammars, GLR Parsing and Tomita

2.2.1 Ambiguous Grammar Parsing

A significant amount of work has been carried out in the field of ambiguous grammar parsing, and a number of techniques have been proposed. One of the earliest is the *chart parsing* algorithm proposed by Martin Kay (1980) and based on *Earley's Algorithm*, a top-down predictive parser presented by J. Earley [Ear 70]. Initial chart parsing algorithms have been improved upon and variations have been introduced, such as left-corner (LC) and head-corner (HC) variants [Sik 93], many of which make use of CF grammars modified with additional components. Similar techniques have also been proposed, such as Lang's algorithm (1974) a technique for pseudo-parallel processing of non-deterministic push-down automata (NPDA, see [Aho 86]).

2.2.2 GLR Parsing and Tomita's Algorithm

Introduced in section 1.1.3, generalised LR (GLR) parsing is an approach to ambiguous grammar parsing proposed in the 1980's. It makes use of certain techniques associated with traditional shift-reduce LR parsing. A number of texts contain sections that introduce the concepts behind GLR parsing. For example, the following is taken from *Compilers – Principles, Techniques and Tools*:

> If we consider LR parsing tables in which each entry can contain several actions, we obtain non-deterministic LR parsing, often known as generalised LR (GLR) parsing. A kind of generalised LR parsing was proposed by M. Tomita in his paper *Efficient Parsing for Natural Language* (1986). He uses a *graph-structured stack* instead of a single stack in order to deal with multiple parses of a single sentence. [Alo 97]

A succinct description and analysis of Tomita's algorithm is found in John Carroll's thesis *Practical Unification-Based Parsing of Natural Language:*

> With this (Tomita's) algorithm, the LR table is allowed to contain multiple entries (i.e. action conflicts), perhaps caused by ambiguities in the grammar. Whenever the parser reaches a state in which there is an action conflict, the stack divides, and branches corresponding to the analyses for each of the actions are pursued, conceptually in parallel. Conversely, if separate analyses end up in the same state at the same point in the input string, the branches of the stack corresponding to the analyses are joined together at that state, and subsequent actions are applied to the single merged analysis, rather than to each branch separately….
> ….If parsing halts successfully, the analysis on the single arc in the graph-structured stack will be a packed parse forest, encoding all possible analyses of the input string. [Car 93]

Tomita's algorithm, produces a *packed forest* of trees, each representing a valid syntactic structuring of the input sentence under the specified grammar.

2.3 NL Parsing

Natural language grammars are a subset of the ambiguous grammars. NL parsing is a good test for any serious ambiguous grammar parser because of the immense complexity inherent within natural language.

The study of NL processing is a crossover between the fields of computer science and linguistics. Both are extremely interested in the possibilities presented by computational NL parsing. Christopher Butler gives a brief historical perspective:

> Almost as soon as it was understood that computers could manipulate symbols as well as numbers, a rush was on to translate texts automatically from one language to another. This enterprise led to some of the first parsers. Although the enthusiasm of the 1950s and 1960s was later dampened by the realisation that sophisticated analysis of meaning was also required, practical systems were produced which helped human translators perform the task more quickly. [But 92]

Many sources describe NL processing techniques. [Dow 85] looks at the major theories and approaches developed from the 1950s up until about 1985. Such techniques as transformational parsers, Augmented Transition Networks (ATN's), Logic grammars, Unification and Semantic guidance are discussed in [But 92], [Gri 86] and [Dow 85].

NL parsing and processing are active research areas, and a number of research groups around the UK have been involved with various related projects. One such project, called LOLITA, was carried out at the University of Durham. LOLITA stands for *Large-scale, Object-based, Linguistic Interactor, Translator and Analyser*. Further information can be found in Callaghan's PhD thesis [Cal 97]. Another group researching the field of NL processing operates from the University of Sheffield. They have been involved with a number of projects, including the development of an architecture known as GATE (General Architecture for Text Engineering) which they hope will be used as infrastructure for computational linguistics and language engineering. GATE is described in detail in [Gai 95].

2.4 Forest Processing

A possible direction for the advanced stage of the project is to consider how to efficiently process the forest that is the result of a successful parse. [Bil 89] looks at the structure of parse forests and how we might take advantage of them. The application of forest processing techniques varies depending on the type of data being represented.

2.4.1 Semantic Processing

Semantic processing is concerned with deriving *meaning* from syntactic structures such as parse forests representing NL sentences. This demands a large searchable knowledge base against which facts can be understood in context and inferences can be drawn. It presents an enormous challenge, to which no entirely satisfactory solutions have yet been proposed. Several prototypical approaches have been suggested for performing such tasks. For example, M. Schiehlen discusses a system for building semantic representations directly from shared parse forests such as those produced by Tomita's algorithm [Sch 96], and a number of projects have attempted to produce real-world systems that tackle the problem. A certain amount of success has been achieved in this area, although the level of syntactic and semantic complexity inherent in natural language makes it extremely difficult.

2.4.2 Malformation Handling

Another possible application of forest processing is in the area of handling malformed input from sources such as world-wide-web pages. No material was found that discussed this problem directly, but dealing with malformed HTML is a well-documented concern in web-browser design. In principle, we could allow malformations as part of the underlying grammar and then use a GLR algorithm to parse (possibly malformed) input into a forest. Efficient forest processing techniques and heuristic algorithms could be used to extract the most likely interpretation from the forest of possibilities. This may provide a more robust and efficient method of dealing with malformed HTML than is current employed in web-browsers.

2.5 Haskell

In this section we discuss some of the more advanced Haskell topics relevant to the project, pointing to resources containing further information.

2.5.1 Evaluation, Interpretation and Compilation

In an imperative programming language commands must be executed in *sequence*, because the use of variables and assignment induces the notion of *state*. As the program progresses, the state of the system may be modified, therefore sequence must be preserved for the result of the program to be consistent. Consider the following Java fragment:

```
int x = 1;
x = 2;
x = x + 1;
```

If the compiler did not uphold the sequence of these commands, the value of x might end up as either 1, 2 or 3.

In a pure functional language such as Haskell there is no inherent concept of state. A program is simply a functional expression, the result of which will necessarily be well-defined if the function itself is well-defined. An expression is evaluated by *reducing* it to its simplest, or *canonical* form. Evaluation in Haskell is *non-strict*. This is a form of *call-by-name* evaluation in which arguments to functions are only evaluated if they are required to get a result. For more information see section on compilation of functional languages in [Gru 00]. Non-strict evaluation gives rise to a novel style of programming. For example consider the following expression:

```
take 2 [0..]
```

The function `take` returns the first `n` elements of a list `L` where `n` is the first argument and `L` is the second. The construct `[0..]` generates an infinite list of integers, beginning at 0. In a strict environment, both arguments to `take` need to be evaluated, and the program would crash as it tried to generate the infinite list of integers. However, in Haskell only the first two elements of the list are generated, because the compiler/interpreter recognises that to evaluate `take` in this case requires only the first two elements of the list. Non-strict evaluation can be exploited in powerful ways to give elegant, modular implementations of algorithms and other programming paradigms.

2.5.2 Monads

Often when designing a solution to a problem it is necessary to maintain some data structure throughout all or part of a computation (state). Haskell does not have explicit state, and passing such structures through a series of functions can become somewhat complicated and messy. One solution is provided by *monads*. The concept of a monad is drawn from *category theory*, and they can be used in a functional language such as Haskell to hide the complexity of explicitly passing state values through the stages of a computation. Monads are a class of types providing methods by which we can string a number of actions together in *sequence*, passing values through the computation. A Monad must provide *bind* and *return* operations. *Bind* allows monads to be chained together to form a sequence of actions and *return* allows a value to be wrapped in a monadic type constructor. The following is a simple model of a chain of monadic actions with state:

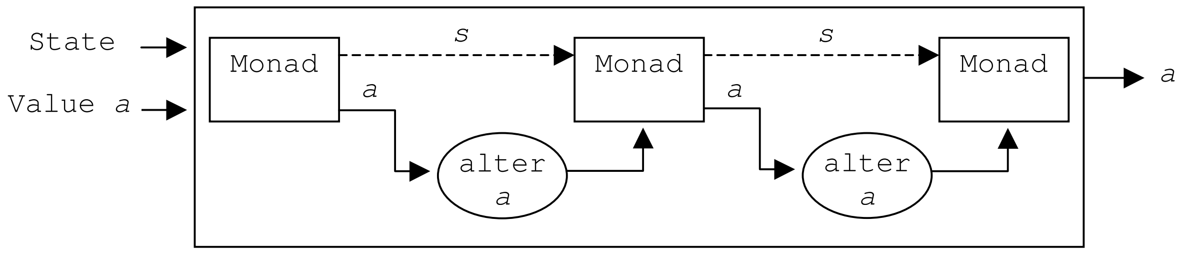


Figure 2.1: Chain of monadic actions

In this example the dotted line denotes an implicit passing of a *state variable*. We create the chain of monadic actions and pass in an initial state value along with an initial value for the result. Finally, the result of the entire computation is returned.

In fact, monads can be used to model a number of imperative programming paradigms, such as IO. Note that if we were to remove the state value from Figure 2.1, then we would simply have a *sequenced* chain of actions. This is how IO is handled within Haskell. See [Pey 00] for more information on the use of monads in Haskell.

2.5.3 Profiling

Analysis and evaluation of the implementation are important aspects of the project. Profiling tools provide effective methods of analysing Haskell code for time and space efficiency. The Glasgow Haskell Compiler[2] (GHC) provides two main profiling tools:

- Cost-centre profiling
- Heap profiling

Cost-centre profiling generates information about the sections of code responsible for the cost of evaluation in terms of time and memory allocation. It also shows a breakdown of the call hierarchy, listing the number of times a specified function was called and where it was called from. This is useful for isolating particularly expensive functions so that they can be optimised.

Heap profiling generates a graphical representation of the heap usage throughout execution of a program, stratified by individual functions. This is useful for identifying the memory behaviour of a program and eliminating space leaks. It should be noted that when dealing with non-strict evaluation it is difficult to predict how different parts of the code will behave. Optimisation is made much easier through the guidance of profiling tools.

[2]See http://www.haskell.org/ghc/

Ch 3. Design

This chapter looks at design issues associated with the project. We focus on the structure and operation of Tomita's algorithm. Initially, the traditional design of the algorithm as proposed by Tomita will be considered, followed by a discussion of the choices made in the design of the Haskell version of the algorithm for the project. A final section will be given to the extension of *Happy*.

3.1 Tomita's Algorithm – Traditional Approach

The traditional design model of Tomita's algorithm is based on imperative programming techniques, although the principles discussed are also relevant to a functional design model.

3.1.1 Parse Forest Structure

The output from a GLR parsing algorithm such as Tomita's is a collection of all valid syntax trees derivable from the input sentence under a given grammar. Thus, in terms of accuracy, it would suffice simply to return a list of these trees. However, the level of ambiguity present in many grammars, especially in the area of NL, renders this technique impractical due to the amount of memory and high cost of computation required to store and manipulate such large collections of trees. An example is given in Callaghan's thesis [Cal 97], in which an analysis of the sentence *"I own a car"* under LOLITA's wide-coverage English grammar produced over 13,000 possible syntactic interpretations.

Tomita proposed an efficient method of representing a collection of trees using a structure known as a *parse forest*. Two techniques are used to achieve this efficiency:

- Subtree sharing
- Local ambiguity packing

It is often the case in ambiguous grammar parsing that much of the substructure contained in distinct syntax trees is actually identical. Consider the *sentence "I saw Fred in the car"*. Two possible syntactic interpretations are as follows:

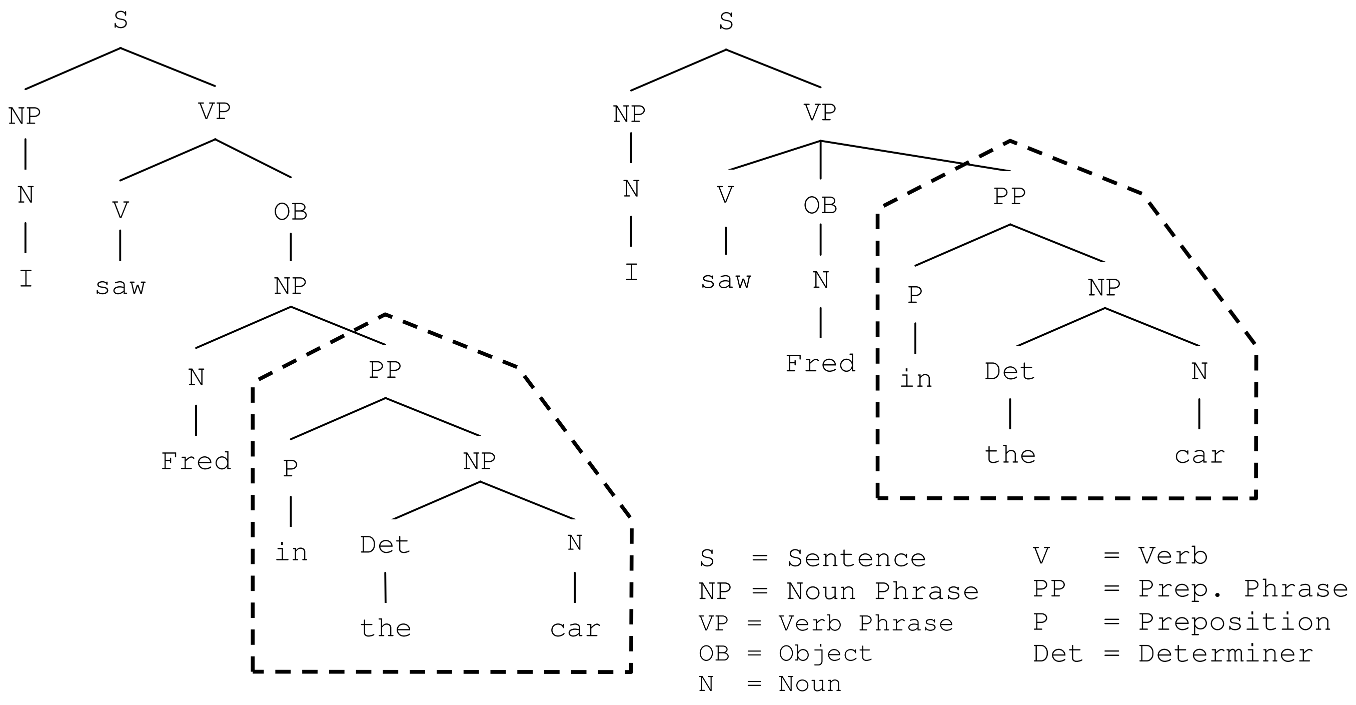
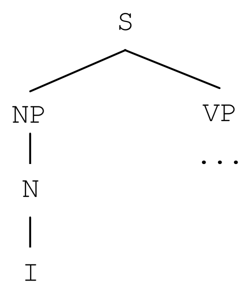


Figure 3.1: Identical structure in syntax trees

The first interpretation associates the prepositional phrase (highlighted) with the noun 'Fred', the second with the verb 'saw'. Note, however, that the structure of the prepositional phrase is identical in both interpretations. This can be exploited by a method known as *subtree sharing*, where identical constituents are represented by a single object.

Also note in Figure 3.1 that the top-level structure is identical in both trees:



This symmetry can be exploited by a technique known as *local ambiguity packing*, where distinct sub-nodes having the same category (VP in the case above) are 'packed' into a single node. This *packed node* can then be incorporated into the forest in exactly the same way as a standard node. Thus, the forest representing the sentence "I saw Fred in the car" incorporating subtree sharing and local ambiguity packing might be represented as follows:

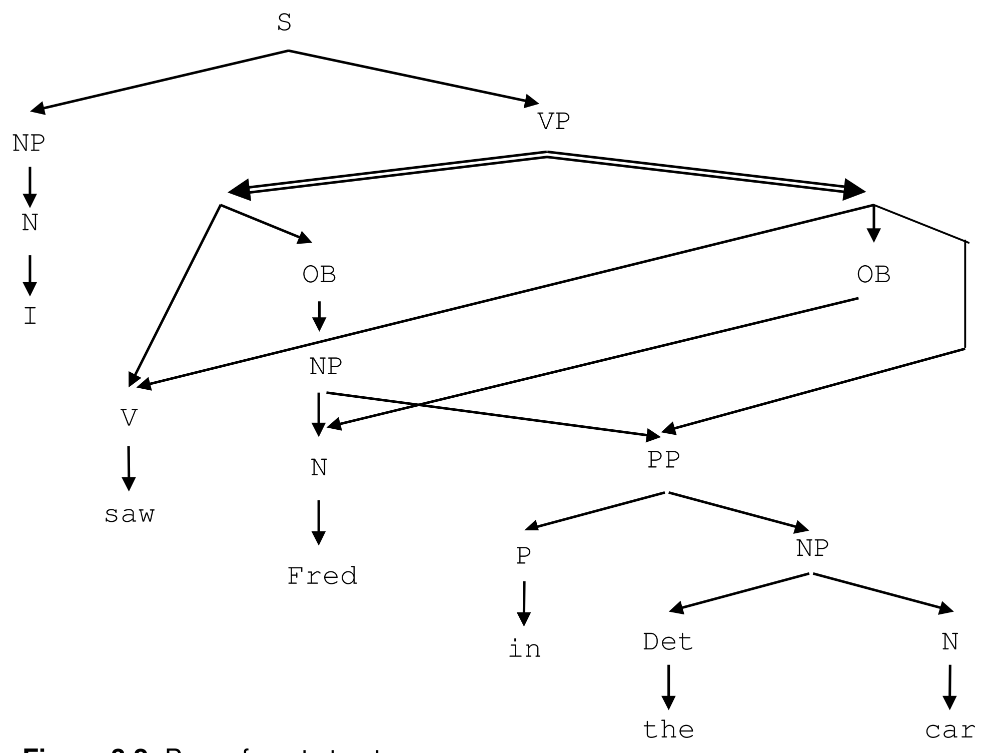


Figure 3.2: Parse forest structure

The double lined arrows denote that the node labelled `VP` is a packed node representing two analyses. Shared subtrees are those with more than one arrow pointing to them. Note that technically, a forest is a directed, acyclic graph (DAG).

3.1.2 GSS Structure (Traditional)

The GSS (Graph Structured Stack) is a data structure fundamental to the design of Tomita's algorithm. It can be conceptualised as a parse stack, offering the common shift and reduce operations. However, it is based on a graph rather than a list. The graph is a DAG, with vertices representing stages in the parse, connected by arcs. Each vertex contains a state number and each arc contains a forest structure representing an analysis of the input consumed thus far.

3.1.3 GSS Operations (Traditional)

As stated earlier, the GSS provides the shift and reduce operations associated with a traditional parse stack. In standard shift/reduce parsing, the operation is always applied to the element on top of the stack; in a GSS, the operation is applied to a particular vertex v, representing an element on top of the stack. There may be multiple elements on top of the stack at any one time. The semantics of the operations are as follows:

- *shift(v)*: a new arc is created, extending from *v,* containing a forest representing an analysis of the token of input being shifted (a single forest *leaf*), linked to a new vertex labelled with the new state, as specified by the *action* table.
- *reduce(v)*: arcs extending from ancestor vertices at a distance of *n* arcs from *v*, where *n* is the number of symbols in the right-hand side of the nominated reduction rule, are removed from the graph, along with the redundant vertices. A new arc is created from each of these ancestor vertices, containing a new analysis consisting of a forest node labelled with the left-hand side of the production rule, whose children are the forests contained on the arcs leading up to *v*. Each of these new arcs is then linked to a new vertex, labelled with the state specified in the *goto* table.

Subtree sharing is incorporated into the GSS in the abstract sense that if two arcs contain identical analyses of part of the input, then a single forest representing the analyses is shared between the arcs. This sharing is then propagated into the resulting parse forest as the analyses are assembled through reductions.

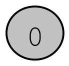
Local ambiguity packing occurs when more than one arc exists between two vertices. This represents a state in which the parser has branched (due to a conflict) and subsequently reduced the branching analyses into the same top-level node. If this is the case, then the analyses can be packed into a single node, and the multiple arcs replaced with a single arc containing the packed forest node.

The parser is initialised with a single vertex representing the initial state (usually 0). The parse then proceeds in accordance with the actions specified in the parse table. When a conflict is identified, all conflicting actions are performed on the vertex in question, conceptually in parallel. If the branching analyses arrive concurrently at the same state, they are joined into a single vertex.

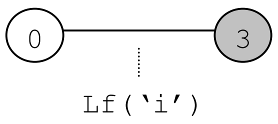
3.1.4 GSS Example (Traditional)

An example will help to clarify these concepts. Consider the following parse of the sentence "`i + i + i $`" under Grammar 1.5 (section 1.1.3), with its associated parse table (Table 1.6), using a traditional Tomita parsing algorithm ($ is the EOF symbol):

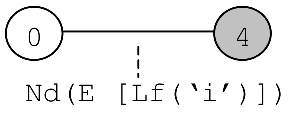
First the GSS is initialised in state 0 (vertices on 'top' of the stack are shaded):



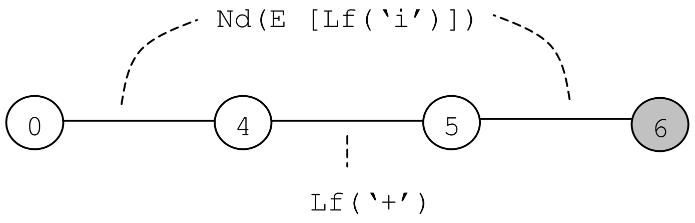
The first token in the input is '`i`' and the corresponding action in state 0 is '`sh3`'. An arc is created from the original vertex to a new vertex representing state 3, containing a forest leaf labelled with the shifted token of input.



The next token of input is '+' and the corresponding action '`re2`'. The reduction is carried out, and the GSS develops as follows:

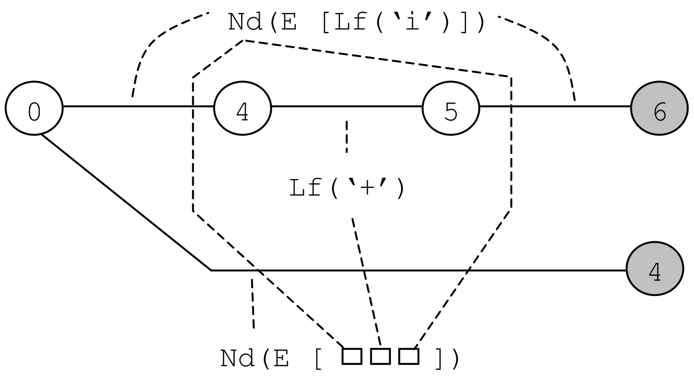


The parse continues deterministically until the fourth token of input is encountered. The GSS at that stage is as follows:

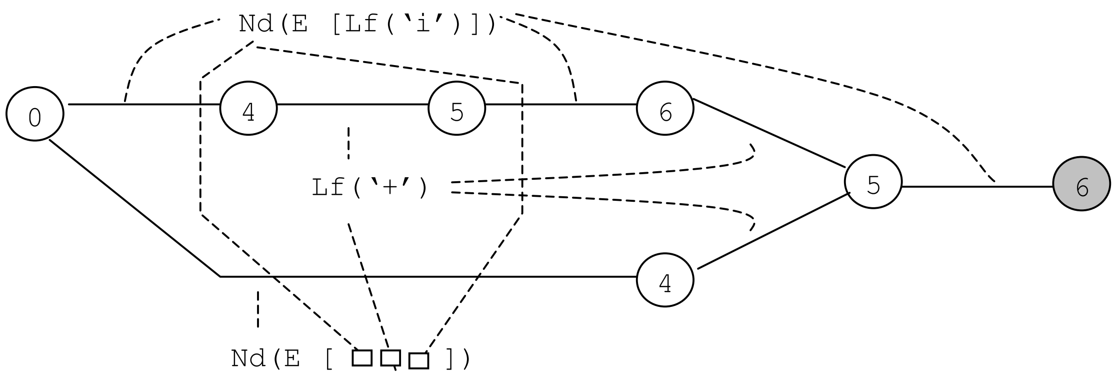


Note that we don't distinguish the first `'i'` from the second - it is only stored once as a shared subtree between the two arcs (0,4) and (5,6). Information about the position of a lexeme in the input sentence is not stored explicitly, although it can be retrieved from higher-level forest structure if required.

