UPTEC IT 14 016
Examensarbete 30 hp
Oktober 2014

A Comparison Between Packrat Parsing and Conventional Shift-Reduce Parsing on Real-World Grammars and Inputs

Daniel Flodin

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Comparison Between Packrat Parsing and Conventional Shift-Reduce Parsing on Real-World Grammars and Inputs

Daniel Flodin

Packrat parsing is a top-down, recursive descent parsing technique that uses backtracking and has a guaranteed linear parse time. Conventional backtracking parsers suffer from exponential parse times in the worst case due to re-evaluating redundant results. This is avoided in packrat parsers with the use of memoization. However, memoization causes packrat parsers memory consumption to be linearly proportional to the input string, as opposed to linearly proportional to the maximum recursion depth for conventional parsing techniques.

The objective of this thesis is to implement a packrat parser generator and compare it with an existing and well-known parser combination called Lex/Yacc which produces shift-reduce parsers. The comparison will consist of pure performance measurements such as memory consumption and parsing time, and also a more general comparison between the two parsing techniques.

The conclusion made from the comparison is that packrat parsing can be a viable option due to its ability to compose modular and extendible grammars more easily than Lex/Yacc. However, from a performance perspective the Lex/Yacc combination proved superior. In addition, the results indicate that similar performance for a packrat parser is hard to achieve on grammars similar to those used in this thesis.

Handledare: Johan Runesson
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Roland Bol
ISSN: 1401-5749, UPTEC IT 14 016
Tryckt av: Reprocentralen ITC

Summary in Swedish

Packratparsning är en parsingsteknik som togs fram av Bryan Ford 2002. Packratparsning är en så kallad top-down parsningsteknik som använder sig utav backtracking. Konventionella top-down parsers som använder backtracking kan i värsta fall ha exponentiell körtid på grund av att de utför redundanta evalueringar. Packratparsers undviker detta genom *memoisering*. Memoiseringen innebär att alla evalueringar som görs under parsningen sparas undan i en separat tabell för att sedan kunna användas utifall samma evalueringar behöver upprepas. Om införande- och uppslagningsoperationerna på memoiseringstabellen kan ske i konstant tid så innebär detta att en packratparser kommer ha en garanterad linjär körtid.

Packratparsers skiljer sig även från mer traditionella parsningsmetoder genom att packratparsers kan vara *scannerless*. Med detta menas att packratparsers ej behöver genomföra den lexikaliska analysen separat. Både den lexikaliska analysen och parsningen kan ske med ett enda verktyg.

Packratparsers använder sig utav parsing-expressiongrammatiker som alltid producerar otvetydiga grammatiker. Detta skiljer så från de mer traditionella context-free grammatikerna som i vissa fall kan producera tvetydiga grammatiker. Packratparsers stödjer även syntaktiska predikat som ger packratparsers möjligheten att använda sig utav en obegränsad lookahead.

I denna rapport jämförs packratparsning med parserkombinationen Lex/Yacc. Lex/Yacc är en parsergenerator som producerar så kallade shift-reduce parsers. Företaget bakom detta examensarbete, IAR Systems, använder i dagsläget Lex/Yacc för parsning av många av deras egna implementerade språk. Dock så uppelever dem att Lex/Yacc har en del brister som till exempel att specifkationerna för Lex/Yacc följer olika syntax, felhanteringen måste implementeras manuellt, svårigheter att ha multipla parsers i samma program och att det generellt sett är svårt att utöka och ändra grammatikspecifikationerna för de två verktygen. IAR Systems är därför intresserade av utifall packratparsning kan hantera dessa brister på ett bättre sätt. Dock får inte eventuella presentandaskillnader mellan dessa två parsningstekniker vara för stora.

För att undersöka detta så har en packratparsergenerator vid namn Hilltop implementerats. Hilltop och Lex/Yacc har genererat parsers för samma grammatiker och dessa parsers har sedan analyserats prestandamässigt; sett till körtid och minnesförbrukning. Det har även gjorts en generell jämförelse mellan Hilltop och Lex/Yacc för att belysa eventuella för- och nackdelar de olika parsningsteknikerna medför. För att få ett så realistikt resultat som möjligt så har grammatikerna som parsergeneratorerna testats på varit grammatiker som IAR Systems använder i produktion. Källkodsfilerna som de genererade parserarna testats på är även dessa tagna från produktionen.