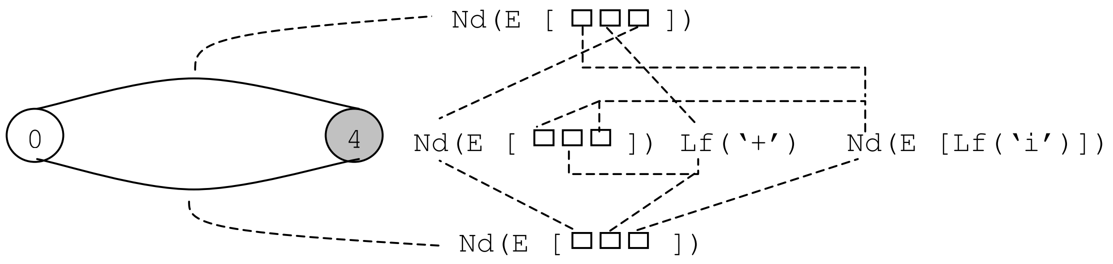
The action for a '+' token in state 6 is '`sh5/re1`'. All reductions are performed before any shifts, leaving the GSS in the following state before the shifts:



Note that there are now two nodes on 'top' of the stack, and also that the forest being built up along the lower arc incorporates subtree sharing. The shift action to be performed on both top nodes is '`sh5`' allowing the two analyses to be united into a single vertex. The final '`i`' is then shifted and reduced to an '`E`', leaving the GSS as follows:



A reduction on the top vertex (state 6) and subsequent reduction on the newly created vertex (also state 6) leaves the GSS in the following state:



Note that there are now two arcs, both extending from the same vertex and conjoining in the same state, containing separate analyses of the input parsed thus far. These analyses can therefore be packed into a single analysis with the category '`E`'. The next action is an '`accept`' on the top vertex (state 4), and the parse terminates with the GSS in the following condition:

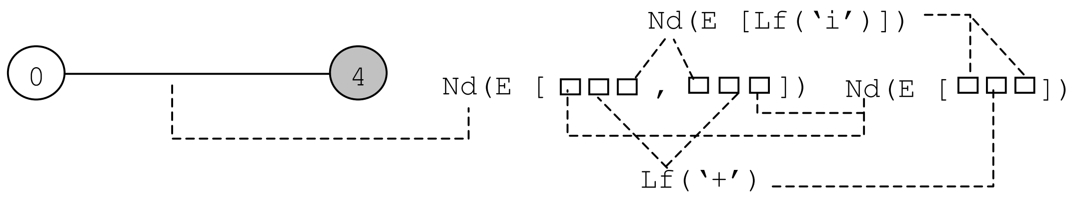


Figure 3.3: Traditional GSS example

If a parse is successful, the final state of the GSS will always be a single arc extending from the initial vertex to the final accepted vertex. The forest contained on this arc represents all valid parses of the input (in this case the two ways of associating the + operator) and is returned as the result of the successful parse.

3.2 Tomita's Algorithm – Functional Approach

Having considered the design of Tomita's algorithm from a traditional viewpoint, we now propose a design model for a functional implementation.

3.2.1 Forest Structure (Functional Model)

As previously discussed, a forest is technically a DAG. We can represent such a graph structure by a list of nodes, each mapped to by a unique key value. Within each node is stored a list of the key values of that node's children.

In a parse forest, each node is labelled with a grammar symbol. Nodes with one or more children are labelled with *non-terminals*; Nodes with no children (leaves) are labelled with *terminals*. A *packed node* contains a list of two or more alternative analyses, each one with its own list of children. The top node in the forest must be specified so that it can be identified among the other nodes in the mapping. The representation can be stated more formally as follows:

A parse forest (under grammar G) is:

a binary relation $PF = \{(k_1,nd_1),...,(k_n,nd_n)\}$

where

$k_i\ (1 \le i \le n)$ is a value of arbitrary type such that
$\forall\ k,nd,nd'\ .\ (k,nd)\in PF \wedge (k,nd')\in PF \Rightarrow nd=nd'$ (*uniqueness of domain values*)

and nd_i is a forest node such that:

$nd_i\ (1 \le i \le n) = (gs,[a_1,...,a_m])$ (gs – *grammar symbol*)

where

$gs \in (terminals(G) \cup non\text{-}terminals(G))$

and a_i is an analysis such that:

$a_i\ (1 \le i \le m) = [c_1,...,c_j]$

where c_i is a child node key value such that:

$c_i\ (1 \le i \le j) \in domain(PF)$

We should further stipulate that the forest contain no duplicate nodes:

$\forall\ k,k',nd\ .\ (k,nd)\in PF \wedge (k',nd)\in PF \Rightarrow k=k'$ (*uniqueness of range values*)

Note that the properties of the relation are those of an *injective function*.

3.2.2 GSS Structure (Functional Model)

In Tomita's original specification, the GSS is designed and implemented as a DAG. We could represent the GSS, then, in the same way as the forest discussed in the previous section, using an injective functional mapping of key values to vertices. However, we should consider that once constructed, a forest node need not be revisited (except for equality testing & packing), whereas vertices in the GSS require extensive updating as the parse progresses. If a mapping such as the one suggested above were used to represent the GSS, a vertex that required updating would have to be located, removed, broken down, rebuilt and reinserted into the map – a complicated and computationally inefficient procedure. Moreover, as reduce actions are performed on the GSS, certain sections become redundant. If the graph is represented as a map, redundant arcs and vertices must be explicitly removed to maintain storage efficiency. This requires extra computation that we wish to avoid in a language such as Haskell which has its own garbage collection process.

Peter Ljunglöf has carried out some work on designing a functional version of Tomita's algorithm. In chapter 6 of his licentiate thesis [Lju 02] he proposes a data structure for the GSS which is suited to a functional programming environment.

Rather than an explicit DAG, Ljunglöf suggests representing the GSS as a collection of trees, where each tree represents a parse stack with a distinct top-level state. When a conflict occurs, the stack is split into two identical stacks and both operations are performed, conceptually in parallel. Although it appears that two stacks have been created in the place of one, the identical lower sections of each stack are stored internally as single objects. This means that the data structure exists in memory as a DAG, even though it appears as a collection of trees to the programmer. This concept is illustrated by the following diagram:

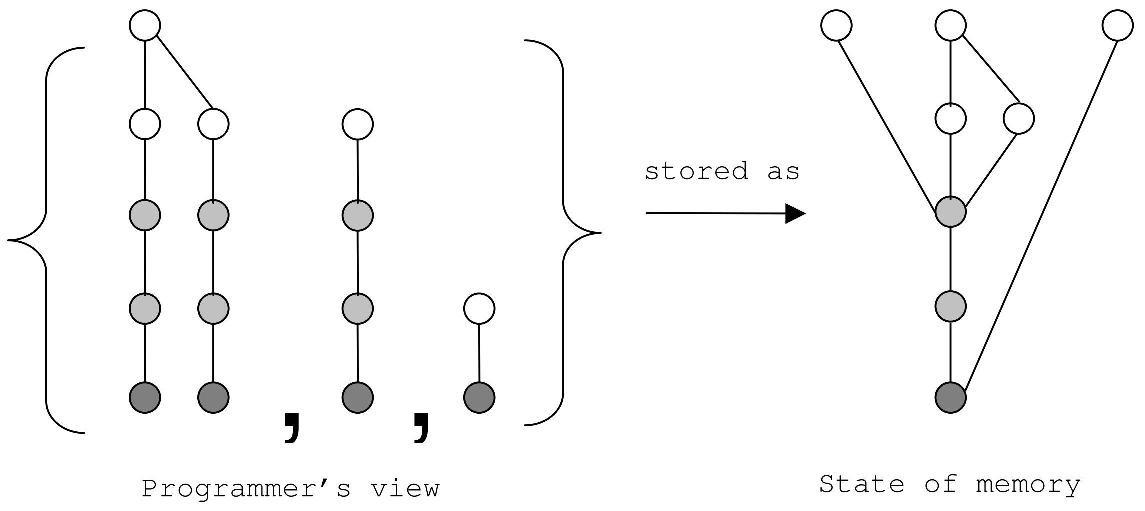


Figure 3.4: Collection of trees stored internally as a DAG

All objects in Haskell are stored internally by reference, thus when a stack is duplicated, its pointer is copied and the stack itself remains as a single object.

By representing the GSS in this form, we no longer have to worry about working with an explicit graph structure. Much of the computation can be performed on trees, a data structure well-suited to the functional environment. Furthermore, the collection of trees can be represented as a list, allowing the use of well-established functional techniques for working with lists of objects.

It should be noted that although we borrow the basic idea for the GSS representation from Ljunglöf, other aspects of the design model, such as the parse forest structure and main driver are entirely different.

3.2.3 GSS Operations (Functional Model)

We now consider the design model for GSS operations, in light of the new representation.

The top-level node of each tree in the collection corresponds to one of the vertices on top of the stack in the tradition GSS model. Thus, we apply *shift* and *reduce* operations to top-level nodes as we would apply them to top vertices in the traditional GSS. If the same state is reached concurrently by two separate trees, their top-level nodes are *merged* to form a single tree.

Subtree sharing is incorporated into the new GSS model by use of parse forest node indices. Because the parse forest is represented as an injective functional mapping of key values to forest nodes, the index values act as pointers to each distinct sub-analysis and are used to represent sharing between different sections of the GSS.

Local ambiguity packing occurs when the parser reaches a state in which two trees are identical except for their top-level analyses, in which case their top-level analyses can be packed and the trees combined:

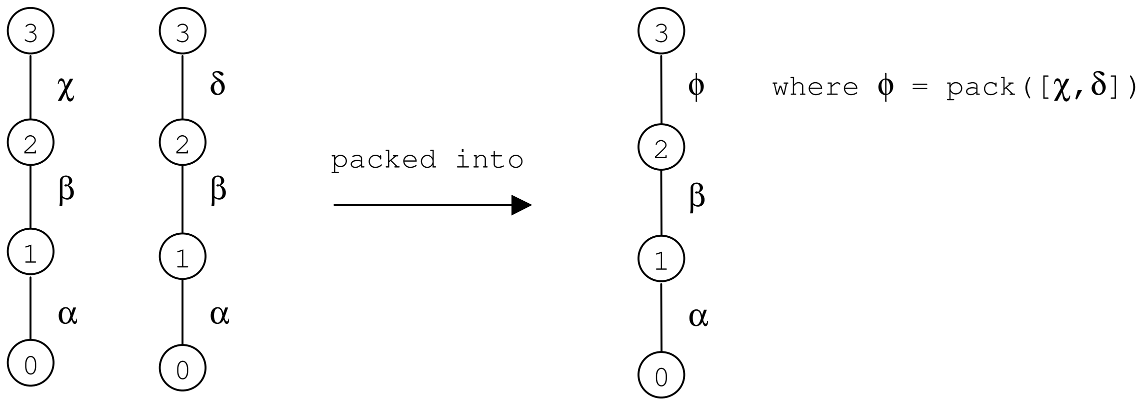
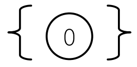


Figure 3.5: Local ambiguity packing in functional GSS model

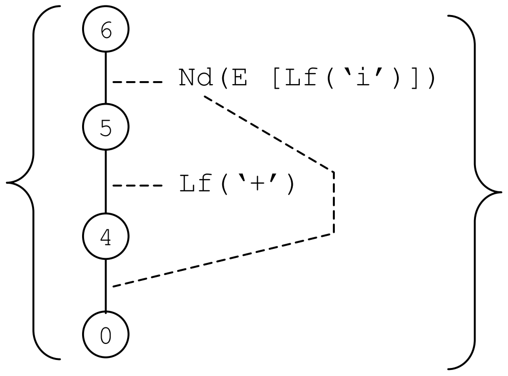
3.2.4 GSS Example (Functional Model)

As an example, let us take the same grammar, sentence and parse table used in Figure 3.3, and demonstrate the parsing process under the new GSS model (nodes shared at memory level are shaded):

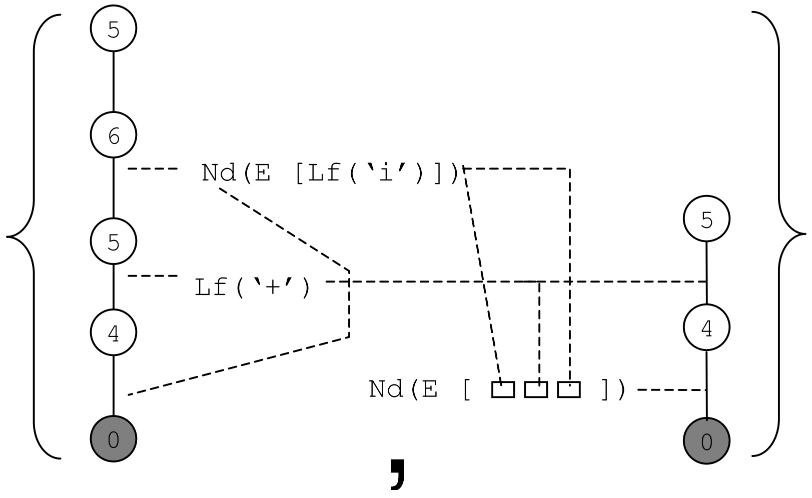
Firstly the parser is initialised with the collection containing a single tree with a single node in state 0 (the initial state):



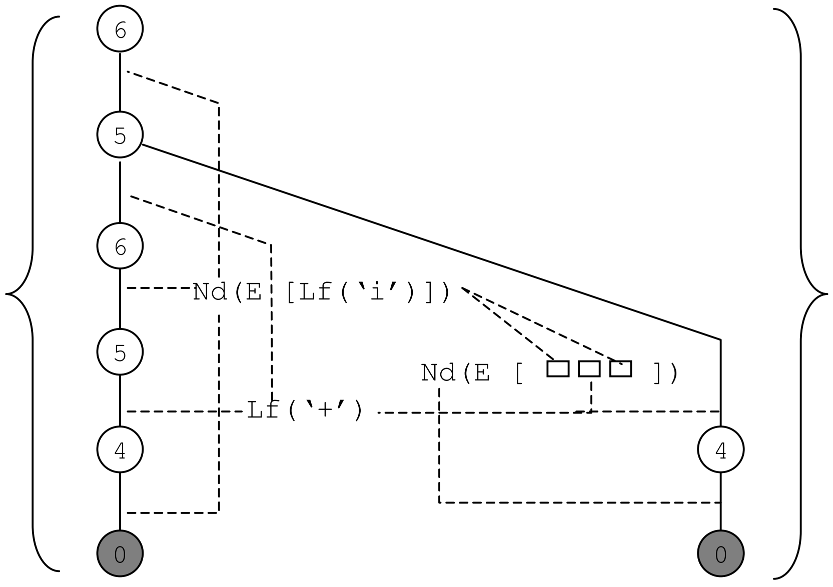
The parse then continues deterministically until the conflict on the fourth token of input:



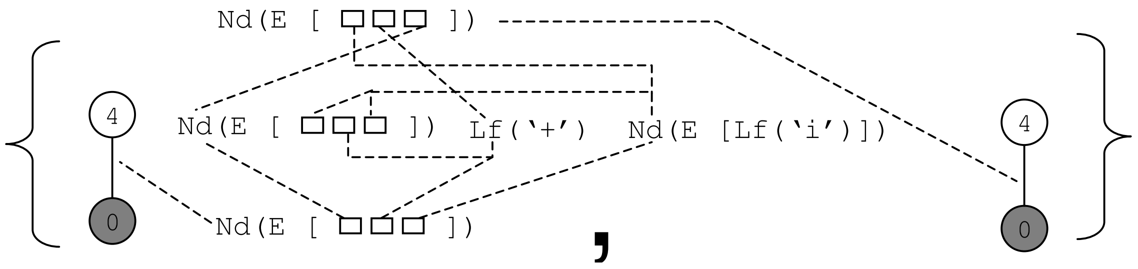
At this point, the conflict is dealt with by duplicating the tree and performing both shift and reduce actions. The result is as follows:



Note that the top nodes now both have the same state, and can therefore be merged, combining the two trees. The final `'i'` is then shifted and reduced to an `'E'` to leave the GSS in the following state before the EOF symbol:



Performing the next two reductions leaves the GSS as follows:



Note that the collection now contains two identical trees, bar their top-level analyses. The means that packing can occur, before the final action is an accept on the top node (state 4) of the remaining tree:

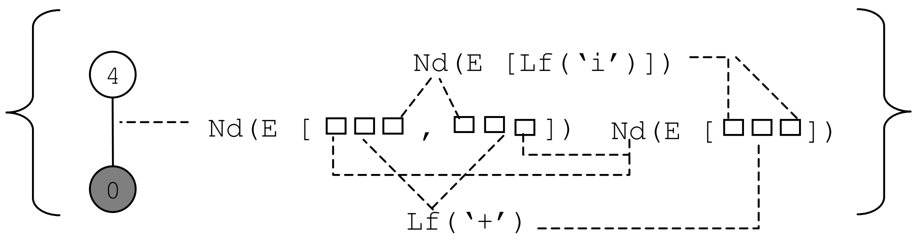


Figure 3.6: Example of tree-collection GSS parse

Importantly, the resulting forest is identical to that produced by the traditional GSS model (Fig 3.3).

3.3 High-Level Structure

In this subsection, we consider the design of the high-level structure in the functional model of Tomita's algorithm.

3.3.1 State Handling

The high-level structure of the algorithm suggests some use of *state*. For instance, the forest must be grown as the parse progresses, requiring regular equality testing on its nodes, as well as various structural updates. The monadic framework within Haskell is a good mechanism for handling such issues. This is discussed in detail in Chapter 4, but for now it will suffice to note that the parse forest can be thought of as a state value, available throughout the computation.

3.3.2 Driving the Parser

We now consider the design of the parser *driver*, or the method by which a string of input tokens is converted into a parse forest.

As each token of input is encountered, the contents of the top of the GSS are examined and the relevant actions performed, as directed by the parse table. If there are no more tokens, there should be a single element on top of the stack in the *accept* state, and the parse terminates. Note that the last input token is always the EOF symbol, as in standard LR parsing (see 1.1.2). In pseudo-functional code, this might look as follows:

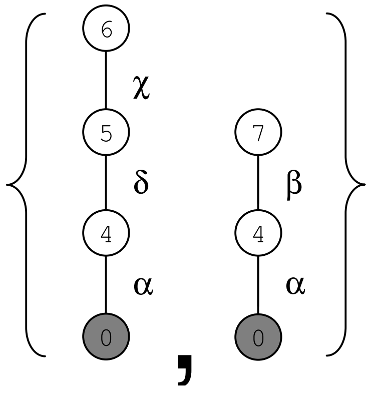
```
doParse []          gss = returnForest
doParse (tok:toks) gss = doParse toks (doActions tok gss)

(doActions is a function that performs the relevant actions
 on the top elements of the GSS and returns a new GSS)
```

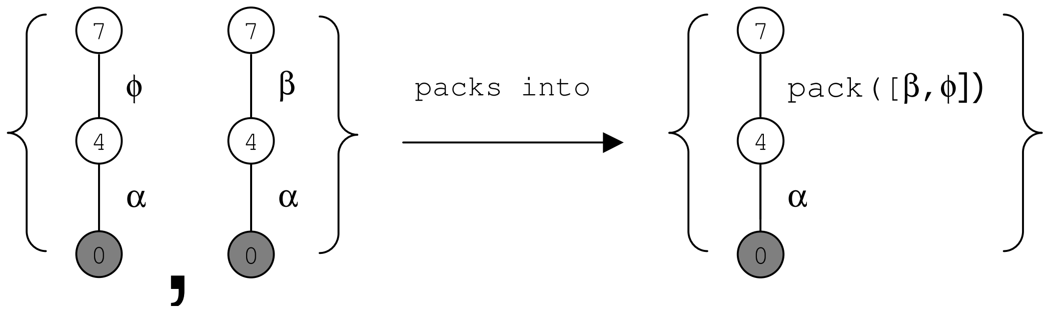
Figure 3.7: Top-level GLR parsing process

The most complex aspect of the GLR parsing process is represented by the `doActions` function in the illustration above. Essentially, every top-level node in the GSS must be examined and the relevant action performed as directed by the parse table. For shift actions a single *shift* operation is performed on the relevant node. Reduce actions are more complicated though, as performing a *reduce* operation may leave the GSS with new top-level nodes which must also be examined. This requires a closure operation that iteratively reduces top-level nodes.

Performing reductions is complicated further by the fact that every time a reduction is performed, we must check to see if local ambiguity packing can occur. Allowing packing to occur at the earliest possible stage requires us to reduce trees in reverse height order. Consider the following example:



Reducing the first tree by the production rule '$\phi \rightarrow \delta\,\chi$' with goto state 7 will lead us to a position where packing can occur:



However, if the second tree were reduced before the first (say by the rule '$S \rightarrow \alpha\,\beta$' with goto state 2) this opportunity to pack would be missed, leaving us in the following position:

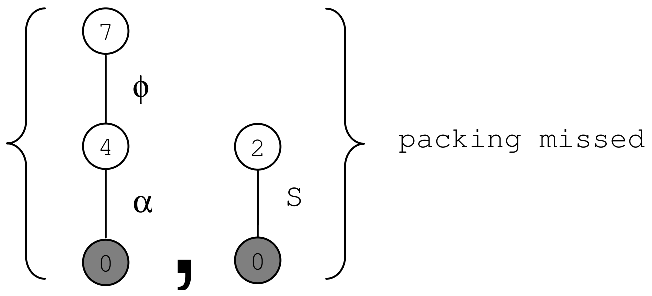


Figure 3.8: Necessity for reverse height-ordered reducing and packing

Similarly, if we applied reductions across the whole collection without packing between each one we would end up in the same state.

Thus, the tallest tree must always be reduced first, and the new trees created by the reduce operation must be introduced into the collection before the next reduction is considered.

We now consider conflict handling. The representation chosen for the GSS makes it relatively easy to handle conflicts when they arise. We simply duplicate the tree upon which the conflict has arisen and ensure that both actions are performed. As discussed earlier, the underlying mechanics of Haskell handle the sharing of the lower sections of the trees. We will define some type for actions returned by the parse table, for example:

```
data GLRAction
 = Reduce [Rule]          -- one or more reductions
 | Shift State [Rule]     -- shift & one or more reductions
 | Accept                 -- accept
 | Error                  -- blank entry
```

Figure 3.9: GLRAction data type

Note that `Reduce` and `Shift` can represent either single actions or Reduce/Reduce and Shift/Reduce conflicts respectively.

Tying these factors together, we can now specify a structure for `doActions` from Figure 3.7 *(assume GSS is a list of trees reverse-ordered by height)*:

```
doActions tok = (shiftAll tok).(reduceAll tok)


reduceAll tok []       = []
reduceAll tok (tr:trs) =
 case getAction tr tok of
  Reduce rs   -> reduceAll tok $ pack (reduce tr rs) trs
  Shift st rs -> (tr,st) ++ reduceAll tok $ pack (reduce tr rs) trs
  _           -> reduceAll tok trs
 where
  action tr tok = function that looks up the entry for the top state of tr and the current
                  token tok in the parse table

  reduce tr rs = function that performs all specified reduce operations rs on tr and
                 returns the resulting list of new trees

  pack ts ts' = function that converts ts and ts' into a single list of (reverse height
                ordered) trees, performing packing where possible


shiftAll tok gss =
 [ shift tr st | (tr,st) <- gss ]
 where
  shift tr st = function that performs shift on tr, returning the new tree with top state st
```

Figure 3.10: Structure of `doActions` and auxiliary functions

For each tree in the GSS, `reduceAll` looks up the relevant action in the parse table for its top state and the current token. If `Reduce` is specified then the reduction is performed, the new trees are packed into the existing GSS and a recursive call to `reduceAll` is made. If `Shift` is specified then again all reductions are performed, but this time the current tree is paired with its shift state and concatenated onto the result of the recursive call. If `Accept` or `Error` is specified, then we simply *let go of* the current tree and continue, as its related derivation (erroneous or complete) will already be stored in the forest (see next section for discussion of error handling). `shiftAll` then maps the shift operation over all remaining trees in the GSS, now paired with their respective shift states.

A useful addition to the parser driver is the ability to supply as input a string of *sets of tokens*, rather than just individual tokens. For example, in the area of NL parsing a single lexeme is often associated with a number of different parts of speech, i.e. the word '*park'* can be either a noun or verb. To illustrate this concept, input might be structured as follows:

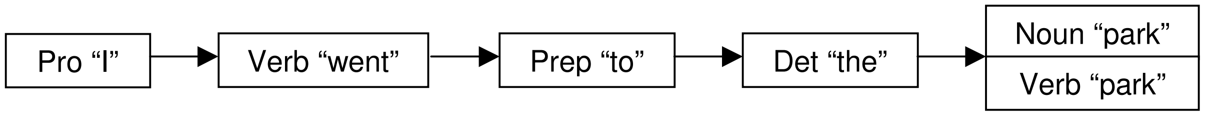


Figure 3.11: Input token structure

This can easily be incorporated into our design model. When a set containing more than one token is identified at some stage in the input, we simply duplicate the parse stack (list of trees), perform the relevant actions on both stacks and merge the results. In the example above, the analysis of the sentence with 'park' as a verb will result in an error.

3.3.3 Error Handling

An error occurs in the GLR parsing process when there are no entries in the parse table for a specific token/state index. In standard LR parsing, an error occurs in exactly the same way and the required determinism of the process signifies a parse failure. However, in GLR parsing an error only signifies a failure to parse one of possibly many derivations. A complete parse failure occurs only if the process fails to produce any valid derivations. Thus, we divide potential errors into two categories:

- *Partial* errors, signifying failure to parse a particular derivation.
- *Total* errors, signifying failure to parse any valid derivations.

More concretely, a partial error occurs when the parse table entry for the top state of a tree in the collection under the current token specifies an `Error`. When a partial error is identified, we can either ignore it and continue searching for a valid derivation, or store information about the tree that caused the error, along with the current token, and return it at the end of the parsing process. A total error occurs when the parsing process arrives at a state in which the GSS is empty; i.e. there are no more valid sub-derivations that can be pursued. In this case, the process should report a complete parse failure.

3.4 Integration with *Happy*

The chapter concludes with a look at the design issues associated with the integration of the functional version of Tomita's algorithm with the Haskell parser-generator tool *Happy*.

3.4.1 Analysis of *Happy*

Happy is an LALR(1) parser-generator tool, written in Haskell. It takes as input a file specifying a context-free grammar and generates an LR parser using LALR(1) table generation technology. If successful, a new Haskell file is created containing the parser, which can then be compiled either as a stand-alone program or incorporated into some larger system. As well as the grammar, the input file requires further information to be specified about the resulting parser, such as the type of tokens in the language and the lexer function which converts strings of text into tokens. *Happy* is executed directly from the command prompt, and has a number of options that can be specified by setting various flags. The following diagram illustrates the high-level internal structure of *Happy*.

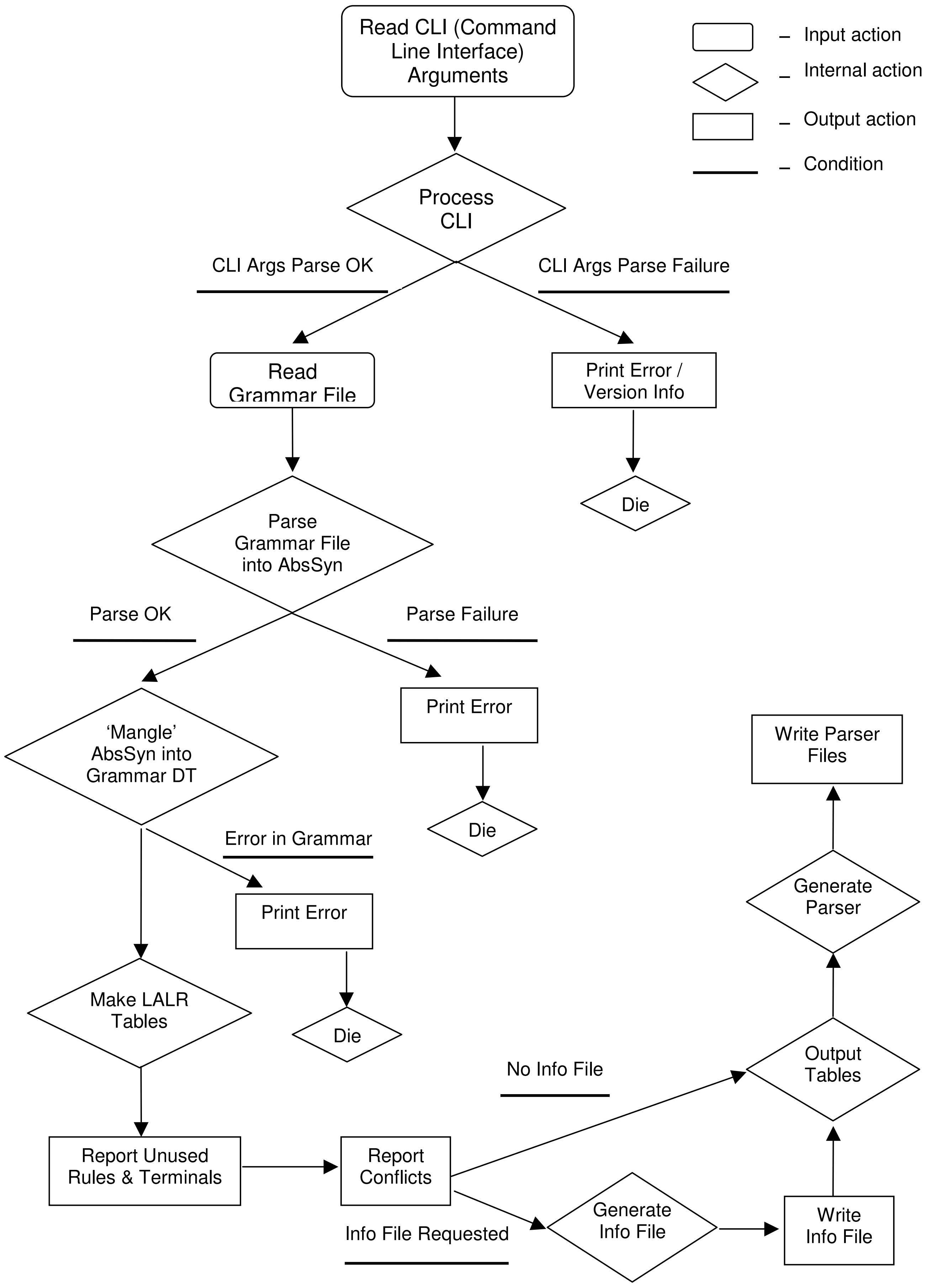


Figure 3.12: High-level internal structure of *Happy*

3.4.2 Integration of GLR component

The project integrates the functional implementation of Tomita's algorithm with *Happy*, allowing it to generate GLR parsers for languages derived from ambiguous CF grammars. This is carried out by incorporating a section of code into *Happy* that produces a GLR parser based on a template containing the functional implementation of Tomita's algorithm. The resulting GLR parser uses the LALR parse tables generated by *Happy*. A new flag is added, allowing the user to specify which type of parser to generate (standard LR or GLR). The integration involves altering the high-level structure of *Happy* at the following point:

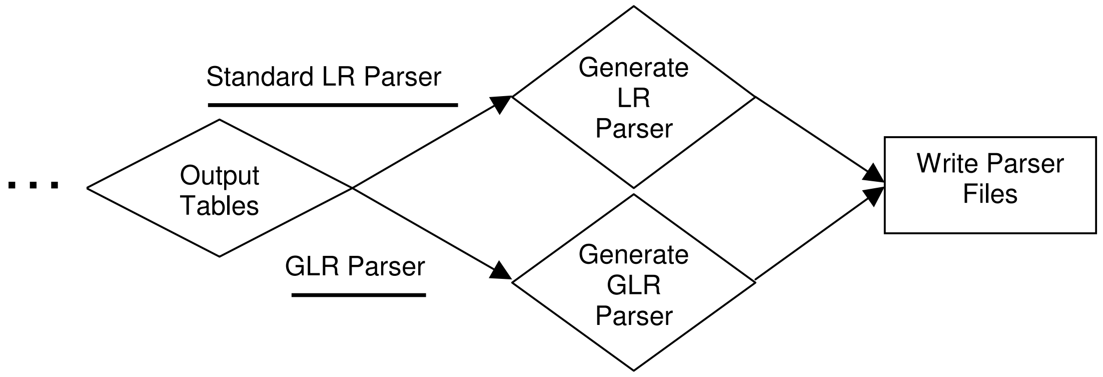


Figure 3.13: Amendment to internal structure of *Happy*

The rest of the internal structure of *Happy* remains unchanged. Further details about the modifications made are found in Chapter 4.

3.4.3 Transparency

An important aim of the integration is *transparency*. That is, as far as the user is concerned the process of creating a GLR parser should be the same as for a standard LR parser, the only difference being the setting of the flag governing which option to take. Thus, as many as possible of the original options and features found in the standard LR version of *Happy* are preserved in the extended version. This includes such features as user-definable token types and the incorporation of additional module declaration code, as well as the ability to output information about the parse table structure.

A feature available in the LR version of *Happy* that is not yet incorporated into the GLR extended version is the embedding of production-translation code into the parser. It is possible to convert a parse tree into a Haskell expression as the parse progresses, by providing code that is evaluated along with each reduction. See [Mar 00].

This is relatively straightforward in deterministic LR parsing, but becomes much more complicated when dealing with non-determinate GLR parsing, where lower level sub-derivations exist in possibly many higher-level contexts (arising from ambiguities in the grammar). Thus, the translation required on part of a sentence cannot always be carried out without knowing the context in which it exists. In fact, many different translations may be required on a single sub-derivation. This is evident in the area of NL parsing, for instance, where a part of a sentence is often meaningless without appealing to the context of the whole. A possibility might be to apply *context-insensitive* translations to relevant sections as the parse progresses, and postpone context-sensitive processing until higher-level sections have been parsed. Further discussion is beyond the scope of the project, but would be an interesting area of research.

Ch 4. Implementation

The aim of this chapter is to provide information about the implementation of the design models described in chapter 3. We consider the implementations of the three main components of the GLR parser based on Tomita's algorithm:

- The data type representing parse forests.
- The data type representing the tree structures that comprise the GSS.
- The parser driver.

We also consider some of the optimisations that were carried out on the implementation to increase its efficiency and to outline the modifications made to *Happy*. The chapter concludes with a description of the implementation of the technique used to decode parse forests into their component trees.

4.1 Parse Forest Data Type

In this subsection we consider the structure and operations of the data type implementing the parse forest model described in section 3.2.1.

4.1.1 ForestNode and SetMap Structure

The internal structure of a parse forest is that of a DAG. We have seen that this can be represented as an injective functional mapping of key values (henceforth referred to as *indices*) to forest nodes (3.2.1). The two main components of the model are:

- A data structure representing forest nodes, stored as elements in the map.
- A data structure representing the map itself.

We implement the forest node data structure in Haskell as an algebraic data type:

```
data ForestNode a = FNode a [[Int]]
```

Where the arbitrary type variable `'a'` represents the grammar category associated with the node, and the list `[[Int]]` represents a (possibly empty) packed collection of analyses, each consisting of a list of forest node indices.

The injective functional map is implemented as a new ADT known as a *SetMap*. The basic type constructor is:

```
data SetMap a = SM [Int]        -- unused indices
                [Element a]     -- the relation

data Element a = EL (Int,a)     -- (index,value) mapping
                 Int            -- uses of element
```

Elements of the relation are pairs of indices and values, where indices are of type `Int` and values are of arbitrary type. Unused indices are stored as an infinite list of fresh `Int` values (see 2.5.1 on non-strict evaluation). Each element also has an `Int` field representing the number of times it has been used by a client (see 4.1.2 for details). The type system in Haskell isn't strong enough to enforce the injective functional property of the relation, but we can handle this behind the ADT barrier.

4.1.2 SetMap Operations

The SetMap data type implements a number of operations. They are described as follows:

- `initSM :: SetMap a`
  The relation itself is initialised as an empty list, and the list of unused indices is initialised as an infinite list of integers, beginning at zero.

- `addElem :: Eq a => a -> SetMap a -> (Int,SetMap a)`
  The relation is searched for values matching the value to be added; if such a match is found, the `SetMap` is returned unchanged, paired with the matching value's index. If no match is found, a fresh index is taken from the head of the *unused indices* and paired with the new value, then added to the head of the relation. The amended `SetMap` is returned, paired with the new index.

  When a new element is added, its *number of uses* field is initialised to 1. If an element already exists for a given value, its *number of uses* is incremented.

- `getElem :: Int -> SetMap a -> Maybe a`
  If the element corresponding to the given index is found in the relation it is returned, wrapped up in the `Maybe` data type, using the `Just` constructor, otherwise the `Maybe` value `Nothing` is returned.

- `decElem :: Int -> SetMap a -> (Maybe a,SetMap a)`
  If the element corresponding to the given index is found in the relation its *number of uses* field is decremented. When this reaches 0 the element is removed from the relation and the newly-freed index is appended to the head of the *unused indices* list. The element's value is then returned, wrapped in `Just`, paired with the amended `SetMap`. If the element corresponding to the given index does not exist in the relation, then `Nothing` is returned, paired with the unchanged `SetMap`.

A full listing of the `SetMap` operations is provided in Appendix A.

4.2 TStack Data Type

In section 3.2.2 the GSS was modelled as a collection of trees. The project implements these trees as a new ADT known as a `TStack`[3]. The following subsection looks at the structure and operations of the TStack ADT.

4.2.1 TStack Structure

A `TStack` is an ADT with the following type constructor:

```
data TStack a = TS Int                -- state
                   [(a,TStack a)]   -- [(element on arc , child)]
```

The first field in the constructor is an `Int` value representing the state of the top-level node in that TStack, and in standard tree form the second field is a list of children, along with the element that resides on the arc connecting the top-level node to each of its children. A TStack can be pictured diagrammatically as follows:

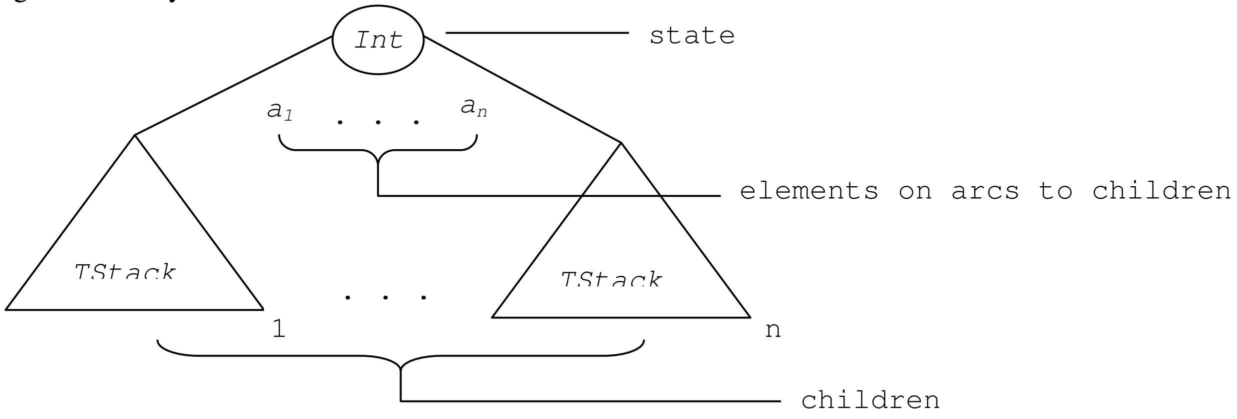


Figure 4.1: TStack ADT

[3] The *basic idea* for the `TStack` is borrowed from [Lju 02] although the implementation is somewhat different (see 3.2.2).

4.2.2 TStack Operations

- `initTS :: TStack a`

  Initialises a new `TStack` node with an empty list of children in state 0.

- `push :: a -> Int -> TStack a -> TStack a`

  Takes an element and a state along with an existing `TStack`, and returns a new `TStack` in the new state, whose single child is the old `TStack` paired with the input element. Illustrated as follows:

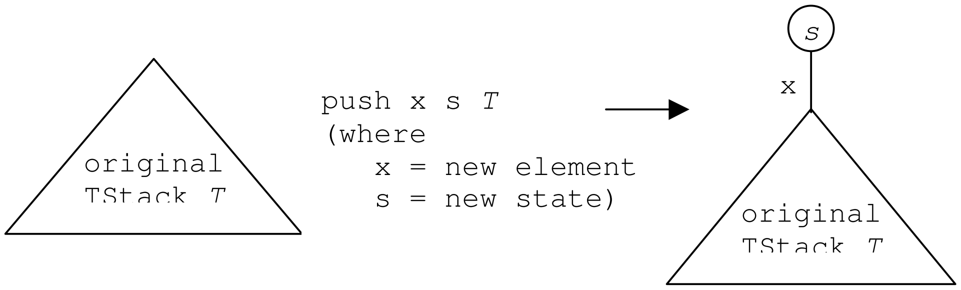


Figure 4.2: Push operation on TStack

- `pop :: Int -> TStack a -> [([a],TStack a)]`

  Recursively pops a given number of nodes from the given `TStack`, returning a list of the descendant `TStacks`, each one paired with a list of elements collected between it and the top node in the input `TStack`.

- `popF :: TStack a -> TStack a`

  Returns the *first* immediate descendant of the given `TStack`. Useful if we know that a `TStack` has only one child.

- `top :: TStack a -> Int`

  Returns the value in the *state* field of a `TStack`.

- `vals :: TStack a -> [a]`

  Returns a list of the elements on the arcs from the top node of a given `TStack`.

- `height :: TStack a -> Int`

  Returns an `Int` representing the number of arcs between the top node in a given `TStack` and its furthest descendant.

- `merge :: [TStack a] -> [TStack a]`

  Takes a list of `TStacks` and joins those with the same top states. To illustrate:

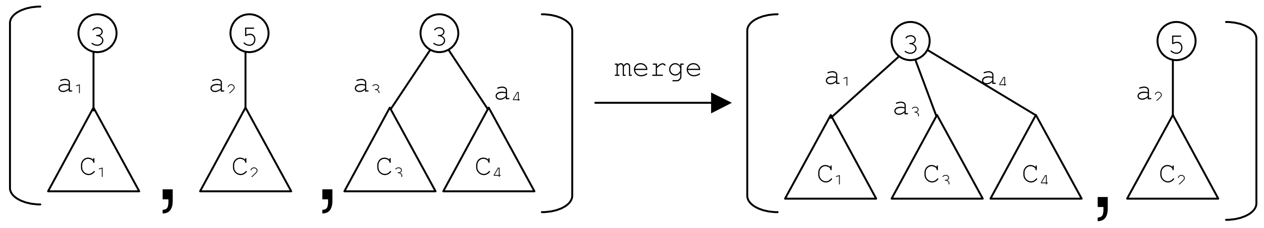


Figure 4.3: Merge operation on list of `TStacks`

A full listing of the TStack operations and their type signatures is provided in Appendix B.

Additionally, we overload the `compare` function for the `TStack` ADT, making it a member of the `Ord` class. This allows us to reverse order lists of `TStacks` by height, a necessity if packing is to occur whenever possible at the parser-driver level (3.3.2 – Figure 3.8).

4.3 High-Level Implementation

In this subsection we consider the implementation of the design model for the high-level structure of Tomita's algorithm proposed in section 3.3.

4.3.1 State Handling – Monadic Structure

It was noted in section 3.3.1 that some notion of state and sequence needs to be incorporated into the design and implementation model, allowing us to grow the parse forest throughout the various stages of the parse. In this subsection we will consider how the project implements state and sequence using the monadic framework in Haskell (see 2.5.2).

We need to carry the growing parse forest through the computation as an updateable state value. We therefore define a monad that allows us to accomplish this using a simple monadic type definition:

```
data ST s a = MkST (s -> (a,s))
```

In our model, the type variable `s` denotes the state value, and the type variable `a` denotes the GSS. We then define simple bind `(>>=)` and `return` operations from which we can construct monadic computations.

We also need to define operations that allow us to access and modify the state value (the parse forest in our case). To this end we implement two operations:

```
fromS :: (s -> b) -> ST s b
setS :: s -> ST s ()
```

The first is used to apply a function to the state value and return the result; the second replaces the current state value with the given one in subsequent computation.

Finally, we implement a function called `runST` which takes an initial state value along with a constructed monadic computation, and returns the final state (the completed parse forest):

```
runST :: s -> ST s a -> s
```

4.3.2 Parser Driver

We now consider the implementation of the parser driver in light of the design model specified in section 3.3.2.

The `doParse` function of figure 3.7 is implemented as `foldM` (part of the `Monad` library) applied to a function f, an initialised GSS and the list of input tokens, where f is a monadic implementation of the `doActions` function from figure 3.10, and the initialised GSS is simply a list containing a single initialised TStack object (see `initTS` operation, 4.2.2). The implementation is as follows:

```
doParse :: [Token] -> ST Forest [TStack FID]
doParse toks = foldM doActions [initTS] toks
```

where

```
type FID     = Int                          -- forest node indices
type Forest  = SetMap (ForestNode GSymbol)  -- the parse forest

data GSymbol = symbols in the grammar: terminals (Tokens) & non-terminals
```

This has the effect of constructing a monadic computation where `doActions` is applied to each token in the input list, modifying the GSS through the computation. To retrieve the completed parse forest we simply apply `runST` to an initialised parse forest and the result of `doParse`:

```
parse :: [Token] -> Forest
parse toks = runST initSM (doParse toks)
```

The implementation of `doActions` is as follows:

```
doActions :: [TStack FID] -> Token -> ST Forest [TStack FID]
doActions gss tok
 = do
      gss’ <- reduceAll tok gss
      shiftAll tok gss’
```

Using `do` notation makes clear what is happening. All relevant reduce operations are first performed on the GSS (list of `TStacks`), followed by all shift operations. Note that the state value (parse forest) is hidden by the monadic structure. We continue to flesh-out the implementation by providing definitions for the `shiftAll` and `reduceAll` operations:

```
shiftAll :: Token -> [TStack FID] -> ST Forest [TStack FID]
shiftAll tok [] = return []
shiftAll tok stks
 = do fid <- addNode (FNode tok [])
      return $ merge [ push fid st stk | (stk,st) <- stks ]
```

```
reduceAll :: Token -> [TStack FID] -> ST Forest [TStack FID]
reduceAll tok [] = return []
reduceAll tok (stk:stks)
 = case action (top stk) tok of
      Reduce rs   -> redAll rs
      Shift st rs -> do { ss <- redAll rs ; return $ (stk,st):ss }
      Accept      -> reduceAll tok stks
   where
    redAll rs = do let reds = concat [ reduce stk m n | (m,n) <- rs ]
                   stks' <- foldM pack stks reds
                   reduceAll tok stks'
```

These operations closely resemble their respective design models given in figure 3.10, raised to the monadic level. Note that both rely on encodings of the parse tables as functions. These are called `action` and `goto` and have type signatures:

```
action :: State -> GSymbol -> GLRAction
goto   :: State -> GSymbol -> State
```

where

```
       type State = Int
```

and the data type `GLRAction` is defined as follows (relate to Figure 3.9):

```
data GLRAction = Shift Int [Reduction]
               | Reduce [Reduction]
               | Accept
               | Error
     deriving Eq

type Reduction = (GSymbol,Int)
```

The function `reduceAll` makes use of an auxiliary function `reduce`:

reduction candidate — *reduction rule mother* — *number of daughters in reduction rule* — *children of new analysis (daughters)* — *ancestor* — *reduction rule mother*

```
reduce :: TStack FID -> GSymbol -> Int -> [([FID],TStack FID,GSymbol)]
reduce stk m n =
 [ (fids,stk',m) | (fids,stk') <- pop n stk ]
```

Note the use of `pop`, as described in 4.2.2.