Föga förvånande så framkommer det under resultatanalysen att parsers genererade utav Hilltop skiljer sig prestandamässigt, både i avseende för körtid och minnesanvändning, gentemot parsers genererade utav Lex/Yacc. Denna skillnad i prestanda beror delvis på att implementationen utav Hilltop endast har pågått under en begränsad tid vilket resulterat i en del noterbara brister, bland annat hur strängjämförelser hanteras. I sitt nuvarande tillstånd så skiljer sig parsers genererade utav Hilltop gentemot Lex/Yacc med en faktor 18.4-25.9 med hänsyn till körtid, och en faktor 25.9-34.9 vad gäller minnesåtgång. Dock så tros dessa faktorer kunna minska väsentligt med hjälp av diverse förbättringar på implementationen. Skillnaderna tros kunna reduceras så mycket så att det istället skiljer en faktor två eller tre mellan genererade parsers. Packratparsers tros dock inte kunna bli prestandamässigt bättre än shift-reduce parsers genererade utav Lex/Yacc, i alla fall inte för grammatiker liknande de som har använts i detta examensarbete.

Vissa av de brister som Lex/Yacc kombinationen innehar hanteras på ett bättre sätt utav Hilltop. Främst så är Hilltops och packratparsers förmåga att kunna modulärisera och utöka/ändra

sina grammatiker klart bättre än för grammatiker specificerade med Lex/Yacc.

Bristerna relaterat till felhantering och möjligheten att ha flera parsers i samma program är för tillfället ej implementerat i Hilltop. Dock kan en generell felhanteringen implementeras och därmed behöver användaren ej implementera någon felhantering manuellt. Även stöd för att ett program ska kunna innehålla multipla parsers skall kunna implementeras utan större svårigheter.

Förmågan att enkelt kunna modulärisera och utöka sina grammatiker med Hilltop och packratparsers gör att packratparsergeneratorer passar bra om nya liknande grammatiker implementeras eller om grammatiker omdefineras på en regelbunden basis. Detta gör att en packratparsergenerator kan vara att föredra inom detta användningsområde. Däremot, om användaren är mer intresserad av ren prestanda utav sin parser och inte regelbundet ändrar eller skapar nya grammatiker så står sig parsergeneratorkombinationen Lex/Yacc sig väldigt starkt, och är i detta fall att föredra gentemot en packratparsergenerator.

Contents

Acronyms

AST *abstract syntax tree.*

CFG *context-free grammar.*

DCG *definite clause grammar.*

DSL *domain-specific language.*

GTDPL *generalized top-down parsing language.*

LALR *look-ahead LR.*

LL *left-to-right leftmost derivation.*

LR *left-to-right rightmost derivation.*

PEG *parsing expression grammar.*

RE *regular expression.*

TS *TMG recognition schema.*

Chapter 1

Introduction

1.1 Background

Parsing is a thoroughly researched topic in computer science. The initial research regarding parsing focused on parsing natural (human) languages [13]. Due to the nature of natural language, these methods were designed to effectively parse ambiguous grammars. It was discovered that the methods used for parsing natural languages could also be used for parsing machine-oriented (programming) languages [13]. However, programming languages are usually not designed to be ambiguous.

Parsing is usually split into two phases: *lexical analysis* and *parsing*. The lexical analysis divides the input text (string) into smaller parts, so-called *tokens*. These tokens are then sent sequentially to the parser.

During parsing, the parser uses a grammar to determine whether the input string is part of the accepting language or not. The language of the grammar is defined through a set of grammar rules or productions. Each production can then in turn consist of several different choices. The parser uses these productions throughout the parsing to decide whether to accept the input string or to reject it.

Top-down parsing is a parsing technique that performs a *left-to-right leftmost derivation* (LL) of the input string. This can be done with either prediction, backtracking or a combination of the two. A top-down parser that uses prediction (LL(k)) bases its decisions on *lookahead*, where the parser is able to "look ahead" k number of symbols of the input string. A top-down parser that uses backtracking instead evaluates each production and its choices in turn; if a choice/production fails the parser backtracks on the input string and evaluates the next choice/production, if the choice/production succeeds the parser merely continues.

Bottom-up parsing is a parsing technique that instead performs a *left-to-right rightmost derivation* (LR) of the input string. A commonly used bottom-up parsing technique is called *shift-reduce* parsing. A shift-reduce parser uses two different actions during parsing: *shift* and *reduce*. A shift action takes a number of symbols from the input string and places them on a stack. The reduce action reduces the symbols on the stack based on finding a matching grammar production for the symbols. The decisions regarding whether to shift or reduce are done based on lookahead.