To complete the definition of `reduceAll` we need to define `pack`. We considered the packing mechanism in section 3.2.3, and its function definition is as follows:

GSS — *children of new analysis (reduction rule daughters)* — *reduced TStack* — *new analysis node label (reduction rule mother)*

```
pack :: [TStack FID] -> ([FID],TStack FID,GSymbol) -> ST ... (Monad)
pack stks (fids,stk,m)
 = do let st = goto (top stk) m
      case fnd (\s -> top s == st && popF s == stk) stks of
       Nothing     -> do    fid <- addNode (FNode m [fids])
                            return $ insert (push fid st stk) stks
       Just (s,ss) -> do    let oid = head (vals s)
                            (Just (FNode _ ch),f) <- fromS (decElem oid)
                            setS f
                            fid <- addNode $ FNode m (fids:ch)
                            return $ insert (push fid st stk) ss
```

`pack` scans the GSS for a packing candidate (note that there can only be one, since the GSS already represents a fully-packed list of `TStacks`). If none is found, a new forest node is created containing the single analysis and its index is pushed onto the reduced `TStack`, along with the *goto* state. The `TStack` is then inserted (reverse height ordered) into the GSS. If a packing candidate is found, a new forest node is created containing the packed analysis and `decElem` is called to indicate that the old node has one less referee. The packed node index is then pushed onto the reduced `TStack` along with the *goto* state and inserted into the GSS.

Both `reduce` and `pack` use an auxiliary function called `addNode` – a simple function which adds a node to the parse forest and returns its index.

Section 3.3.2 ends with a note on accepting strings of *sets of tokens* as input rather just individual tokens. We incorporate this into our implementation model by accepting `[[Token]]` as the input type to `doParse`:

```
doParse :: [[Token]] -> ST Forest [TStack FID]
```

We then modify `doActions` so that it takes a list of tokens as input, duplicates the GSS for every token in the list, performs the relevant actions on each and then merges them back into a single GSS upon completion:

```
doActions :: [TStack FID] -> [Token] -> ST Forest [TStack FID]
doActions gss toks
 = do
      gsss <- sequence acts
      return (merge $ concat gsss)
 where
  acts = [ reduceAll tok gss >>= shiftAll tok | tok <- toks ]
```

4.3.3 Error Handling

In our discussion of the implementation thus far, we have somewhat over-simplified the picture, leaving out such issues as that of error-handling. We now move on to consider how error-handling can be incorporated at a monadic level.

We modify the type definition for the ST monad to include a form of *exception handling* using the `Maybe` data type:

```
data ST s e a = MkST (s -> e -> (Maybe(a,s),e))
```

Note that we include an extra type variable `e`, representing a state value in which we can 'collect' parse errors as they occur.

We expand the *bind* `(>>=)` operation so that if a `Nothing` value occurs at any stage in the computation we halt execution and simply return the value in `e`. In our design model, such a situation would represent a *total error* (see 3.3.3).

We alter the `runST` function so that it returns both the final state and the error state. In this way we can display *partial errors* along with the successful parse. These partial errors can be recorded by means of a newly defined function, `chgE`, which applies a given function to the error state:

```
chgE :: (e -> e) -> ST s e ()
```

We need some way of triggering a total error, and to this end we define a function called `throwE`, which enables us to inject a `Nothing` value into the computation and thus halt execution, returning only the error state.

We extend the `reduceAll` function to handle errors (blank parse table entries) when they arise:

```
...
 = case action (top stk) tok of
     Reduce rs   -> redAll rs
     Shift st rs -> do { ss <- redAll rs ; return $ (stk,st):ss }
     Accept      -> reduceAll tok stks
     Error       -> do { chgE (\es -> tok:es) ; reduceAll tok stks }
...
```

Note that in this implementation we only record the particular token that caused the error. We could however supply more useful error information by analysing the stack in which the error was found.

If at any stage of the parse there are no `TStacks` left in the GSS, then a total error has occurred and `throwE` is called to terminate the parse. This is accomplished by extending the `doActions` function to test for the empty GSS:

```
doActions [] _ = throwE
```

We define a new type to represent parse success or failure:

```
data GLRResult = ParseError [GSymbol]
               | ParseOK [GSymbol] Forest
```

Finally, we extend the top-level function to pattern match on the `Maybe` value returned by running the monad and identify a successful parse from a failure:

```
parse :: [Token] -> GLRResult
parse toks
 = case runST initSM [] (doParse toks) of
     (Nothing,e) -> ParseError e -- total error
     (Just(f),e) -> ParseOK e f  -- success + partial errors
```

4.4 Optimisation

The basic implementation model described in this chapter is the result of refining and simplifying the code over many iterations. However, a number of optimisations were carried out on the basic model presented in this chapter in order to increase its efficiency. In this section we consider just one of them.

Section 5.4.3.1 highlights a specific instance in which *profiling* was used to identify an area requiring optimisation. A significant proportion of the expense of packing in the basic implementation is related to equality testing on `TStack` objects, a prerequisite for packing candidacy. 5.4.3.1 describes a technique that dramatically improves its efficiency, involving labelling each `TStack` node with a unique ID value. In terms of the implementation this requires an extra field in the `TStack` type definition:

```
data TStack a = TS Int                    -- state
                   Int                    -- node ID
                   [(a,TStack a)]         -- [(element on arc , child)]
```

When we push a new node onto a `TStack` object, we now need to assign it an ID value. Thus, the type of `push` is amended to:

```
new element     new state     ID value
        ↘           ↓         ↙
   push :: a -> Int -> Int -> TStack a -> TStack a
```

We overload the `(==)` operator for `TStacks` so that it simply compares the top node ID:

```
instance Eq (TStack a) where
 (TS _ id _) == (TS _ id' _) = id == id'
```

We must now assign unique ID's each time we perform `push` on a `TStack` in the parser driver. To accomplish this we extend the monadic structure of the implementation to include a further state element:

```
data ST s r e a = MkST (s -> r -> e -> (Maybe(a,s,r),e))
```

Its value will be a list of `Ints` (initialised to `[0..]`). Every time we perform `push`, we will use a function called `readR` that takes an `Int` from the head of the list and returns it:

```
readR :: ST s [Int] e Int
readR = MkST $ \s (i:is) e -> (Just(i,s,is),e)
```

For example, `shiftAll` now appears as follows:

```
shiftAll tok [] = return []
shiftAll tok stks
 = do fid   <- addNode (FNode tok [])
      stks' <- sequence (shifts fid)
      return $ merge stks'
 where
  shifts fid = [ do { nid <- readR ; return (push fid st nid stk) }
               | (stk,st) <- stks ]
```

Section 5.4.3.1 explains why this optimisation is both correct and efficient.

4.5 Modifications to *Happy*

In this final subsection we consider the modifications made to *Happy* in order to integrate it with our implementation of Tomita's algorithm.

4.5.1 High Level Modification

Happy is composed of a number of Haskell modules. The high-level structure of the program, as illustrated in Figure 3.12, is contained in a module called `Main`. Figure 3.13 shows diagrammatically where the modification is made in the high-level structure of *Happy* to integrate it with the implementation of Tomita's algorithm. In practice this means altering the code in the module `Main` to include an extra flag (`-l`) and a conditional statement that triggers the generation of a GLR parser if the `-l` flag has been set by the user, and otherwise continues to produce a standard LR parser in the usual way. As shown in 3.4.2, we allow *Happy* to generate the LALR parse tables from the input grammar in both cases, and also to produce additional information about the tables if requested before introducing the following conditional statement:

```
...
if OptGLR `elem` cli
then produceGLRParser outfilename   -- specified output file name
                      template_dir  -- template files directory
                      action        -- action table (:: ActionTable)
                      goto          -- goto table (:: GotoTable)
                      header        -- header from grammar spec
                      tl            -- trailer from grammar spec
                      g             -- grammar object
else
...
```

Note that `OptGLR` is the internal name for the `-l` flag. We pass all the necessary data to the function `produceGLRParser` which writes the GLR parser using the given information about the grammar and the template containing the functional implementation of Tomita's algorithm.

4.5.2 GLR Code Generation

We now consider the code added to *Happy* which combines grammar-specific data with the implementation of Tomita's algorithm to produce a GLR parser. This code exists in the form of a new module added to the source code of *Happy* called `ProduceGLRCode`. It exports a single method, `produceGLRParser` which is called as part of *Happy's* modified high-level structure (see above).

The process of generating a GLR parser is one of assembling the various sections of the new parser file in `String` format and then outputting them to the designated file. This process is illustrated at a high-level by the following diagram:

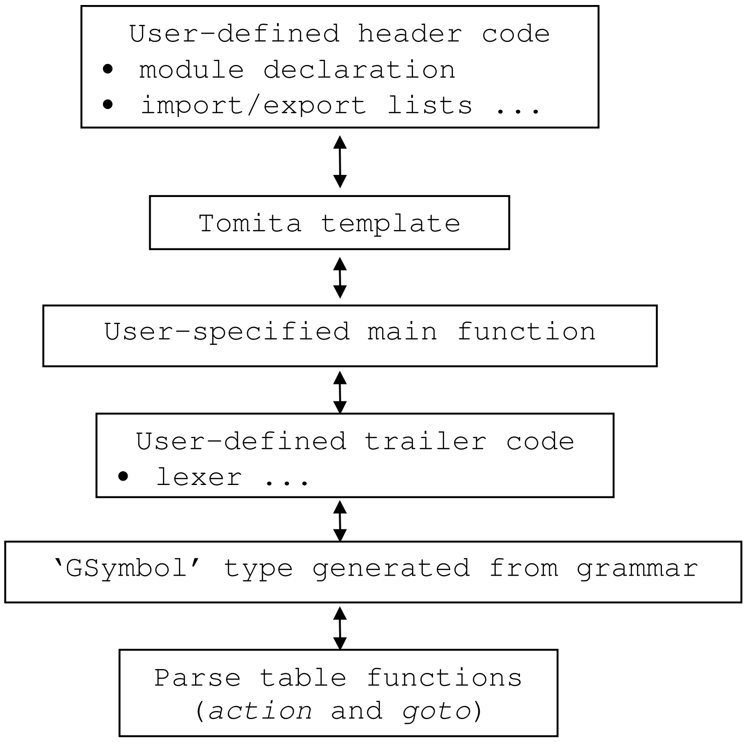

User-defined header code
• module declaration
• import/export lists ...
Tomita template
User-specified main function
User-defined trailer code
• lexer ...
'GSymbol' type generated from grammar
Parse table functions
(action and goto)

Figure 4.4: Breakdown of GLR code generation

The module contains a function called `mkFile` which gathers the textual data for each of the blocks shown above, calling auxiliary functions where necessary, concatenates them and writes them to the specified output file. Some of the phases, such as copying the code from the template or adding user code verbatim, are a simple case of grabbing text from a file or variable. Other phases are more complex; for example to generate the parse table functions we must coerce the data from the relevant tables into a format that can be written into the parser as a function definition.

Another phase requiring a certain amount of work is that of generating the `GSymbol` data type from the input grammar. Values of this type will be used to label the nodes of resulting parse forests. The `GSymbol` DT includes representations for the non-terminals of the grammar, as well as the terminals (tokens – the type of which is specified by the user) and the internal EOF symbol. The grammar-specific definition for the `GSymbol` DT is as follows:

data GSymbol = NT_1 | ... | NT_n | HappyTok *TokenType* | HappyEOF

where: *n* = *the number of non-terminals in the grammar*
TokenType = *the user-defined type for tokens (terminals)*

Much of the detail regarding the manipulation of internal *Happy* data structures into GLR parser code is excluded from this section, but a complete listing of the `ProduceGLRCode` module can be found in Appendix C.

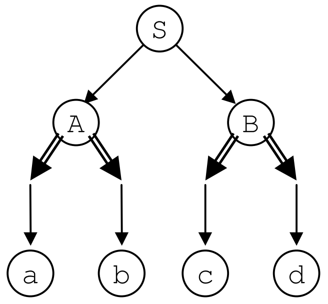
4.6 Forest Decoding

The project implements a technique for decoding a parse forest into a list of individual syntax trees (1.4.1). This is useful for analysis, as it is somewhat difficult to decipher even a moderately complex parse forest by hand. We use a standard data structure for representing trees:

```
data Tree a = TreeNode a [Tree a]
```

A forest node can be thought of as an encoding of a number of trees. Bear in mind that a forest node consists of `n` analyses, where `n` ≥ 0. Thus, to decode a forest node into its constituent trees we recursively decode its children in all analyses, and for each analysis we generate a list of all possible trees derivable from ordered combinations of the lists of trees for each child, returned by the recursive call. The recursion is founded in the instance where the forest node has no children (a leaf), in which case we return a list containing a single corresponding tree leaf. To decode an entire forest, we apply the decoding operation to its top-level node.

A simple example will help to clarify the decoding process. Consider the following hypothetical parse forest:

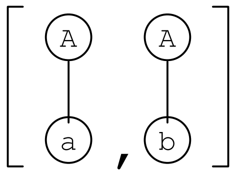


The node labels denote hypothetical grammar categories:

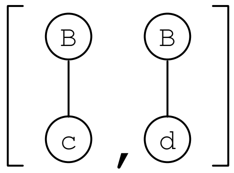
- upper case for non-terminals
- lower case for terminals

Note that `A` and `B` are packed nodes, each containing two analyses.

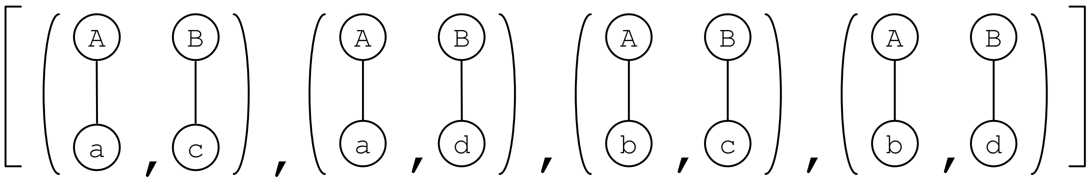
We decode the forest by applying the operation described above to the top-level node `S`. This calls for the recursive decoding of nodes `A` and `B`, and in turn nodes `a`, `b`, `c` and `d`. Once the first leaf node `a` is reached, the recursion bottoms out and the singleton list `[TreeNode a []]` is returned. The same is true for each of the forest leaf nodes. The node `A` consists of two analyses, each with a single child. From the first analysis, we have only a single derivable tree, as is the case with the second analysis, and so decoding `A` returns the following list of trees:



Similarly, decoding `B` returns:



We now have all the results necessary to decode the top-level node `S`. It consists of a single analysis with children `A` and `B`, and thus we generate all possible ordered combinations of the trees returned by their respective decoding calls:



This operation is similar to generating the Cartesian product of two sets, however we have a possible `n` sets, where `n` ≥ 1. The final list of trees returned is:

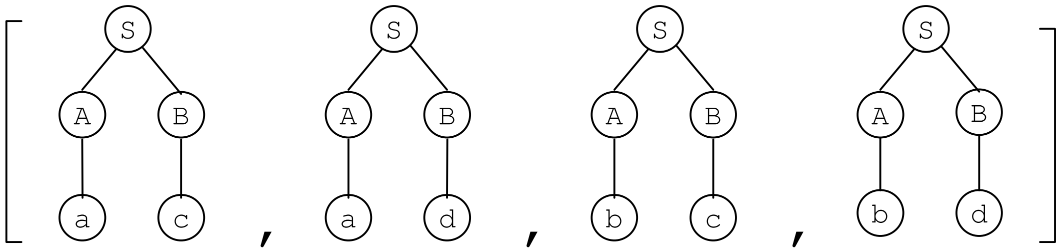


Figure 4.5: Forest decoding process

The Haskell code for the forest decoding process is listed in Appendix D.

Ch 5. Results and Evaluation

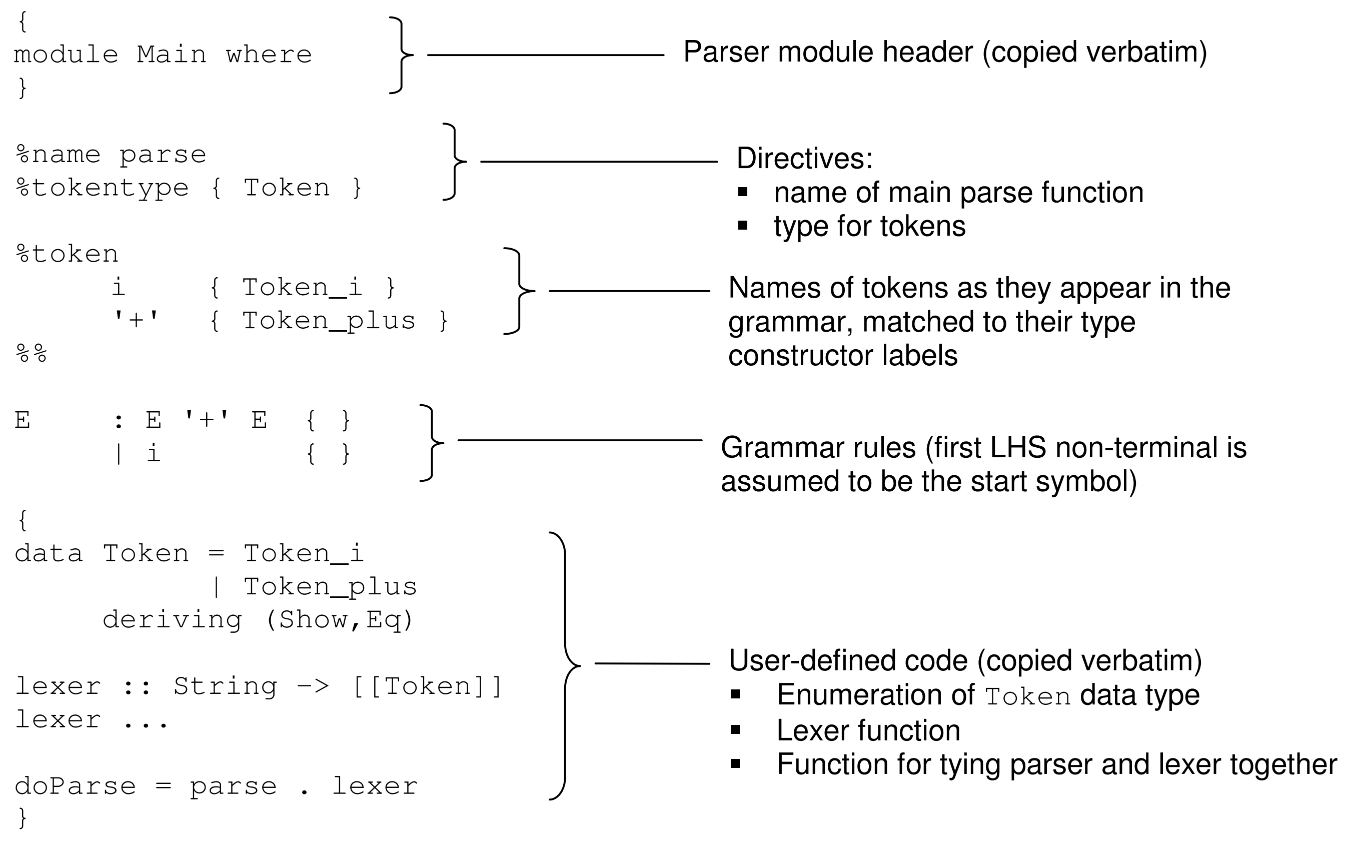
In this chapter we consider the results achieved during the project, focusing on output from our implementation of Tomita's algorithm integrated with *Happy* (we will henceforth refer to the modified version of *Happy* as *HappyGLR*). We then evaluate the implementation for correctness and efficiency, using theoretical and profile-based methodology. Finally, comparisons are made with similar contemporary implementations of Tomita's algorithm.

5.1 Generating a Parser Using *HappyGLR*

The process of generating a GLR parser using *HappyGLR* is essentially the same as for a standard LR parser (see 3.4.1). An input file is supplied, specifying the grammar, along with various related information. Let us consider the simple ambiguous grammar seen earlier:

$$E \rightarrow E + E \mid i$$

An input specification for this grammar would appear similar to the following:



```
{
module Main where
}

%name parse
%tokentype { Token }

%token
      i     { Token_i }
      '+'   { Token_plus }
%%

E     : E '+' E  { }
      | i        { }

{
data Token = Token_i
           | Token_plus
     deriving (Show,Eq)

lexer :: String -> [[Token]]
lexer ...

doParse = parse . lexer
}
```

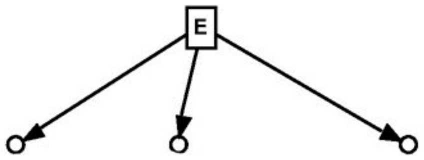
Parser module header (copied verbatim)
Directives:
▪ name of main parse function
▪ type for tokens
Names of tokens as they appear in the grammar, matched to their type constructor labels
Grammar rules (first LHS non-terminal is assumed to be the start symbol)
User-defined code (copied verbatim)
▪ Enumeration of Token data type
▪ Lexer function
▪ Function for tying parser and lexer together

Figure 5.1: Input file for E+E grammar

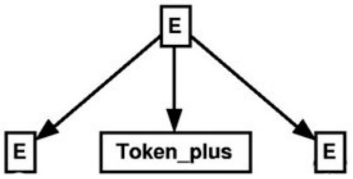
5.2 Output – daVinci

To make it easier to analyse parse results the project implements techniques for converting parse forests and lists of syntax trees (decoded from forests – see 4.6) into formats recognised by the graph-visualisation tool *daVinci*[4]. The results presented in this chapter will consist largely of graphs and trees produced by *daVinci*.

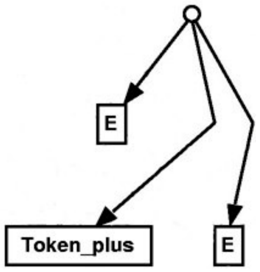
Packed nodes are represented in *daVinci* by arrows pointing to small circles. For example:



represents a node with three packed analyses.



and



represent single analyses.

It should be noted that the ordering of parse forest nodes is not necessarily preserved under *daVinci.* The Haskell code that deals with the translation of parse forests and lists of syntax trees into *daVinci* format is listed in Appendix E.

5.3 Results

In this subsection we present a sample of the results obtained from generating GLR parsers for several ambiguous context-free grammars using *HappyGLR*.

5.3.1 E+E Grammar

Recall the following simple ambiguous grammar:

```
    E → E + E | i
```

[4] See http://www.informatik.uni-bremen.de/daVinci/

Its *HappyGLR* input specification is given in Figure 5.1. Running the generated parser (through the *daVinci* translation mechanism) on the input string `'i+i+i'` produces the following parse forest:

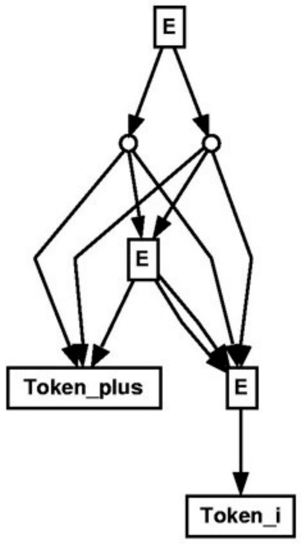


Figure 5.2: Parse forest: `'i+i+i'` (*daVinci*)

The forest decodes into the following collection of trees:

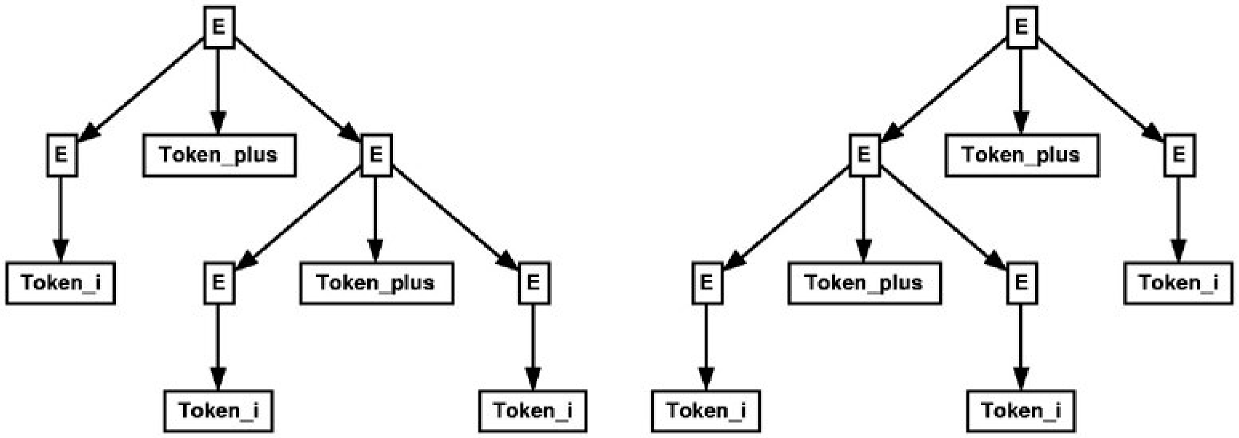


Figure 5.3: Syntax trees: `'i+i+i'` (*daVinci*)

This is the output we would expect, representing the ambiguity in associating the + operator. If we extend our input sentence to `'i+i+i+i'` the result is the following parse forest:

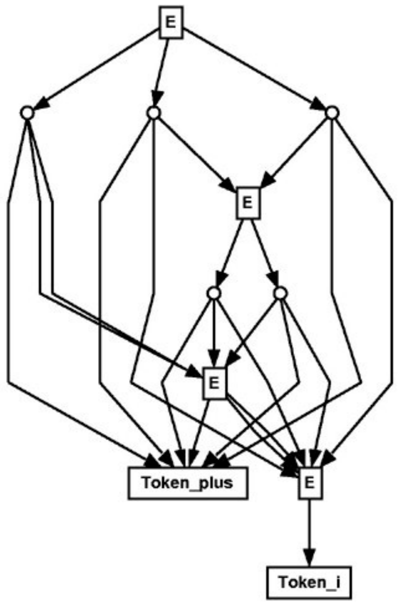


Figure 5.4: Parse forest: `'i+i+i'` (*daVinci*)

Note that we now have two levels of packed nodes. The forest decodes into the following collection of trees:

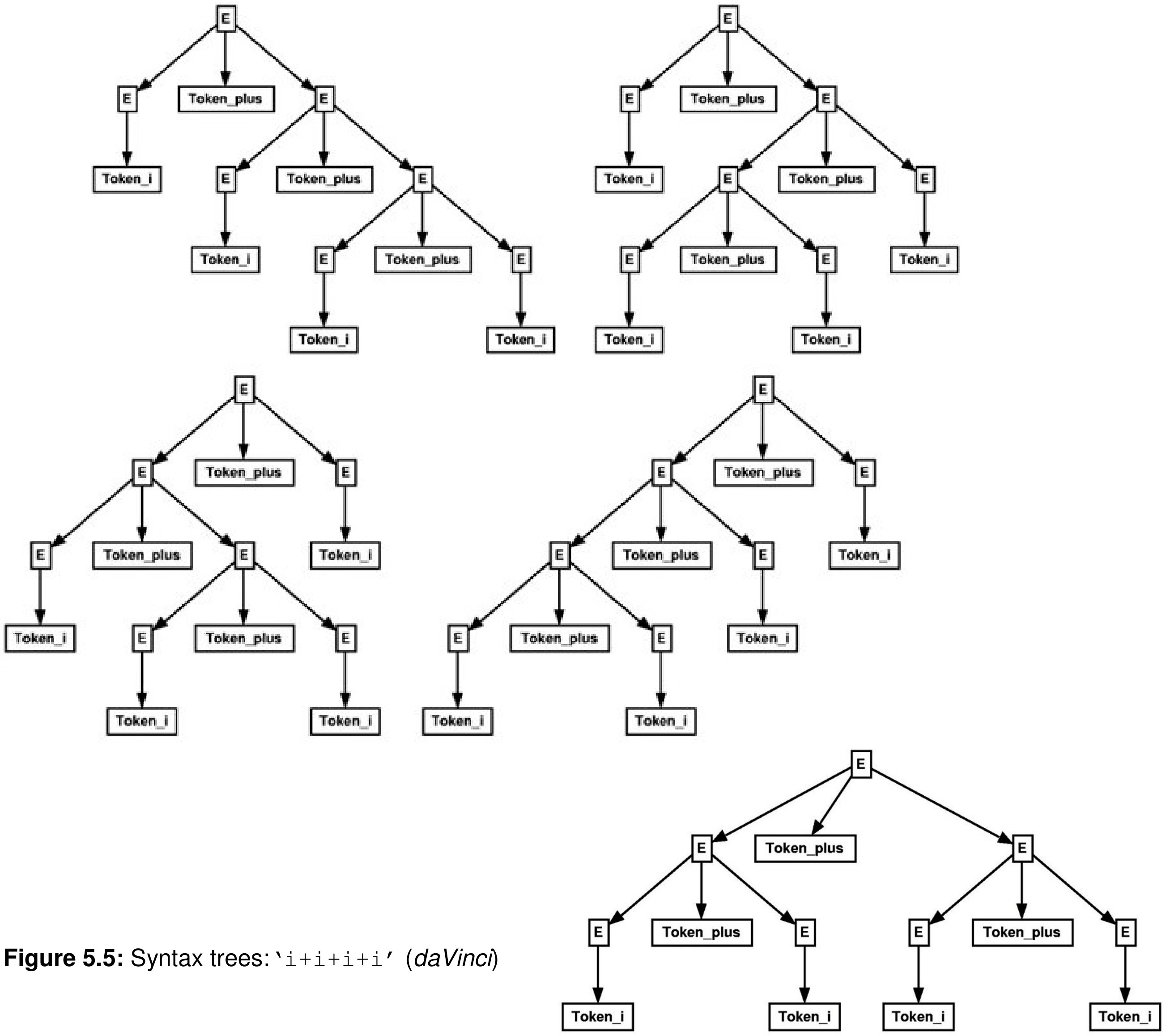


Figure 5.5: Syntax trees: `'i+i+i+i'` (*daVinci*)

The following figure illustrates how complexity increases rapidly as a function of the number of + operators in the input string under our simple ambiguous grammar (see 5.4.1). The forest is the result of parsing the sentence `'i+i+i+i+i+i+i+i+i+i'` (9 +'s) and encodes 4862 distinct syntax trees:

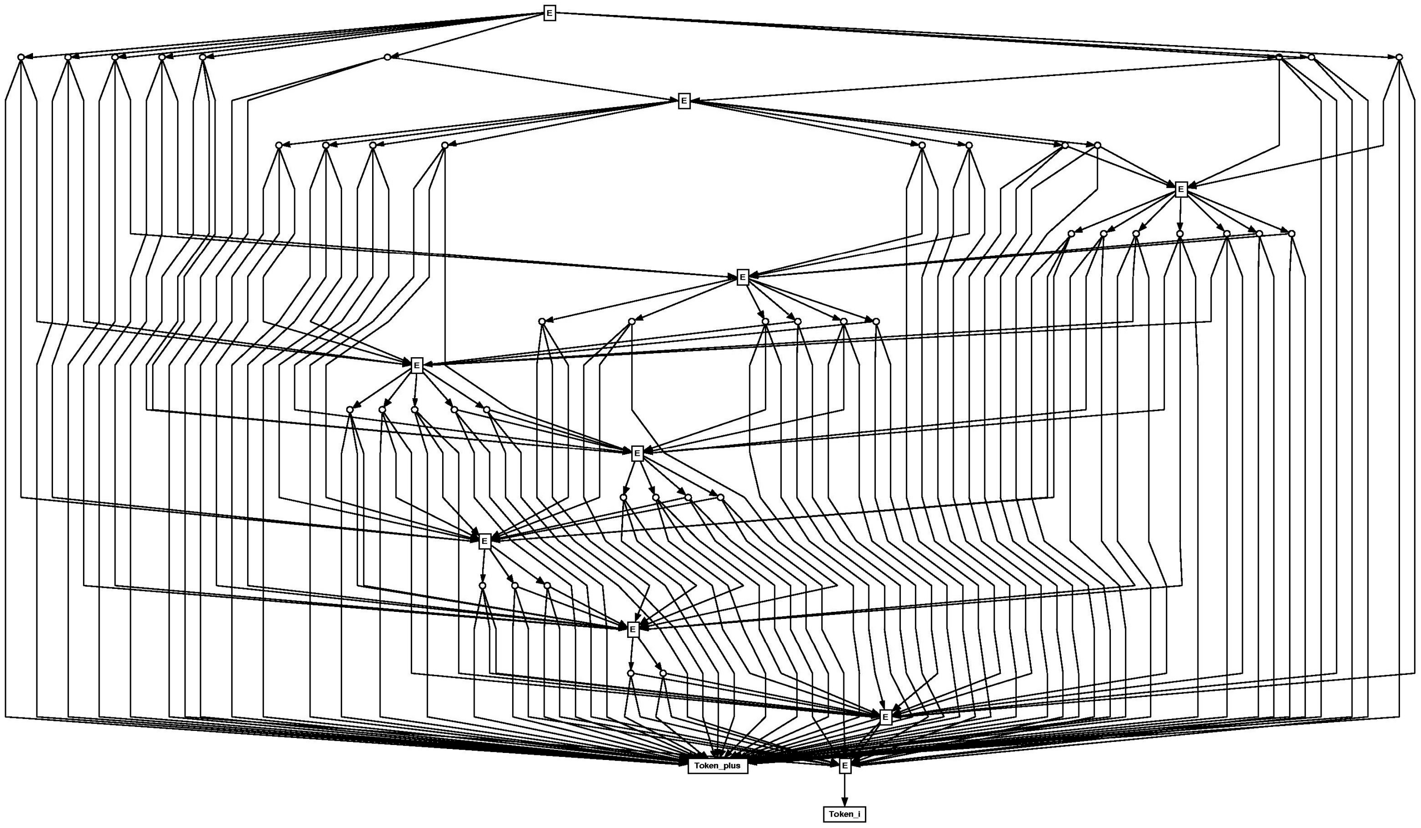


Figure 5.6: Parse forest: 9+'s (*daVinci*)

5.3.2 Simple NL Grammar

We now consider a simple natural-language grammar for English. Due to its simplicity it is of limited practical use, but will serve as a basic illustration. The grammar consists of about thirty production rules, and we will refer to it as *SimpleNL*. It is listed in Appendix F.

Parsing the sentence *"I saw a man in the park with a telescope"* under the SimpleNL grammar generates the following parse forest:

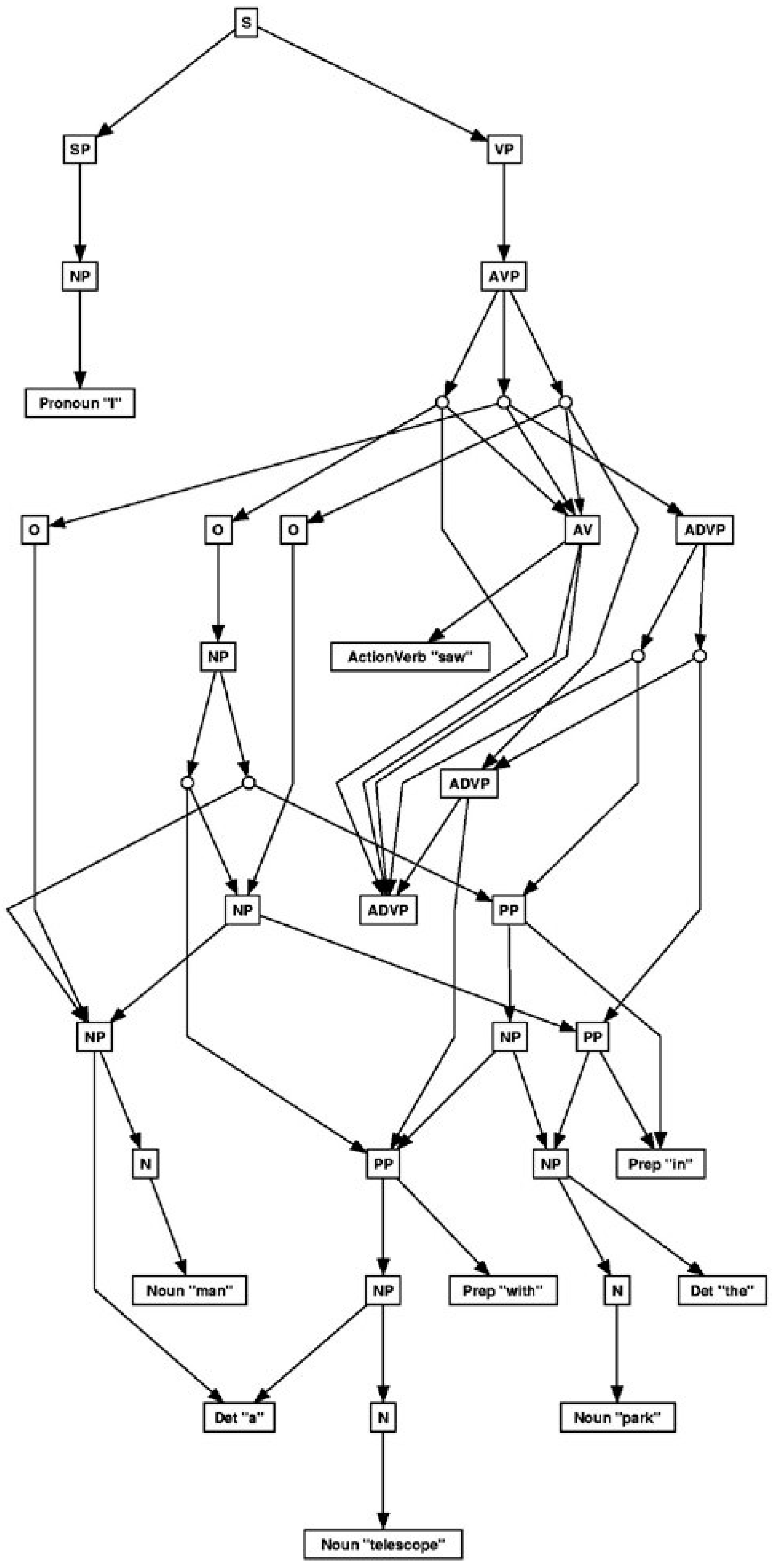


Figure 5.7: Parse forest: *"I saw a man in the park with a telescope"* (*daVinci*)

Note that non-terminal nodes without children correspond to empty production rules in the grammar. The forest decodes into the following collection of syntax trees:

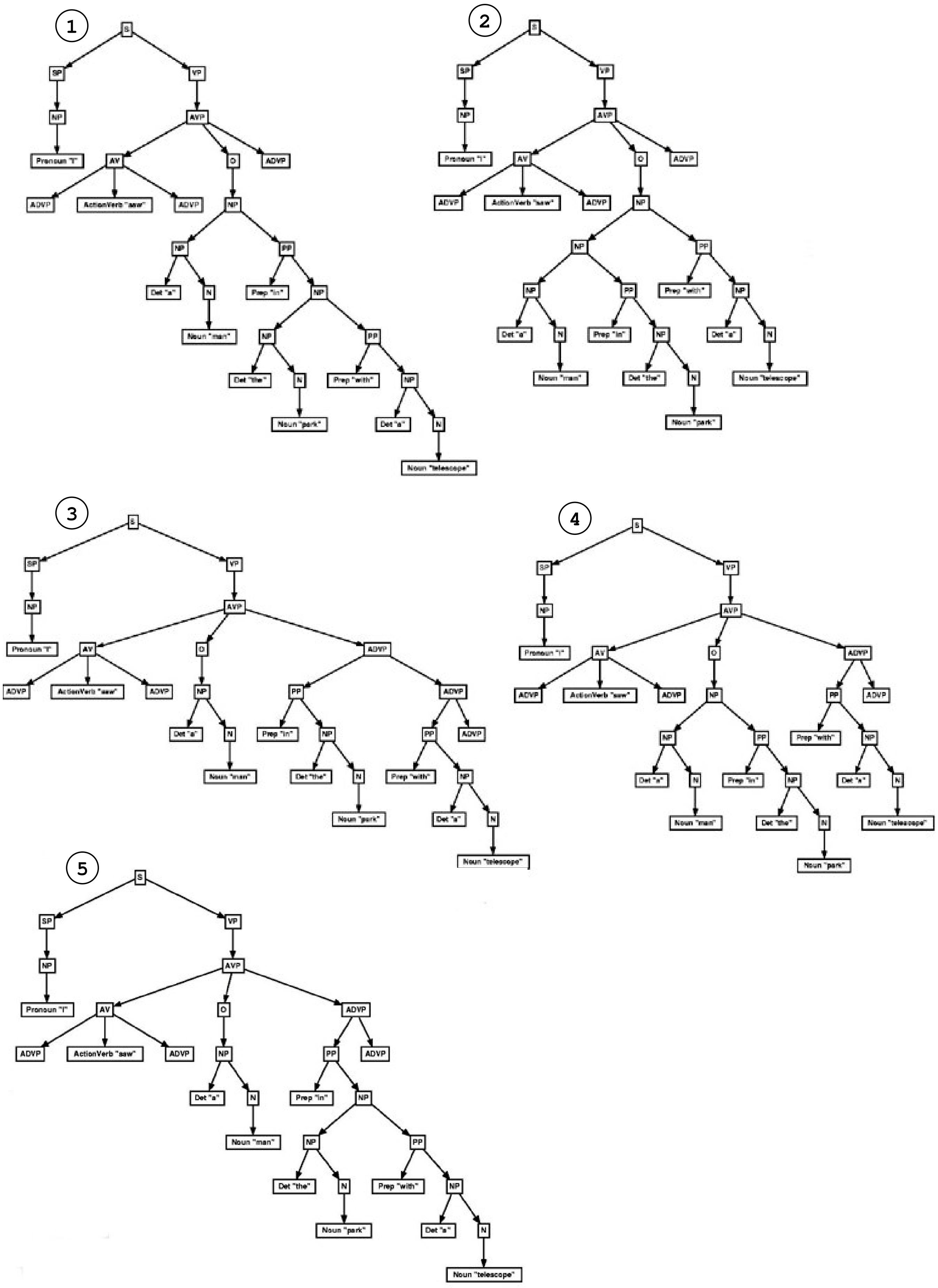


Figure 5.8: Syntax trees: *"I saw a man in the park with a telescope"* (*daVinci*)

The five syntax trees correspond to different syntactic interpretations of the input due to ambiguity in the interpretation of the two prepositional phrases. The following diagram illustrates the structure of each of the trees, highlighting the part of speech associated with each of the prepositional phrases in each case:

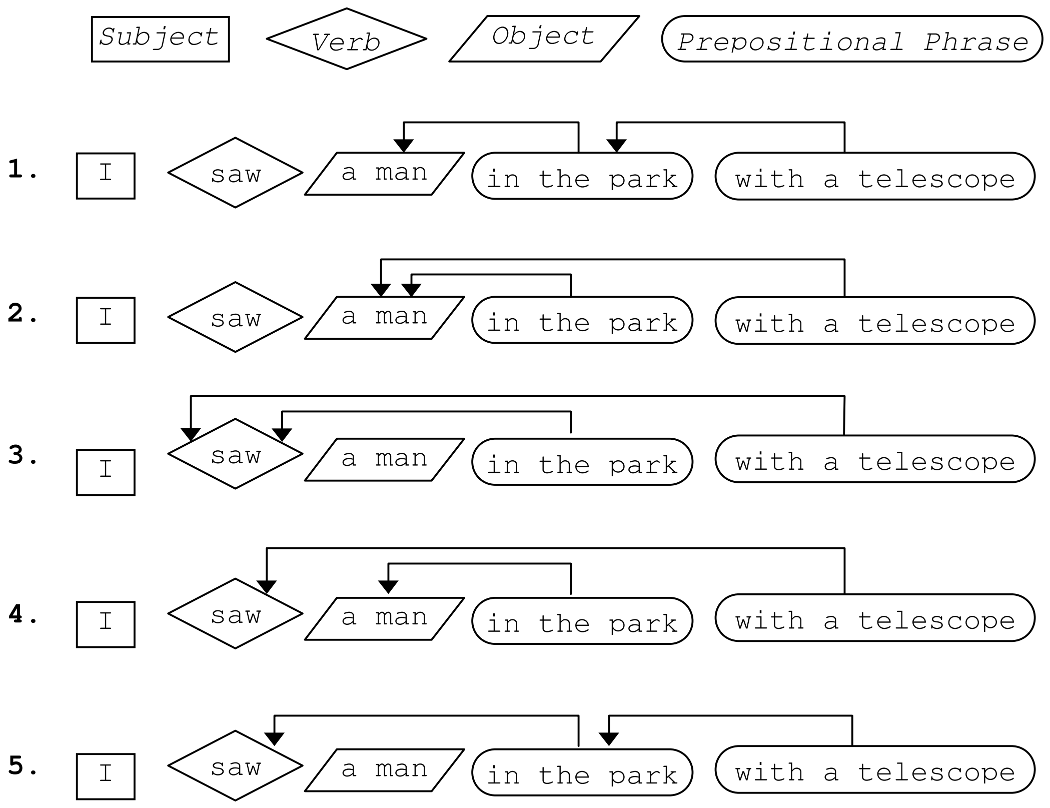
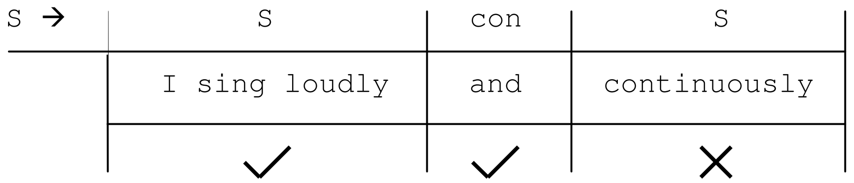


Figure 5.9: Syntactic interpretations of *"I saw a man in the park with a telescope".*

Let us consider another input sentence: *"I sing loudly and continuously"*. This example is chosen to highlight the error-handling mechanism of the parser. Parsing this sentence under the SimpleNL grammar generates the following raw output:

```
ParseOK [HappyTok (Adverb "continuously")] ...
```

This tells us that the parse has been successful, but that a partial error (see 3.3.3) has occurred while processing the token adverb 'continuously'. Recall that the implementation of the error-handling mechanism only identifies the token at which the error occurred, although it could be extended to return more useful information (see 4.3.3). The error occurs because the parser attempts to derive the sentence under the production rule '`S -> S con S`'. The following diagram illustrates this:



The parse forest generated from the above input sentence is as follows:

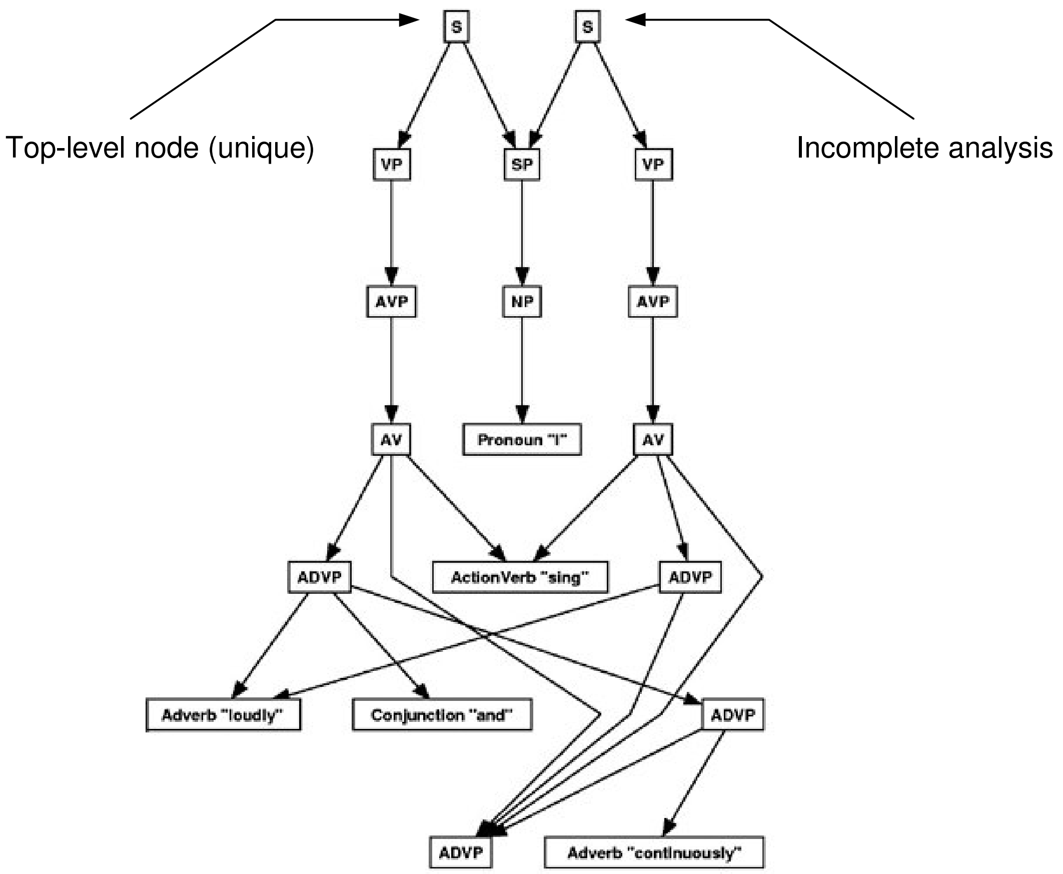


Figure 5.10: Parse forest: *"I sing loudly and continuously"* (*daVinci*)