1.1.1 Packrat Parsing

Several different parsing techniques have been developed over the years, both for parsing ambiguous and unambiguous grammars. One of the latest is packrat parsing [10]. Packrat parsing is a top-down recursive descent parsing technique that uses backtracking and has a guaranteed linear parse time. This differs from conventional top-down backtracking algorithms which suffer from exponential parsing time in the worst case. This exponential runtime is due to performing redundant evaluations caused by backtracking. Packrat parsers avoids this by storing all of the evaluated results to be used for future backtracking. Thus, the backtracking procedure performs no redundant computations. This storing technique is called *memoization* and is one of the reasons for the guaranteed linear parsing time for packrat parsers. The memory consumption for conventional parse algorithms is linear to the size of the *maximum recursion depth* occurring during the parse. This can in the worst case be the same as the size of the input string. However, in practice the maximum recursion depth is significantly smaller than the size of the input string. In contrast, with the use of memoization, the memory consumption for a packrat parser is linearly proportional to the size of the input string.

Packrat parsing uses *parsing expression grammar*s(PEGs) which always produce unambiguous grammars. It has been proven that *all* LL(k) and LR(k) grammars can be rewritten into a PEG [13]. Thus, packrat parsing is able to parse all context-free grammars. In fact, it can even parse some grammars that are non-context-free [13].

Another characteristic of packrat parsing is that it is *scannerless*. This means that there is no need for a separate lexical analyser and a separate parser. In packrat parsers, they are both integrated in the same tool, as opposed to the Lex[20]/Yacc[19] approach where Lex provides the lexical analysis and Yacc contributes with the parsing.

1.2 Problem Description

The contribution of this thesis is to compare packrat parsing with the more traditional shift-reduce parsing tools Lex and Yacc. The reason for choosing Lex/Yacc is due to the fact that the company behind this thesis is IAR Systems [35] and Lex/Yacc are their currently used tools for parsing. Lex/Yacc produce powerful and efficient parsers for a subset of context-free grammars, namely *look-ahead LR* (LALR)(1) grammars. LALR(k) parsers uses the same actions as a shift-reduce parser but uses a deterministic state machine during parsing [1]. However, Lex/Yacc are two rather old tools (mid 1970's) and they suffer from several disadvantages:

- Yacc only supports one token of lookahead (LALR(1)).
- The source code for Lex and Yacc are specified in separate files and follow different syntax.
- Hard to debug. The error handling needs to be implemented manually which can be a non-trivial task to make it report useful error messages. This can lead to issues regarding ambiguity detection.
- The generated parser produce global identifiers which may cause conflicts if a program is designed to parse multiple languages.
- Troublesome to find compatible versions of the tools and few versions are actively maintained.
- Generally hard to extend. Adding a new production may introduce conflicts for unrelated productions.
- Hard to modularise, i.e., the whole grammar needs to be contained in the same specification file.

It is believed that packrat parsing will be slower and use more memory than the shift-reduce parsing technique used by Lex/Yacc. However, packrat parsing may still be a favourable choice if some of these disadvantages can be addressed. These comparison results will be thoroughly analysed and discussed in this paper to test the hypothesis:

Hypothesis. *When comparing packrat parsing with shift-reduce parsers produced by Lex and Yacc, on realistic grammars and inputs, the increased memory consumption and parsing time will be acceptably small.*

Most of the research regarding packrat parsing performance has been focusing on parsing Java source files [12, 15, 24, 33]. IAR Systems, however, is more interested in how packrat parsers fare on a *domain-specific language* (DSL) and an assembly language.

1.3 Method

1.3.1 Literature Review

Relevant literature was chosen and studied to gain a deeper understanding regarding packrat parsing, parser generators and parsing in general. Due to parsing being a subject researched by a vast amount of people during the last decades, finding relevant information was not a large problem. Even for packrat parsing which is a relatively new parsing technique (2002), there has been a sufficient amount of research for this thesis to be able to effectively evaluate both its beneficial and non-beneficial characteristics.

1.3.2 Implementation

The implementation part of this thesis involved implementing packrat parsers for different grammars in C++. To avoid having to implement a separate parser for each different grammar, a parser generator prototype named Hilltop has instead been implemented. The purpose of Hilltop was to produce a packrat parser for an arbitrary grammar.