The forest appears to have two top-level nodes, but in fact one represents an incomplete analysis arising from the partial error at 'continuously'. Complete parse forests (such as the one above) always posses a *unique* top-level node from which incomplete analyses can be filtered out. Thus, decoding the forest results in the single syntax tree:

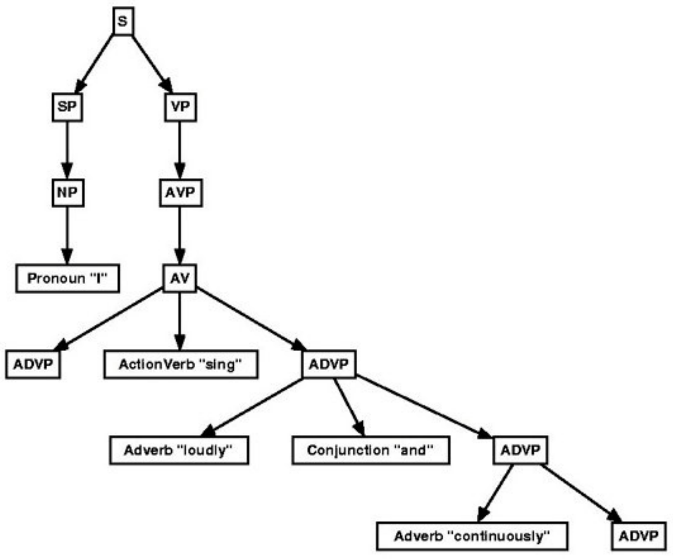
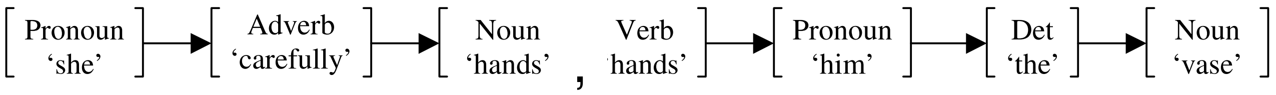


Figure 5.11: Syntax tree: *"I sing loudly and continuously"* (*daVinci*)

The last part of section 4.3.2 describes how the input can be specified in lists of tokens instead of just single tokens. This can be demonstrated by considering the result of parsing the sentence *"she carefully hands him the vase"*, where the lexeme *hands* can function both as an action verb and a plural noun. The input, then, is structured as follows:



The resulting raw output begins:

```
ParseOK [HappyTok (Noun "hands")] ...
```

From this, we can see that the analysis in which *hands* acts as a noun causes a partial error. The resulting forest decodes into the single following syntax tree, representing the successfully parsed analysis in which *hands* functions as an action verb:

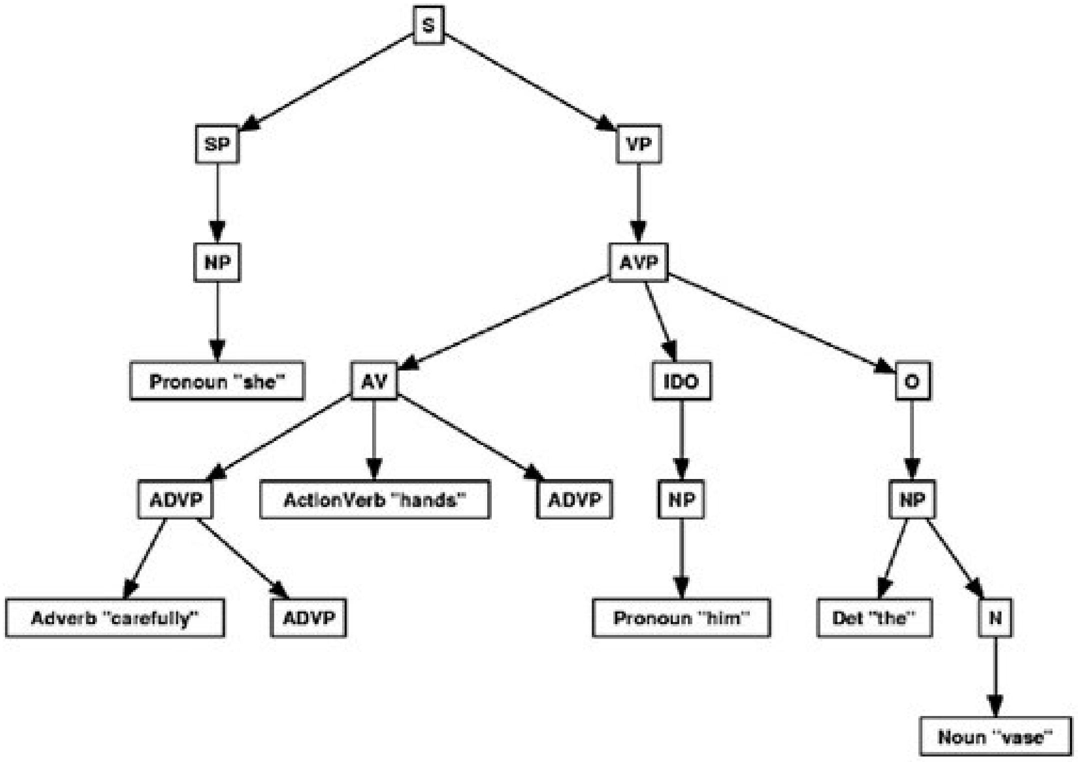


Figure 5.12: Syntax tree: *"she carefully hands him the vase"* (*daVinci*)

5.4 Evaluation

In this section we evaluate the functional implementation of Tomita's algorithm presented in this report for correctness and efficiency, providing examples of how *profiling* was used to optimise the space and time efficiency of the implementation. We conclude by comparing the implementation with its contemporaries.

5.4.1 Correctness

A formal proof of the correctness of Tomita's algorithm is beyond the scope of the project, and we appeal to the reader to consider the correctness of the design and implementation models proposed in this report in light of such references as [Kip 89] and [Car 93].

An informal method of judging the correctness of the implementation is to analyse the output generated by the parser under a specific grammar, and ascertain whether or not it is as we would expect, given some theoretical basis for our expectations. Consider once more the E+E grammar from previous sections:

```
E → E + E | i
```

By thought and manual enumeration we can calculate that for the input string '`i+i+i`' there are two possible syntactic interpretations. Similarly we know that if we extend the input string to '`i+i+i+i`' we would expect five distinct syntactic interpretations. These results are given in 5.3.1, and thus we know that our implementation of the algorithm is correct for these simple examples. We can extend this idea by calculating the number of distinct syntax trees we would expect for a given input string under this grammar, and then running the parser on the same input, decoding the resulting forest and counting the number of distinct syntax trees generated. By analysis of the properties of the grammar, it can be deduced that the following recursive function calculates the number of trees for a given number of '+' operators (n) in the input string:

$$\mathrm{Tr}(0) = 1$$

$$\mathrm{Tr}(n) = \sum_{i=0}^{n} \mathrm{Tr}(i-1) * \mathrm{Tr}(n-i)$$

The function formalises the following two facts:

- If there are no + operators in the input, there is no ambiguity and only one possible interpretation.
- The total number of interpretations derivable from an input sentence is equal to the sum of the number of interpretations derivable from each of the states in which a different + operator in the input gets to be the top node.

We can write the above function in Haskell:

```
tr 0 = 1
tr n = foldr (+) 0 [ tr (i-1) * tr (n-i) | i <- [1..n] ]
```

Using this function we can derive the following table:

# +'s	# syntax trees
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430
9	4862
10	16796

Table 5.13: Results of `'#+ -> #trees'` function

Running the parser on the corresponding input sentences, decoding the resulting forests and counting the number of trees generated allows us to construct the following table:

input	# distinct syntax trees
i+i	1
i+i+i	2
i+i+i+i	5
i+i+i+i+i	14
i+i+i+i+i+i	42
i+i+i+i+i+i+i	132
i+i+i+i+i+i+i+i	429
i+i+i+i+i+i+i+i+i	1430
i+i+i+i+i+i+i+i+i+i	4862
i+i+i+i+i+i+i+i+i+i+i	16796

Table 5.14: Results of counting generated distinct syntax trees

By comparing tables 5.13 and 5.14 it is clear that the output from the parser is as we would expect, at least as far as the number of trees is concerned.

The results in table 5.14 were produced by generating a list of the syntax trees for a given input, testing for uniqueness, and counting the number of elements in the list. Although the test does not guarantee that the set of trees produced by the parser is the complete set of syntactic interpretations for the given input, it does guarantee the uniqueness of each of the trees generated, and due to the convergence of the results achieved strongly suggests that the parser is capturing all distinct derivations.

Given time, we could construct similar tests for more complex grammars, and thus enhance our belief in the correctness of the implementation.

5.4.2 Efficiency

We now turn our attention to an evaluation of the efficiency of the implementation in terms of both computation time and memory usage.

5.4.2.1 Time Complexity

Tomita originally claimed that his algorithm was $O(n^3)$ in general, though possibly worse on densely ambiguous grammars[5]. Kipps analyses the algorithm [Kip 89] and presents evidence to suggest that its complexity is $O(n^{p+1})$ where p is the number of daughters in the longest production in the grammar. We will consider the complexity of the implementation proposed in this report in light of these claims. To make matters less complicated, we will constrain our analysis to the familiar E+E grammar:

```
E → E + E | i
```

Let us consider the case in which input strings are simply lists of individual tokens. If we have an input string of length `n` (inclusive of the EOF token), then the algorithm has time:

*O(n) * complexity(doActions)*

The complexity of `doActions` is:

complexity(reduceAll) + complexity(shiftAll)

The complexity of `reduceAll` is somewhat difficult to calculate. It recursively processes each `TStack` in the list that makes up the GSS, but reductions generate new `TStacks` that must also be processed. The number of `TStacks` in the GSS is bounded by a constant, the number of LR states in the grammar, 6 in this case (this is so because the top states are guaranteed to be distinct when passed to `reduceAll`). When an action other than a reduction is specified for a particular `TStack` by the parse table, a single operation is performed and the `TStack` is removed from consideration. When a reduction is performed on a `TStack`, it generates a number of new `TStacks`, equal to the number of ancestors at a distance of k from the top node (where k is the number of daughters in the reduction rule). The value of k is bounded by the number of + operators encountered thus far in the input minus the number of + operators already analysed at the top level. (the i'th ancestor corresponds to an analysis in which the i'th + operator below the top node has been reduced).

[5] See Carroll's thesis [Car 93], pp. 95-99

Each new `TStack` must be processed in the same way. The number of reductions that can be applied to a succession of `TStacks` is bounded by the number of + operators seen thus far in the input. Whenever a reduction is carried out, the new `TStacks` must be tested for qualification to be packed. Packing reduces the order of complexity by the fact that once we have performed a reduction and exposed `TStacks` representing an analysis for each + operator stage in the input, all subsequent reductions can be packed into one of these existing `TStacks`, giving a bound of $O(n^2)$ as opposed to $O(2^n)$ for the number of reductions required in the `reduceAll` operation. Formulating this gives us the following complexity for the `reduceAll` operation:

$$O(6 * \left[\frac{n_+(n_+ + 1)}{2}\right]) * (complexity(reduce) + complexity(pack))$$

$$\text{where } n_+ = \lfloor n/2 \rfloor$$

The complexity of `reduce` is the number of ancestors (bounded above) times the number of daughters in the longest production rule in the grammar (3 in this case), thus:

$$O(n_+ * 3) \quad = \quad O(n)$$

Packing requires us to search the `TStacks` in the GSS for qualification, and then to update the parse forest with the relevant node (packed or unpacked). If packing has occurred, we also remove the old unpacked node if it is no longer referenced by the GSS. Thus, `pack` has the following complexity:

$$\underbrace{O(n_+)}_{\substack{\text{Search for qualification,}\\ \text{bounded by number of +'s.}\\ \text{Assume that comparison}\\ \text{has constant time.}}} \quad + \quad \underbrace{O(f(n))}_{\substack{\text{Add a node to the}\\ \text{forest, bounded by}\\ \text{some function of the}\\ \text{input length.}}} \quad + \quad \underbrace{O(f(n))}_{\substack{\text{Delete a node,}\\ \text{bounded as for}\\ \text{adding.}}}$$

Because of subtree sharing (see 3.1.1) the number of nodes in the forest grows at a rate of *O(n)*. The add and delete operations are essentially comparisons mapped over the elements in the parse forest, where an individual comparison is bounded by a constant, as is the actual adding and deleting of elements. This leaves the `pack` function with an overall complexity of *O(n)*. It should be noted, however, that in practice the operation is quite expensive due to its multiple sub-components.

The overall complexity of `reduceAll`, then, is

$$O(6 * \left[\frac{n_+(n_+ + 1)}{2}\right]) * (O(n) + O(n)) \quad = \quad O(n^3)$$

The final operation to consider is `shiftAll`. It consists of two sub-operations:

- add a new node to the parse forest
- push the new elements onto all `TStacks` in the GSS and merge

The first has time *O(n)* (see above), and the second has constant time because the number of `TStacks` in the GSS is bounded by a constant, therefore `shiftAll` has time *O(n)*.

Thus the whole implementation has the following time complexity:

$$O(n) * O(n^3) + O(n) = O(n^4)$$

Let us test the analysis by comparing the graph of this function with that of the amount of work done by the parser generated from the grammar in question:

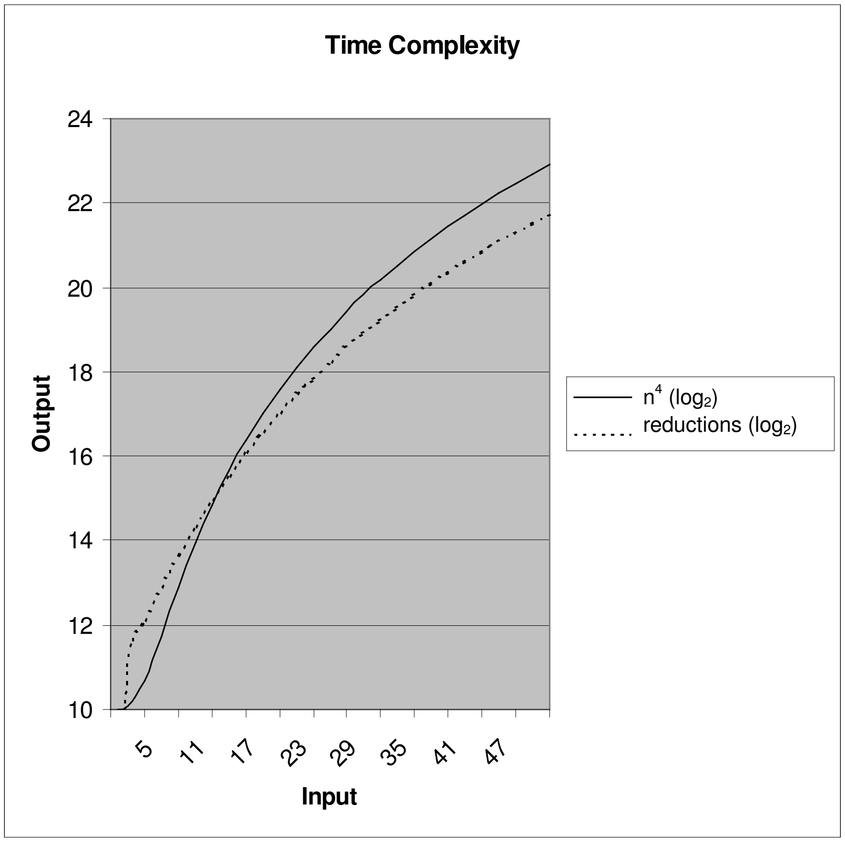


Figure 5.15: Implementation time complexity graph

The values for the implementation were generated by running the parser on inputs of varying lengths and recording the number of reductions[6] required for each input length. The results displayed by the graph support the time complexity analysis given above.

[6] A *reduction* is a single step of computation in the underlying lambda calculus.

The analysis is consistent with Kipps' evaluation of the algorithm, in which we would expect a time complexity of $O(n^4)$ for the grammar in question.

5.4.2.2 Space Complexity

Space complexity for Tomita's algorithm is based on two factors:

- The space required to store the parse forest as it is grown.
- The space required to maintain the GSS while the parse is in process.

Tomita suggests that his parse forest representation takes $O(n^3)$ space in general, though may require more for densely ambiguous grammars[7]. Kipps [Kip 89] argues that the space required to maintain the GSS is $O(n^2)$, although Johnson [Joh 89] shows that for some grammars the space required for the GSS may be exponentially related to the size of the grammar; though this is not the case for the grammars we have considered in this report.

A formal analysis of the space complexity of the implementation is beyond the scope of the project; however, in light of the analyses carried out by Kipps and Johnson, we would expect the space complexity of the implementation to be somewhere in the region of $O(n^2)$ to $O(n^3)$.

Experimentation yields the following results for the E+E grammar (Grammar 1.5):

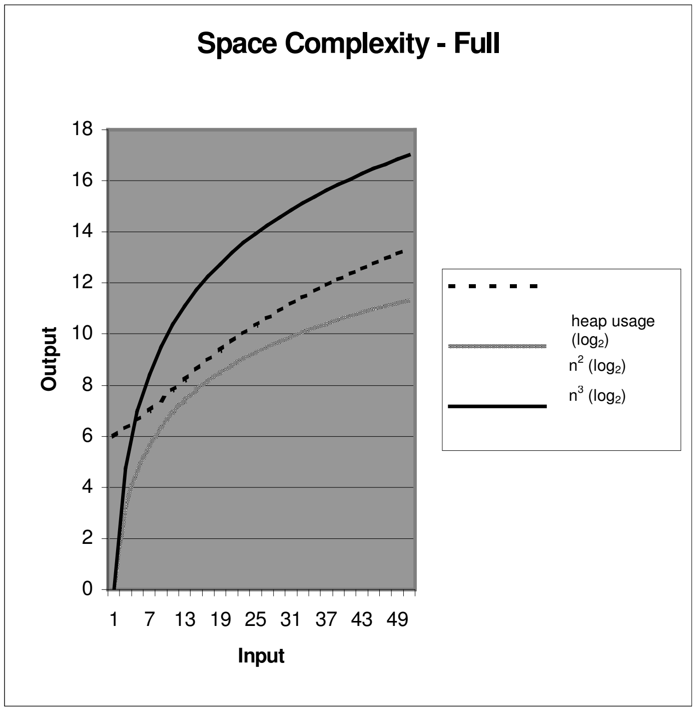


Figure 5.16: Space complexity – full implementation

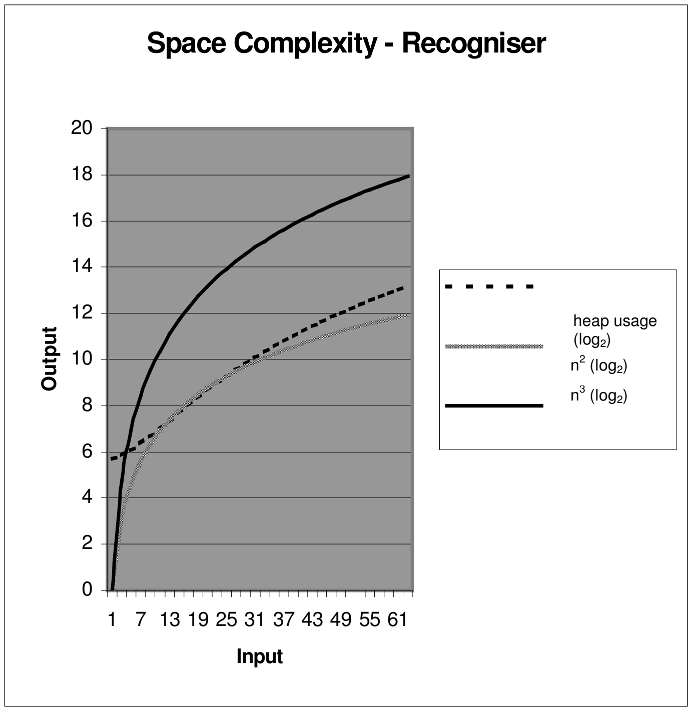


Figure 5.17: Space complexity – recogniser only

[7] See [Car 93], pp. 95-99

Figure 5.16 shows the heap usage of the implementation in relation to the functions n^2 and n^3. From these results it appears that the amount of memory used by the implementation is worse than n^2, although better than n^3. The results for Figure 5.17 were obtained by removing the sections of the code that construct the parse forest, essentially converting the implementation into a *recogniser*. This improves the space complexity, but still does not do enough to make it $O(n^2)$.

We present one final graph, plotting the amount of memory *allocated* during execution (this is not the same as the amount of heap space required) against the same two functions. Again we discover that the complexity is somewhere between $O(n^2)$ and $O(n^3)$:

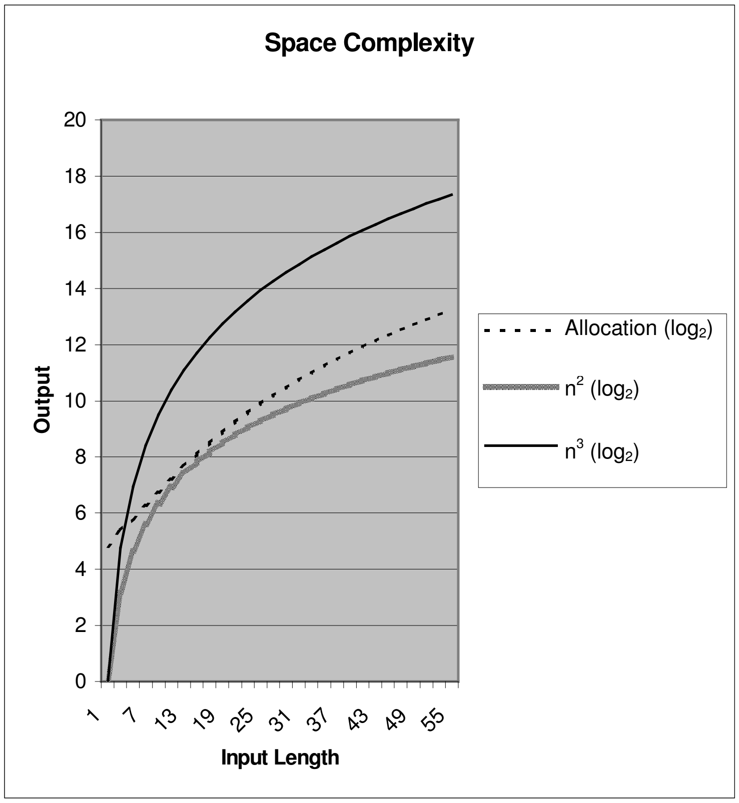


Figure 5.18: Space complexity – memory allocation

5.4.3 Profiling

Profiling was introduced in Chapter 2 as an effective method of iteratively evaluating and improving the performance of Haskell code. In this subsection we present samples of profiling methods used on the implementation to improve its efficiency.