Hilltop has been based on an existing packrat parser generator written in the programming language Ruby that produces packrat parsers in Ruby. This parser generator was altered in a way that allowed it to produce the desired C++ packrat parsers.

1.3.3 Analysis

To analyse the performance of packrat parsers compared with parsers generated by Lex/Yacc; the parser generators received grammars for the same accepting language and were tested on the same input. Since packrat parsers uses a different grammar than Lex/Yacc the grammars had to be specified for both cases. The size of the input varied to visualise how the parsing time for the different parsers scaled. To gain a result that was as realistic as possible, the parsers received inputs and grammars that were used in production at IAR Systems.

The produced packrat parsers were tested with and without memoization to illustrate how it affected parse time and memory consumption.

1.4 Limitations

As mentioned above, an already existing packrat parser generator was used. This was to avoid having to implement the whole parser generator from scratch, thus decreasing the amount of time needed for the implementation part. This effectively increased the amount of time that can be spent on the analysis part of the thesis.

1.5 Thesis Outline

- Chapter 2 discusses parsing in general and explains basic concepts such as regular expressions, context-free grammars, top-down parsing and bottom-up parsing.
- Chapter 3 introduces the theory behind packrat parsing to the reader, involving topics such as PEGs, memoization, how linear parse time is ensured and practical limitations for packrat parsers.
- Chapter 4 includes a brief introduction to Lex/Yacc.
- Chapter 5 describes basic implementation details of Hilltop and a general comparison between packrat parsing and shift-reduce parsing based on information regarding Hilltop and Lex/Yacc.
- Chapter 6 contains the experimental results focusing on parsing time and memory consumption for parsers generated by Hilltop and Lex/Yacc.
- Chapter 7 brings forth related research on the thesis subject.
- Chapter 8 concludes the thesis based on the results.
- Chapter 9 discusses possibilities for future work.

Chapter 2

Parsing

Syntax analysis or *parsing* is used to determine the syntactical structure of the source file (input string). Commonly, this is done with the use of a *grammar*. The grammar uses different *rules* or *productions* to represent this syntactical structure. These productions define the set of input strings (which can be an infinite amount) that the parser accepts. However, determining the syntactical structure of an input is not always a straightforward task due to ambiguousness. If a grammar is ambiguous the input can be represented with different syntactical structures; which structure that is to be chosen needs to be decided by the parser.

One possible approach to decide whether the input string is to be accepted or not is to *tokenize* the input string into smaller parts. For example, when parsing a programming language, the tokenization part consists of splitting up the input string into smaller pieces such as identifying keywords: `if`, `else`, `int`; finding variable names or numbers: `var1`, `var2`, `14256` etc.

These tokens are often times created with the assistance of a *lexical analyser*. The tokens are then sent in sequence to the parser and it is the responsibility of the parser to determine if all of these combined tokens are part of the *accepting language*.

2.1 Regular Expressions

One way of specifying these rules are with the use of *regular expression*s(REs). The rules of an RE can be used to check if an input string is part of its language, or it can be used to generate all possible accepted strings. The following are the standard rules for a RE:

- **Empty string:** ϵ, denotes the RE for the empty string, i.e., no characters.
- **Literal character:** a, denotes the RE for the single character a.
- **Concatenation:** e_1e_2, denotes the RE of all concatenations of a string from the language defined by the RE e_1 with a string from the language defined by the RE e_2.
- **Choice:** $e_1|e_2$, denotes the union of all possible strings able to be generated through the languages defined by the REs e_1 and e_2.
- **Kleene star:** e_1^*, denotes the repetition of the RE e_1 zero or a finite amount of times.

The following examples illustrates how these rules can be used:

$a|b|\epsilon \rightarrow \{a, b\}$

$e_1 = \{ab, cd\}, e_2 = \{ef, gh\}$, $e_1e_2 \rightarrow \{abef, abgh, cdef, cdgh\}$

$a^* \rightarrow \{\epsilon, a, aa, aaa, \dots\}$

Other rules exists but they are considered as syntactic sugar to the standard rules. For instance, the *optional* operator $e_1?$ represents the RE e_1 expressed zero or one time, this can be rewritten into the equivalent RE $(e_1|\epsilon)$. Another commonly used operator is the e_1^+ which denotes the repetition of the RE e_1 one or a finite amount of times, this can be rewritten into the equivalent RE $e_1e_1^*$.