5.4.3.1 Cost Centre Profiling

Profiling the implementation at a relatively early stage of optimisation generated the following cost centre statistics:

```
COST CENTRE          MODULE      %time %alloc

pack                 Main         89.7    9.1
GC                   GC            4.4    0.0
getElem              SetMap        2.9    0.4
merge                TStack        1.5    0.6
getInd               SetMap        1.5   12.9
addElem              SetMap        1.5    1.9
combine              Main          1.5   15.2
vals                 TStack        0.0    2.5
...                  ...           ...    ...
```

Figure 5.19: Cost centre profile before packing optimisation

The profile makes it clear that `pack` accounts for almost 90% of the computational expense. Effort was concentrated on improving the performance of `pack`, and it was discovered that a major cost of packing related to equality testing on `TStacks`. In order to determine whether two `TStacks` can be packed, we must test their immediate descendants for equality. In the original implementation, this took *O(n)* work because in the worst case we had to scale the depth of the entire tree to ensure equality (bounded by *n*, the number of symbols in the input), making the complexity of `pack` $O(n^2)$ and accounting for its high expense. A novel method of improving this complexity was identified: every new node pushed onto a `TStack` is labelled with a unique ID value, which can then be used to compare `TStacks` at the top level, without requiring any traversal of the tree (see 4.4 for implementation details). This is correct for our implementation because new `TStacks` are created by *duplication*. Consequently node ID's are also duplicated, and thus when two `TStacks` reach a point where their top level analyses can be packed, it must be the case that their children represent the same duplicated `TStack`. This improves the complexity of `pack` to *O(n)*, and thus the complexity of the entire algorithm from $O(n^m)$ to $O(n^{m-1})$ where *m* is dependent on the grammar. Other less dramatic optimisations were made to `pack` as a result of analysing cost centre information, and this was reflected in subsequent profiles, such as the following:

```
COST CENTRE          MODULE      %time %alloc

addElem              SetMap       20.0    2.6
pack                 Main         20.0   23.0
main                 Main         20.0    8.1
addNode              Main         20.0    3.8
GC                   GC           20.0    0.0
vals                 TStack        0.0    2.0
push                 TStack        0.0    3.0
...                  ...           ...    ...
```

Figure 5.20: Cost centre profile after packing optimisation

Cost centre profiling was used effectively over many iterations, resulting in code that runs approximately 60 times faster, allocating only around 1.6% as much memory as the original implementation.

5.4.3.2 Heap Profiling

As mentioned in Chapter 2, heap profiling can be used to illuminate the memory behaviour of a program and identify space leaks.

The following is a heap profile from an early stage of the implementation:

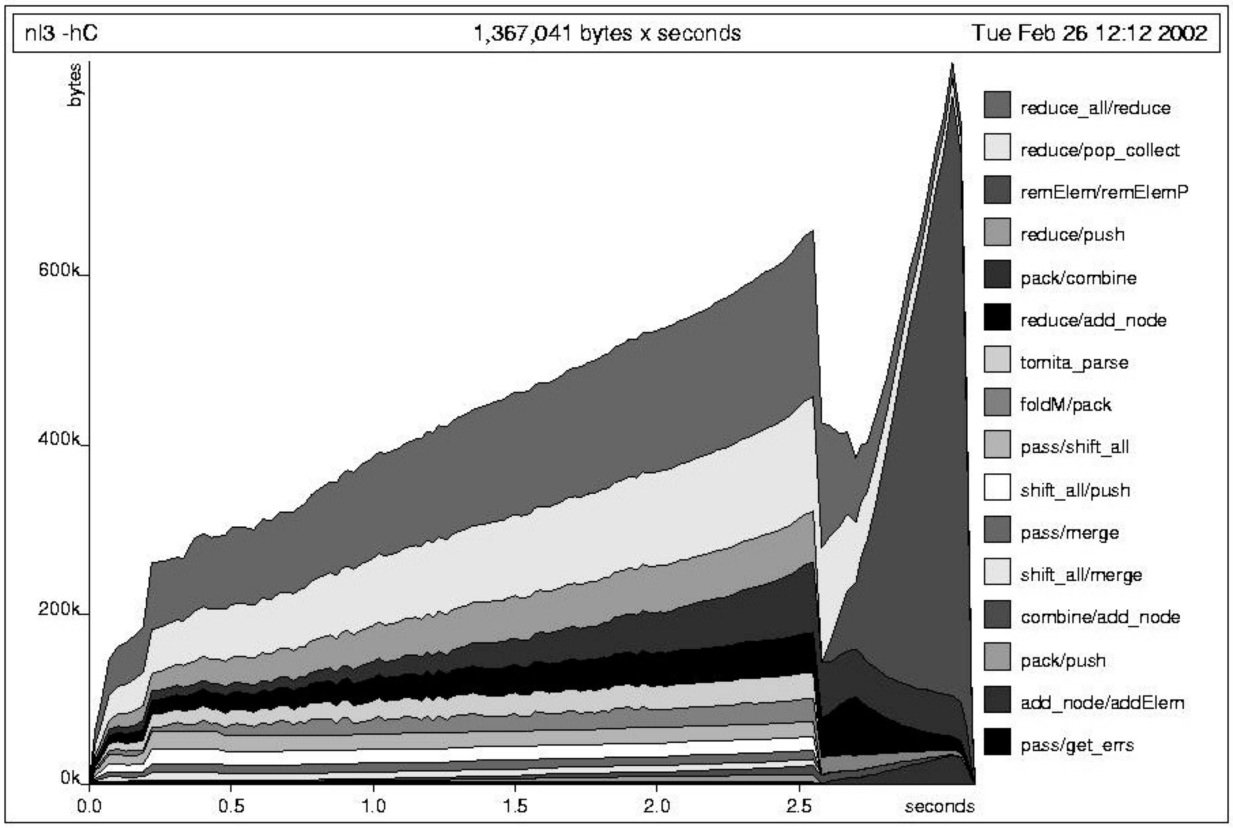


Figure 5.21: Early implementation heap profile (SimpleNL grammar)

This profile was generated from a parser for the SimpleNL grammar (5.3.2). The gradual increase in heap size is what we would expect as the GSS grows through the stages of the parse. However, the sudden jump after 2.5 seconds[8] suggests that memory is being used inefficiently during the construction and manipulation of the parse forest. This was addressed and a number of changes were made, especially to the operations on the `SetMap` data type.

[8] Note that running times reported during heap profiling are inaccurate due to profiling overheads.

Subsequent profiles show a much more even spread, along with significantly reduced heap usage:

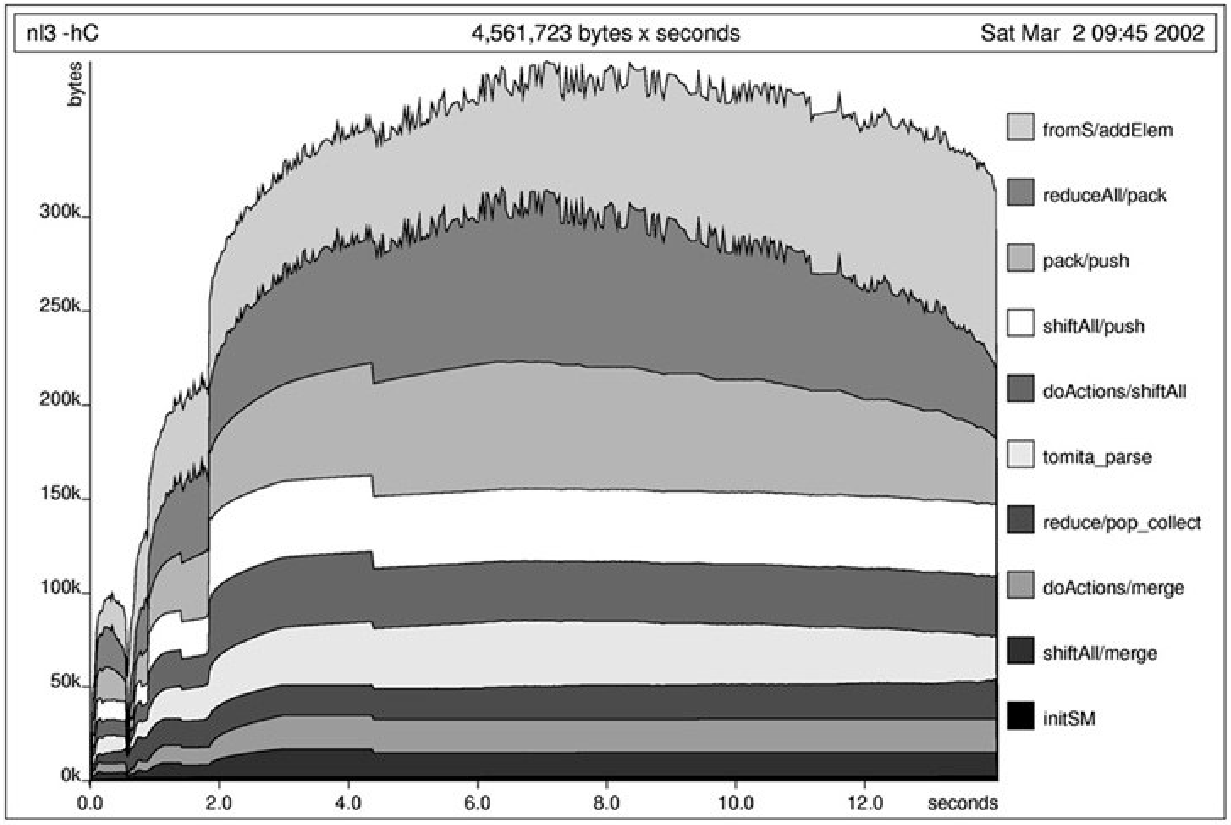


Figure 5.22: Improved heap profile (SimpleNL grammar)

The memory behaviour of the implementation is clearly illuminated by the following profile, generated from a parser for the E+E grammar on an input of around 40 tokens:

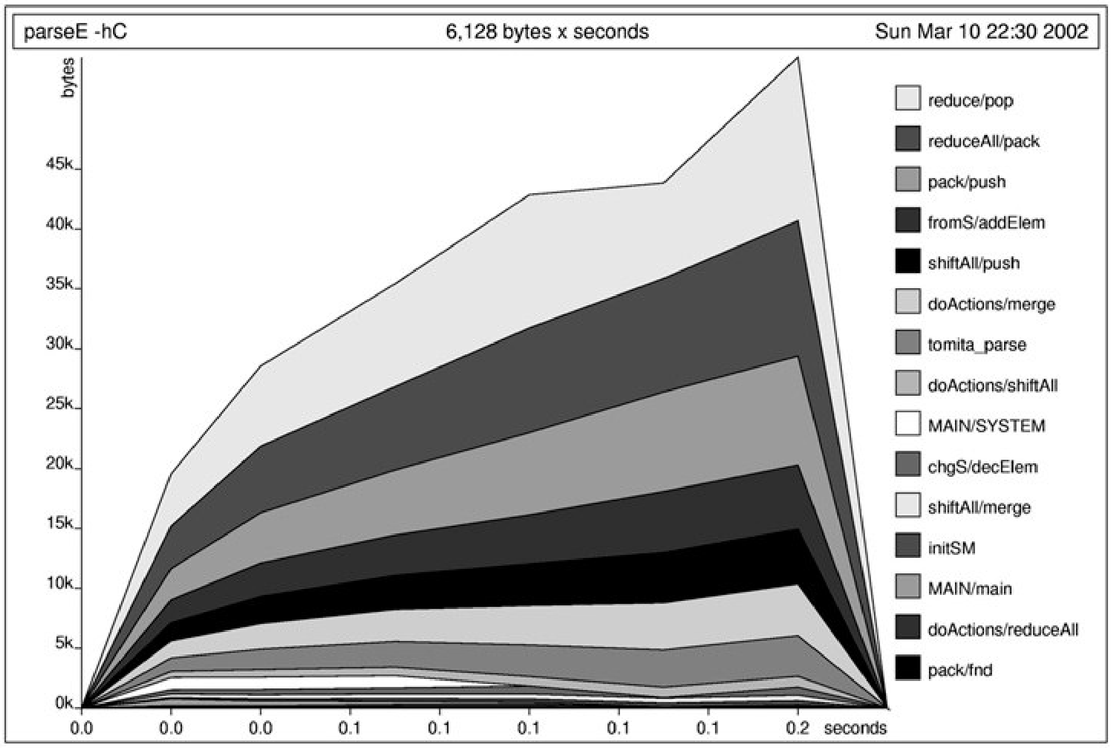


Figure 5.23: Heap profile (E+E grammar)

Initially the heap grows quickly as new forest nodes are generated but are at this stage unlikely to be candidates for subtree sharing. We also find that packing is less common during early stages of the parse, as fewer sub-analyses have already been generated. This rapid growth tails off as the parse progresses, but remains consistent. Finally, we see a sharp decrease as the EOF token in the input is reached and the GSS is recursively reduced to a single `TStack` containing the sentence constituent. Contrast this with the following profile of the parser stripped of its forest-construction mechanism (just a recogniser) running on the same input:

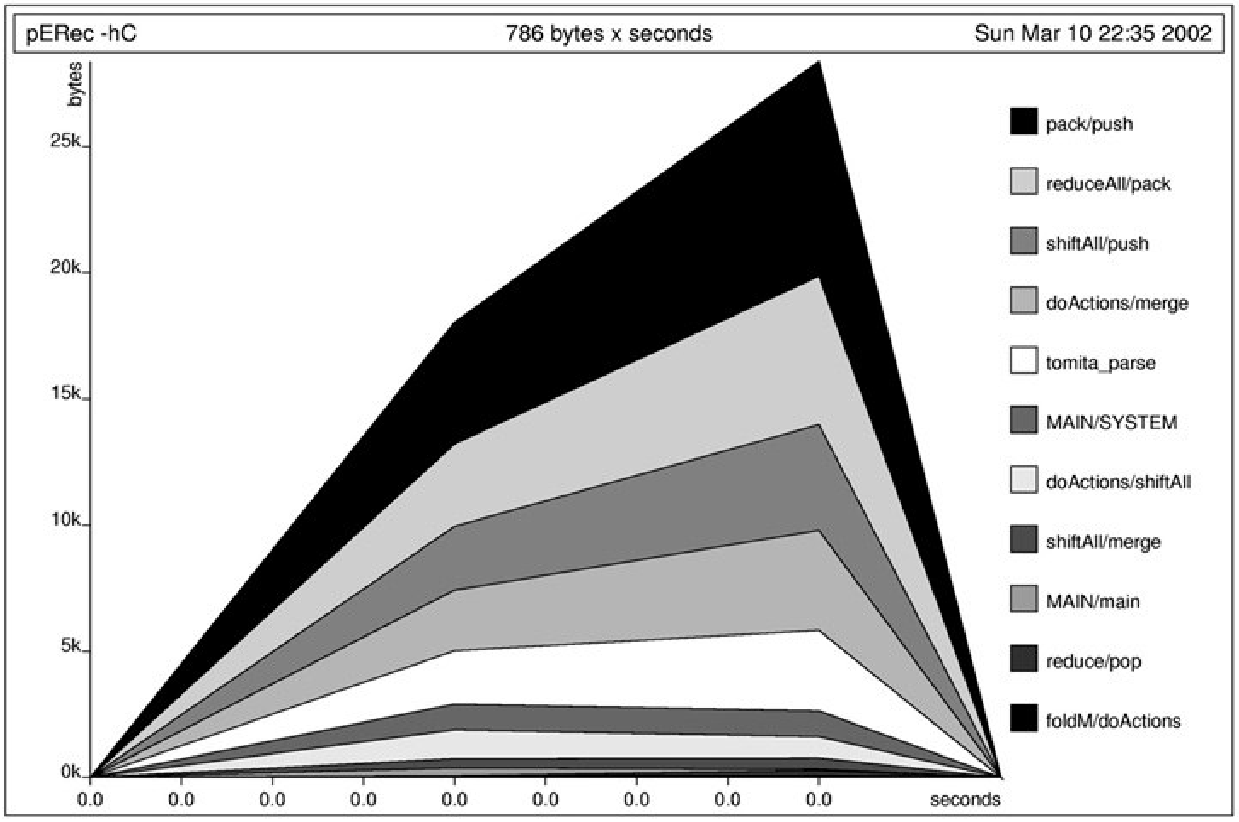


Figure 5.24: Heap profile – recogniser (E+E grammar)

Note that growth is more consistent overall – there is no initial rapid increase as we are not storing forest nodes. Also note that the reduction in overall heap usage is as we would expect.

5.4.4 Comparisons

Due to the recency of the work on Tomita's algorithm, there are few contemporary implementations that we can compare ours with. However, we have been able to obtain a version written in C, and here we assess it for efficiency and quality of code in comparison with our implementation. We also briefly consider the work of Peter Ljunglöf, in which he proposes a purely functional implementation (still in development).

5.4.4.1 C Version

This implementation is available as freeware over the Internet. It creates LR(0) parse tables for specified CF grammars, and produces parse forests for input sentences in the vocabulary of the given grammar. Although the table generation technique is weaker than that used by *HappyGLR*, for a simple grammar such as E+E it produces an identical table. Thus we can provide the same input to both and expect the same results. The following table shows results from timed executions of the two implementations, under the E+E grammar:

Input Length	Execution time (secs)		Nodes in forest	
	Project	C	Project	C
21	0.00	0.00	13	87
41	0.06	0.01	23	272
61	0.24	0.02	33	557
81	0.71	0.04	43	942
121	3.19	0.14	63	2012
161	9.77	0.53	83	3482
201	23.20	1.17	103	5352
241	48.13	2.55	123	7622

Table 5.25: Timed execution comparison

Because of the inherent extra cost of evaluating functional code, we would expect the C implementation to perform significantly faster than our Haskell implementation, and this is affirmed by the results in the table; however this is not really a balanced comparison. The C implementation utilises a different interpretation of subtree sharing, where identical analyses occurring at different stages in the parse are maintained distinctly, reducing the need for equality testing on forest nodes[9]. This increases the speed of the algorithm, but also increases its space complexity in terms of the resulting parse forest (as illustrated by the table). The growth of the number of nodes in the parse forest under the E+E grammar is $O(n)$ for our implementation, whereas for the C implementation it is $O(n^2)$.

It should also be noted that our implementation incorporates certain features such as error-handling, which are absent in the C version.

A significant proportion of the cost of our implementation is incurred by `SetMap` operations, which could be optimised by modelling the internal relation as a search-friendly structure such as a binary tree. A number of other speed increase techniques could be used if necessary, but these must be balanced against maintaining the quality of the code.

[9] Note that we are not discussing the validity of the different approaches to subtree sharing, merely their effect on the efficiency of the algorithm. It should also be noted that our design and implementation models could easily be modified to incorporate the same type of subtree sharing found in the C version, and this would affect its efficiency accordingly.

A final point relates to the quality and concision of the code. Our implementation is significantly shorter than the C version, and demonstrates greater clarity and structural elegance[10], bearing close resemblance to the design model on which it is based. The concision of the code is attested to by the fact that the parser driver for our implementation consists of around 40 lines of code, whereas the C version contains around 160. Including data structures, the figure is around 170 for our implementation and around 310 for the C version. It should be noted that for our implementation this includes two external, reusable ADT's, whereas the data structures for the C version are internal.

5.4.4.2 Ljunglöf's Work

Ljunglöf's work was mentioned in Chapter 3 in relation to the GSS representation. He proposes a purely functional implementation of Tomita's algorithm based on this representation. However, rather than explicitly generating a packed, shared forest, he integrates the forest with the GSS, relying on the compiler/interpreter to handle the internal DAG structure of the GSS as well as the shared elements of the parse forest. This allows elegance of style, but has two significant disadvantages:

- Although the space complexity of the implementation is comparable to Tomita's, the time complexity is exponential in the worst case, as the implementation does not provide a mechanism for local ambiguity packing.
- Because of the implicit nature of the forest, it cannot be directly extracted without the use of impure extensions to Haskell[11].

The developmental status of Ljunglöf's implementation has prevented us from making direct comparisons with ours, although it appears that our implementation may serve as better general-purpose model.

---

[10] Despite the subjectivity associated with the concept of elegance, a visual comparison of the two implementations should convince the reader of the validity of this statement!

[11] See [Lju 02] Chapter 6, esp. *6.6 Discussion* – pg. 91

Ch 6. Conclusions

In this final chapter we consider the achievements of the project in light of the objectives given in Chapter 1 (1.3), and the project deliverables (1.4). We reflect on the success of the project in light of the overall objectives also given in Chapter 1, presenting the advantages and disadvantages of the proposed solution. Finally, we present possible avenues for future development.

6.1 Project Achievements

We will consider in turn each of the major project objectives and discuss whether or not they have been met, and how they relate to specific project deliverables.

6.1.1 Implementation of Tomita's Algorithm in Haskell

This is the main objective of the project and has clearly been met by the implementation modelled in Chapter 3, specified in Chapter 4 and evaluated in Chapter 5. A prototype implementation is part of the basic deliverable for the project (1.4.1), although the final implementation is a result of many subsequent iterative steps of refinement. The advantages and disadvantages of our implementation are discussed in 6.2.1.

6.1.2 Integration with Happy

Integration of the implementation with *Happy* forms part of the intermediate deliverable (1.4.2). Section 3.4 describes how the project meets this objective, and section 4.5 outlines the modifications made to *Happy*. The success of the integration (robustness, transparency etc.) is discussed later.

6.1.3 Analysis and Evaluation

In chapter 5 we analyse and evaluate different aspects of the implementation, looking at both time and space orders as well as results from profiling. This forms part of the advanced deliverable for the project (1.4.3).

6.1.4 Further Possibilities

Chapter 1 mentions a number of other possible developments that could be pursued, mostly relating to the processing of parse forests. There has not been time to pursue these options as part of this project, and they remain for future research (see 6.3).

6.2 Overall Success

Section 1.3.5 states that the overall success of the project depends on:

- Correctness, efficiency and maintainability of the functional implementation of Tomita's algorithm.
- Thorough analysis and evaluation of the implementation.
- Transparent integration with *Happy*.

We will look at each in turn, discussing the advantages and disadvantages of our proposed solutions where appropriate.

6.2.1 Implementation Correctness, Efficiency and Maintainability

In this report we have proposed design and implementation models for a functional version of Tomita's algorithm, written in Haskell. We believe it shows a marked improvement in terms of clarity and maintainability over existing imperative implementations, attested to by its concision and close resemblance to the design model (see 5.4.4). We have presented evidence to suggest that it is correct, although further rigorous analysis would need to be carried out before this could be claimed conclusively. We have carried out tests to show that it is relatively efficient on simple grammars, although as expected, noticeably slower than comparable imperative implementations. A number of further optimisations could be made to narrow this performance gap, but they must be weighed against any ensuing loss of code quality. To summarise, the advantages and disadvantages of our proposed solution include:

Advantages:

- Sound design model, taking advantage of the benefits offered by functional programming.
- Concise, clear and maintainable implementation of the design model.
- Robust, general framework for further development.

Disadvantages:

- Possible improvements needed in terms of time and space efficiency.
- Unclear at this point as to scalability for realistic wide-coverage grammars. Some tests have been carried out on the English grammar used in LOLITA, suggesting that effort needs to be concentrated on efficiently encoding large parse tables.

6.2.2 Analysis and Evaluation

In chapter 5, we provide a reasonably thorough analysis and evaluation of our implementation, fulfilling the criteria for the project. Further analysis could be done on the space complexity of the algorithm, as well as its time complexity in general under an arbitrary grammar.

6.2.3 Integration with Happy

The overall objectives of the project as stated in 1.3.5 include the provision of a transparent integration of our implementation of Tomita's algorithm with the Haskell parser-generator *Happy*. This requires that a user familiar with *Happy* should be able to use *HappyGLR* without difficulty. It is clear from section 3.4 that the mechanism for generating a GLR parser under *HappyGLR* is indeed very similar to that for a standard LR parser. Most of the features available in *Happy* have been incorporated into *HappyGLR*, although a number of features are missing, including the following:

- The provision of error-catching production rules, using the `happyError` mechanism.
- The provision of embedded code to automatically transform parse trees into Haskell expressions.

The first of these could be implemented fairly straightforwardly into the proposed model, but the second presents more of a challenge, requiring research into such issues as *high-level context* in parse forests (see 4.3.3).

6.3 Future Developments

There are many areas of future development related to the work carried out in this project. The parsing mechanism could be improved both in time and space efficiency as well as in such areas as error handling – considering how errors can be usefully and accurately reported. This draws on joint research into the development of functional programming and parsing techniques.

We have already mentioned the possibility of embedding code into GLR parsers under *HappyGLR* (3.4.3). A related area is that of processing complete parse forests once they have been generated. Parse forests allow us to efficiently store large amounts of ambiguous syntactic information, but we are usually required to disambiguate this information before it can be used effectively a specific application domain. For example, in a natural language processing system, we can use an implementation of a parsing technique such as the one proposed in this report to generate a parse forest for an input sentence. The various syntactic representations represented by the forest must then be analysed semantically in order to isolate the intended meaning from the various possibilities.

When considering this type of processing, we seek to utilise the efficiency of the parse forest representation to reduce the amount of computation required; for example, processing shared subtrees multiple times only if their high-level context requires it.

6.4 Concluding Remarks

We conclude by claiming that the overall objectives of the project have been met. We have presented a concise, maintainable functional implementation of Tomita's algorithm that is both correct and reasonably efficient, and have shown this to be the case by analysis and evaluation. We have also presented an integration of our implementation with *Happy*, and successfully used it to generate GLR parsers for a number of ambiguous grammars.

References

[Aho 86] Aho A., Sethi R., Ullman J. (1986): *Compilers – Principles, Techniques and Tools*, Bell Telephone Labs

[Alo 97] Alonso M., Cabrero D., Vilares M. (1997): *A New Approach to the Construction of Generalised LR Parsing Algorithms*, Universidad de La Coruna

[App 98] Appel A. (1998): *Modern Compiler Implementation in Java*, Cambridge University Press

[Bil 89] Billot S., Lang B. (1989): *The Structure of Shared Forests in Ambiguous Parsing*, INRIA and Universite d'Orleans

[But 92] Butler C., (1992): *Computers and Written Texts*, Basil Blackwell Ltd.

[Cal 97] Callaghan P. (1997): *An Evaluation of LOLITA and related Natural Language Processing Systems*, University of Durham

[Car 93] Carroll J. (1993): *Practical Unification-Based Parsing of Natural Language*, University of Cambridge Computer Laboratory

[Cry 87] Crystal D. (1987): *The Cambridge Encyclopedia of Language*, Cambridge University Press

[Dow 85] Dowty D., Karttunen L., Zwicky A. (1985): *Natural Language Parsing – Psychological, Computational, and Theoretical Perspectives*, Cambridge University Press

[Eng 00] English J. (2000): *The Brighton University Resource Kit for Students*

[Ear 70] Earley J. (1970): *An Efficient Context-Free Parsing algorithm*

[Fin 96] Finkel R. (1996): *Advanced Programming Language Design*, Addison-Wesley

[Gai 95] Gaizauskas R., Cunningham H. and Wilks Y. (1995*): A General Architecture for Text Engineering (GATE) -- a new approach to Language Engineering R & D,* Department of Computer Science, University of Sheffield

[Gri 86] Grishman R. (1986): *Computational Linguistics*, Cambridge University Press

[Gru 00] Grune D., Bal H., Jacobs C., Langendoen G. (2000): *Modern Compiler Design*, John Wiley and Sons Ltd.

[Hol 90] Holub A. (1990): *Compiler Design in C*, Prentice-Hall Inc.

[Hun 81] Hunter R. (1981): *The Design and Construction of Compilers*, John Wiley and Sons Ltd.

[Joh 89] Johnson M. (1989): *The computational complexity of Tomita's algorithm*, Proceedings of the 1st International Workshop on Parsing Technologies, Pittsburgh, PA

[Kip 89] Kipps J. (1989): *Analysis of Tomita's algorithm for general context-free parsing*, Proceedings of the 1st International Workshop on Parsing Technologies, Pittsburgh, PA

[Lju 01] Ljunglof P. (2001): Chapter 6, Licentiate Thesis (draft – November 2001)

[Lju 02] Ljunglof P. (2002): Chapter 6, Licentiate Thesis (final draft – March 2002)

[Mar 00] Marlow S., Gill A. (2000): *Happy User Guide*

[Nij 80] Nijholt A. (1980): *Context-Free Grammars: Covers, Normal Forms, and Parsing*, Springer-Verlag

[Pey 99] Peyton-Jones S. (1999): *Report on the Programming Language Haskell 98*

[Pey 00] Peyton-Jones S., et al. (2000): *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*

[Sch 96] Schiehlen M. (1996): *Semantic Construction from Parse Forests*, Institute for Computational Linguistics, University of Stuttgart

[Sik 93] Sikkel K., op den Akker R. (1993): *Predictive Head-Corner Chart Parsing*, GMD – German National Research Center for Information Technology (Sikkel), University of Twente, Computer Science Dept. (Akker)

[Sik 97] Sikkel K., Nijholt A. (1997): *Parsing of Context-Free Languages*, GMD – German National Research Center for Information Technology (Sikkel), University of Twente, Computer Science Dept. (Nijholt)

[Sik 98] Sikkel K. (1998): *Parsing Schemata and Correctness of Parsing Algorithms*, GMD – German National Research Center for Information Technology

[Tan 93] Tanaka H., Tokunaga T., Suresh K., Kentaro I. (1993): *Natural Language Analysis and Generation Technologies*, Department of Computer Science, Tokyo Institute of Technology

[Tho 99] Thompson S. (1999): *Haskell, The Craft of Functional Programming Second Edition*, Addison Wesley

[Tom 85] Tomita M. (1985): *An efficient context-free parsing algorithm for natural languages*, In proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA.

[Wik 01] (2001): *Wikipedia* (free online encyclopaedia), http://www.wikipedia.com/

Appendix A - SetMap ADT:

```
(Module declaration omitted)

%---------------------------------------------------------------------------

> data SetMap a = SM [Int]        -- unused indices
>                 [Element a]     -- the relation

> data Element a = EL (Int,a)     -- key-->value mapping
>                  Int            -- uses of element

%---------------------------------------------------------------------------

> (<>) x y = (x,y)

> initSM :: SetMap a
> initSM = SM [0..] []

> getElem :: Int -> SetMap a -> Maybe a
> getElem i (SM _ rel)
>  = case find (\(EL (k,_) _) -> i==k) rel of
>     Nothing           -> Nothing
>     Just (EL (_,v) _) -> Just v

> addElem :: Eq a => a -> SetMap a -> (Int,SetMap a)
> addElem e sm@(SM ks@(i:is) rel)
>  = case fnd (\(EL (_,v) _) -> v==e) rel of
>     Nothing               -> i <> SM is (EL (i,e) 1 : rel)
>     Just (EL (k,_) j,r) -> k <> SM ks (EL (k,e) (j+1) : r)

> decElem :: Int -> SetMap a -> (Maybe a,SetMap a)
> decElem i (SM is rel)
>  = case fnd (\(EL (k,_) _) -> k==i) rel of
>     Nothing                -> Nothing <> SM is rel
>     Just (EL e@(k,v) 1,r)   -> Just v  <> SM (i:is) r
>     Just (EL e@(k,v) j,r)   -> Just v  <> SM is (EL e (j-1) : r)

> indices :: SetMap a -> [Int]
> indices (SM _ rel) = [ i | EL (i,_) _ <- rel ]

> assocs :: SetMap a -> [(Int,a)]
> assocs (SM _ rel) = [ (k,v) | EL (k,v) _ <- rel ]

%---------------------------------------------------------------------------

> fnd :: (a -> Bool) -> [a] -> Maybe (a,[a])
> fnd _ [] = Nothing
> fnd p (x:xs)  | p x       = Just (x,xs)
>           | otherwise = case fnd p xs of
>                        Just (x',xs) -> Just (x',x:xs)
>                        _          -> Nothing

%---------------------------------------------------------------------------
Show instance

> instance Show a => Show (SetMap a) where
>  show (SM _ rel)
>   = let fn = \(EL e _) -> show e ++ ","
>     in   concat [ "{" , init $ concat (map fn rel) , "}" ]
```

Appendix B - TStack ADT:

(Module declaration omitted)

```
%-----------------------------------------------------------------------------

> data TStack a = TS Int             -- state
>                    Int             -- height
>                    Int             -- ID
>                    [(a,TStack a)]  -- (element on arc , child)

> instance Show a => Show (TStack a) where
>  show (TS st _ id ch) = show st ++ "[" ++ concat (map show ch) ++ "]"

> instance Eq (TStack a) where
>  (TS _ _ v _) == (TS _ _ v' _) = v==v'

> instance Ord (TStack a) where
>  compare (TS _ h _ _) (TS _ h' _ _)
>   | h <= h'   = GT
>   | otherwise = LT

> initTS id = TS 0 1 id []

> push :: a -> Int -> Int -> TStack a -> TStack a
> push x st id stk@(TS _ h _ _) = TS st (h+1) id [(x,stk)]

> pop :: Int -> TStack a -> [([a],TStack a)]
> pop 0 ts           = [([],ts)]
> pop n (TS _ _ _ ch) = [ (xs ++ [x] , stk')
>                       | (x,stk) <- ch
>                       , let rec = pop (n-1) stk
>                       , (xs,stk') <- rec ]

> popF :: TStack a -> TStack a
> popF (TS _ _ _ ((_,c):_)) = c

> top :: TStack a -> Int
> top (TS st _ _ _) = st

> vals :: TStack a -> [a]
> vals (TS _ _ _ ch) = fst $ unzip ch

> height :: TStack a -> Int
> height (TS _ h _ _) = h

> merge :: [TStack a] -> [TStack a]
> merge stks
>  = [ TS st h id ch
>    | st <- nub (map top stks)
>    , let ch = concat [ x | TS st' _ _ x <- stks , st==st'  ]
>          h  = foldl1 max [ x | TS st' x _ _ <- stks , st==st' ]
>          id = head [ x | TS st' _ x _ <- stks , st==st' ]
>    ]
```

Appendix C – ProduceGLRCode module:

```
> module ProduceGLRCode ( produceGLRParser ) where

> import Grammar
> import Array
> import Char ( isUpper )
> import List ( nub )

%-----------------------------------------------------------------------------
File and Function Names

> tomTempl td = td ++ "/Tomita.lhs"
> parseName g = fst3 $ head (starts g)

%-----------------------------------------------------------------------------
General Functions

> fst3 (x,_,_)   = x
> fst4 (x,_,_,_) = x
> snd4 (_,x,_,_) = x

> addArrow str
>  | str == [] || head str == '>'
>     = str
>  | otherwise
>     = "> " ++ str

%-----------------------------------------------------------------------------
Main exported function

> produceGLRParser outfilename template_dir action goto header trailer g
>  = do
>      let name  = takeWhile (/='.') outfilename ++ ".lhs"
>      let gsMap = mkGSymMap g
>      let tbls  = mkTbls action goto gsMap g
>      mkFile name tbls (parseName g) template_dir header trailer g

%-----------------------------------------------------------------------------
Function that generates the file containing the Tomita parsing code.
Most of this code is taken from the template, whose name is given at
the top of this file. Its location is obtained from the main Happy driver.

> mkFile :: FilePath      -- Output file name
>       -> String          -- LR tables - generated by 'mkTbls'
>       -> String          -- Start parse function name
>       -> String    -- Templates directory
>       -> Maybe String  -- Module header
>       -> Maybe String  -- User-defined stuff (token DT, lexer etc.)
>       -> Grammar         -- Happy Grammar
>       -> IO ()
>
> mkFile flname tables start templdir header trailer g
>  = do
>      templ <- readFile (tomTempl templdir)
>      case trailer of
>       Nothing  -> error "Incomplete grammar specification!"
>       Just str -> writeFile flname (content templ str)
>  where
>   content tomitaStr userInfo
>    = unlines [ moduleDec
>              , tomitaStr
>              , parseFn
>              , unlines $ map addArrow (lines userInfo)
>              , unlines [ mkGSymbols g ]
>              , typeForToks
>              , tables ]
>    where
>     parseFn = unlines [ concat [ "> " , start , " = tomita_parse " ] ]
>     typeForToks = unlines [ "> type UserDefTok = " ++ token_type g ]
>     moduleDec
>      = case header of
>       Nothing -> ""
>       Just h  -> unlines [ unlines $ map addArrow (lines h) ]

%-----------------------------------------------------------------------------
> mkTbls :: ActionTable       -- Action table from Happy
>      -> GotoTable           -- Goto table from Happy
>      -> [(Int,String)]      -- Internal GSymbol map (see below)
>      -> Grammar             -- Happy Grammar
>      -> String
> mkTbls action goto gsMap g
>  = unlines [ writeActionTbl action gsMap g
>          , writeGotoTbl goto gsMap ]

%-----------------------------------------------------------------------------
Create a mapping of Happy grammar symbol integers to the data representation
that will be used for them in the GLR parser.

> mkGSymMap :: Grammar -> [(Int,String)]
> mkGSymMap g
>  =  [ (i, (token_names g) ! i)
>     | i <- tail $ non_terminals g ]      -- Non-terminals
>    ++ [ (i, "HappyTok (" ++ mkMatch tok ++ ")")
>     | (i,tok) <- token_specs g ]        -- Tokens (terminals)
>    ++ [(eof_term g,"HappyEOF")]         -- EOF symbol (internal terminal)
>  where
>   mkMatch tok = unwords $ replace "$$" "_" (words tok)

> replace :: String -> String -> [String] -> [String]
> replace thisStr withStr inHere
>  = map (\wrd -> if wrd == thisStr then withStr else wrd) inHere

> toGSym gsMap i
>  = case lookup i gsMap of
>     Nothing -> error "No representation for symbol " ++ show i
>     Just g  -> g

%-----------------------------------------------------------------------------
Take the ActionTable from Happy and turn it into a String representing a
function that can be included as the action table in the GLR parser.

> writeActionTbl :: ActionTable -> [(Int,String)] -> Grammar -> String
> writeActionTbl acTbl gsMap g
>  = unlines [ concat [ mkLines , errorLine ] ]
>  where
>   name      = "action"
>   mkLines   = concat $ map mkState (assocs acTbl)
>   errorLine = concat [ "> " , name , " _ _ = Error" ]
>   mkState (i,arr)
>    = unlines $ filter (/="") $ map (mkLine i) (assocs arr)
>   mkLine state (symInt,action)
>    = case action of
>       LR'Fail     -> ""
>       LR'MustFail -> ""
>       _        -> concat [ startLine , mkAct action ]
>    where
>     startLine
>      = concat [ "> " , name , " " , show state , " (" , getTok , ") = " ]
>     getTok = toGSym gsMap symInt
>   mkAct act
>    = case act of
>       LR'Shift newSt _ -> "Shift " ++ show newSt ++ " []"
>       LR'Reduce r    _ -> "Reduce " ++ "[" ++ mkRed r ++ "]"
>       LR'Accept  -> "Accept"
>       LR'Multiple as _ ->
>      let as' = nub as in
>      case ([ st | LR'Shift st _ <- as' ],[ r  | LR'Reduce r _ <- as' ]) of
>       ([],rs)   -> "Reduce " ++ mkReds rs
>       ([st],rs) -> "Shift " ++ show st ++ " " ++ mkReds rs
>    where
>     rule r  = lookupProdNo g r
>     lhs r   = toGSym gsMap (fst4 $ rule r)
>     arity r = show $ length (snd4 $ rule r)
>     mkRed r = "(" ++ lhs r ++ "," ++ arity r ++ ")"
>     mkReds rs = "[" ++ tail (concat [ "," ++ mkRed r | r <- rs ]) ++ "]"

%-----------------------------------------------------------------------------
Do the same with the Happy goto table.

> writeGotoTbl :: GotoTable -> [(Int,String)] -> String
> writeGotoTbl goTbl gsMap
>  = unlines [ mkLines ]
>  where
>   name    = "goto"
>   mkLines = concat $ map mkState (assocs goTbl)
>
>   mkState (i,arr)
>    = unlines $ filter (/="") $ map (mkLine i) (assocs arr)
>
>   mkLine state (ntInt,goto)
>    = case goto of
>       NoGoto  -> ""
>       Goto st -> concat [ startLine , show st ]
>    where
>     startLine
>      = concat [ "> " , name , " " , show state , " " , getGSym , " = " ]
>     getGSym = toGSym gsMap ntInt

%-----------------------------------------------------------------------------
Create the 'GSymbol' ADT for the symbols in the grammar

> mkGSymbols :: Grammar -> String
> mkGSymbols g = concat [ dec
>                       , tail $ concat [ "| " ++ sym ++ " " | sym <- syms ]
>                       , tok
>                       , eof
>                       , der ]
>  where
>   syms = [ (token_names g) ! i | i <- tail (non_terminals g) ]
>   dec  = "> data GSymbol = "
>   tok  = "| HappyTok " ++ token_type g ++ " "
>   eof  = "| HappyEOF" ++ "\n"
>   der  = ">      deriving (Show,Eq)"
```

Appendix D – GLRDecoder module:

```
> module GLRDecoder ( Tree(..) , trees , prTrees) where

> import SetMap
> import Maybe
> import List
> import <GLR Parser module>

%-----------------------------------------------------------------------------
Data types and structures

> data Tree a = TNode a [Tree a]
>             |
>             TLeaf a
>             deriving Eq

%-----------------------------------------------------------------------------
Show functions

> instance Show a => Show (Tree a) where
>  show tree = showTree 0 tree

> showTree :: Show a => Int -> Tree a -> String
> showTree i (TLeaf nc) = concat [ spc i , show nc , nl ]
> showTree i (TNode nc trees)
>  = concat [ spc i , show nc , nl , showSubTrees ]
>  where
>   showSubTrees = concat $ map (showTree (i+2)) trees

> nl    = "\n"
> spc n = take n $ repeat ' '

%-----------------------------------------------------------------------------
Decoding process

> prTrees :: GLRResult -> IO ()
> prTrees (ParseError es)  = putStr $ "Parse Failed!\n" ++ show es
> prTrees (ParseOK es fSM) = putStr $ show es ++ (unlines $ map show doTrees)
>  where
>   doTrees = trees fSM

> trees :: Forest -> [Tree GSymbol]
> trees fSM = let topNode = head $ indices fSM
>             in  mkTree fSM topNode

> mkTree :: Forest -> Int -> [Tree GSymbol]
> mkTree sm fID
>  = case (fromJust $ getElem fID sm) of
>     FNode nc []  -> [TLeaf nc]
>     FNode nc fss -> mkNode nc sm fss

> mkNode :: GSymbol -> Forest -> [[Int]] -> [Tree GSymbol]
> mkNode nc sm fss
>  = [ TNode nc ts
>    | fs <- fss
>    , ts <- cross_prod (map (mkTree sm) fs) ]

> cross_prod :: [[a]] -> [[a]]
> cross_prod []       = [[]]
> cross_prod (as:ass) = [ b:bs | b <- as , bs <- cross_prod ass ]
```

Appendix E – ForestToDV / TreesToDV modules:

(Skeleton code provided by Paul Callaghan, used with permission)

```
> module ForestToDV (toDV) where

> import SetMap

> import Main

> import DaVinciTypes hiding (Edge(..) , Node(..))
> import qualified DaVinciTypes (Edge(..) , Node(..))
> import DaVinciAttributes

--

> show_gsymbol (HappyTok x) = show x
> show_gsymbol t            = show t

> g2n (n, FNode x [bs])
>  = mk_box id (show_gsymbol x)
>  $ [ DaVinciTypes.R (NodeId $ show j) | j <- bs ]
>  where
>   id = show n

> g2n (n, FNode x bss)
>  = mk_box id (show_gsymbol x)
>  $ [ mk_circle (id ++ "." ++ show i) ""
>                [ DaVinciTypes.R (NodeId $ show j)
>                | j <- js ]
>    | (i,js) <- zip [0..] bss ]
>  where
>   id = show n

---
the following create daVinci node representations

> mk_box = mk_node box_t
> mk_circle = mk_node circle_t
> mk_plain = mk_node text_t

> mk_node a id nm ts
>  = DaVinciTypes.N (NodeId id) (Type "") [a,text nm]
>  $ [ (mk_edge id n) t | (n,t) <- zip [1..] ts ]

> mk_edge id child_no t@(DaVinciTypes.R (NodeId id2))
>  = DaVinciTypes.E (EdgeId eId) (Type "") [] t
>  where
>   eId = concat [id,":",id2,"(",show child_no,")"]

> mk_edge id child_no t@(DaVinciTypes.N (NodeId id2) _ _ _)
>  = DaVinciTypes.E (EdgeId eId) (Type "") [] t
>  where
>   eId = concat [id,":",id2,"(",show child_no,")"]

> toDV :: String -> IO ()
> toDV s
>  = do  case (doParse s) of
>         ParseOK _ f -> let dv = show $ map g2n $ assocs f
>                        in  writeFile "out.daVinci" dv
>         _           -> error "Parse Error!"
> module TreesToDV (toDV) where

> import SetMap

> import GLRDecoder
> import Main

> import DaVinciTypes hiding (Edge(..) , Node(..))
> import qualified DaVinciTypes (Edge(..) , Node(..))
> import DaVinciAttributes

--

> show_gsymbol (HappyTok x) = show x
> show_gsymbol t            = show t

> tree2DVNode tree_id = tree2DVNode_ (show tree_id ++ "R")

> tree2DVNode_ id (TLeaf x)
>  = [ mk_box id (show_gsymbol x) [] ]

> tree2DVNode_ id (TNode x children)
>  = thisNode : concat  [ tree2DVNode_ (id ++ show i) c
>                 | (i,c) <- zip [1..] children ]
>  where
>   thisNode  = mk_box id (show_gsymbol x) childRefs
>   childRefs =
>    [ DaVinciTypes.R (NodeId $ id ++ show i) | (i,c) <- zip [1..] children ]

---
the following create daVinci node representations

> mk_box = mk_node box_t
> mk_plain = mk_node text_t

> mk_node a id nm ts
>  = DaVinciTypes.N (NodeId id) (Type "") [a,text nm]
>  $ [ (mk_edge id n) t | (n,t) <- zip [1..] ts ]

> mk_edge id child_no t@(DaVinciTypes.R (NodeId id2))
>  = DaVinciTypes.E (EdgeId eId) (Type "") [] t
>  where
>   eId = concat [id,":",id2]

> toDV :: String -> IO ()
> toDV s
>  = do
>     let p = doParse s
>     case p of
>      ParseOK _ f ->
>       do
>         let ts = trees f
>         writeFile "out.daVinci" $ show (dvTrees ts)
>      _ -> error "Parse Error!"
>  where
>   dvTrees ts = concat $ [ tree2DVNode i t | (i,t) <- zip [1..] ts ]
```

Appendix F – SimpleNL grammar specification:

```
{
module Main ( main
            , doParse
            , GLRResult(..)
            , Forest
            , ForestNode(..)
            , GSymbol(..)
            , GLRResult(..)
            )
where

import System
import NLToken  <defined in a separate module>
import NLLexer  <defined in a separate module>
}

%name parse
%tokentype { NLToken }

%token
      n     { Noun $$ }
      vs    { StateOfBeingVerb $$ }
      va    { ActionVerb $$ }
      pro   { Pronoun $$  }
      det   { Det $$ }
      prep  { Prep $$ }
      adj   { Adjective $$ }
      adv   { Adverb $$ }
      con   { Conjunction $$ }
      ','   { Comma }
%%

S      : SP VP          { }
       | S con S        { }

SP     : NP             { }

VP     : AVP            { }
       | SBVP           { }

AVP    : AV             { }
       | AV O ADVP      { }  -- I saw him in the park / I hit him hard
       | AV IDO O       { }

AV     : ADVP va ADVP   { }

SBVP   : vs             { }
       | vs O           { }  -- am a cold person
       | vs adj         { }  -- am cold
       | vs PP          { }  -- am in the bath

ADVP   : adv con ADVP   { }  -- gently and carefully
       | adv ADVP       { }
       | PP ADVP        { }
       |                { }

O      : NP             { }

IDO    : NP             { }
```

```
NP    : pro            { }
      | det N          { }
      | NP PP          { }

N     : N N            { }
      | ADJ N          { }
      | n              { }

ADJ   : adj ',' ADJ    { }
      | adj ADJ        { }
      | adj            { }

PP    : prep NP        { }

{
doParse = parse.nLLexer

main :: IO ()
main = do
         s:_ <- getArgs
         case parse (nLLexer s) of
          ParseOK es f  -> putStr $ "OK: " ++ show es ++ "\n" ++ show f ++ "\n"
          ParseError es -> putStr $ "ERROR:" ++ show es ++ "\n"
}
```