2.2 Context-Free Grammars

There are, however, limitations regarding REs. They lack the support for recursively defined rules. This is a serious limitation due to the fact that most of the practical languages used needs recursive rules to express its language [12]. *Context-free grammar*s(CFGs) on the other hand are able to define recursive rules. A CFG rule uses the following form:

$N \rightarrow n_1 \mid \cdots \mid n_n$

N is a *nonterminal* symbol and each n_i consists of a sequence of *terminals* and/or nonterminals. Each n_i represents a possible *expansion* of the left-hand side nonterminal. This means that a nonterminal N can have itself as an expansion, thereby creating a recursively defined expression. Terminal symbols will be written with small characters, e.g., a, and nonterminal symbol will be defined with upper case characters, e.g., A. Symbols that are not letters will be interpreted as terminals. The starting symbol of a CFG is the left-hand side nonterminal of the first rule.

To illustrate the usage of a CFG, consider this trivial CFG that recognizes matching parenthesis:

$P \rightarrow PP \mid (P) \mid \epsilon$

The first expansion of P represents zero or a finite amount of parenthesised expressions concatenated, e.g., '()()()'. The second expansion of P denotes nested parenthesis, e.g., '((()))'. The last expansion of P, ϵ, is there for termination reasons.

Similar to REs, CFGs can contain ambiguities. This can be problematic when such a grammar is used for parsing because the increased amount of potential valid parsing alternatives can grow exponentially. And in the worst case, all of these alternatives may need to be evaluated [12]. In addition, it is a non-trivial task to discover if a CFG is ambiguous or not. In general, it is an undecidable problem [1].

2.3 Bottom-Up Parsing

One parsing technique is called *bottom-up*. Bottom-up parsing is done by going from the low-level rules (terminals) and proceeding upwards to the top-most levels (nonterminals), hence the name bottom-up. A bottom-up parser tries to find substrings of the input string that matches *right-hand side* productions of the grammar. If such a production is found, the substring

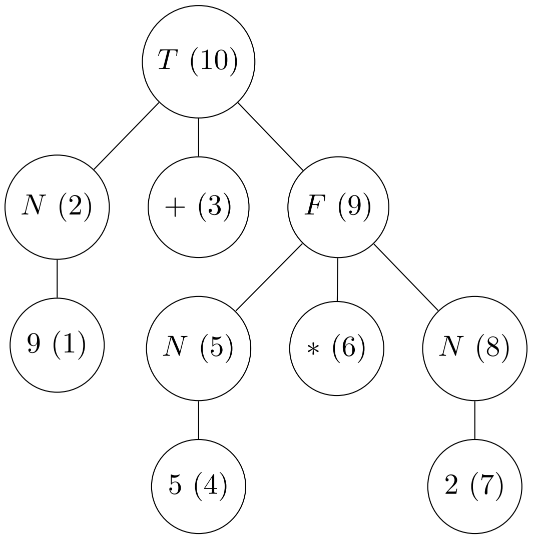

Figure 2.1: A bottom-up parse tree of parsing the string `9+5*2`. The numbers in parenthesis indicate in which order the nonterminal or terminal is evaluated. For brevity, $Term$, $Factor$ and $Number$ are expressed as T, F and N respectively.

gets *reduced*, i.e., the substring gets replaced with the nonterminal at the *left-hand side* of the production. If the whole input string has been reduced to the start symbol, the parsing is a success.

Consider the following grammar, slightly altered from [10]:

Term	→	Term '+' Factor | Factor
Factor	→	Factor '$*$' Number | Number
Number	→	0 | $\cdots$ | 9

When a parser parses a string, it constructs a *parse tree*. In the case of a bottom-up parser this parse tree is constructed from the left-most bottom leaf and continues to the right and leaves the root node (start symbol) for last, a LR of the input string. Figure 2.1 represents the parse tree of a bottom-parser for the string `9+5*2` parsed with the above grammar.

2.3.1 Shift-Reduce Parsing

A *shift-reduce* parser performs two actions on the input string during parsing, *shift* and *reduce*. The reduce action is the same as previously explained. The shift action advances one symbol on the input string and places the symbol on a stack. This stack is used to keep track of which terminals and/or nonterminals that are yet to be reduced. If the only symbol on the stack is the start symbol and the whole input string has been consumed, the parsing is successful.

An illustration of parsing the input string `9+5*2` with the above grammar with a shift-reduce parser is shown in Table 2.1.

At the ninth step when the contents of the stack is $T{+}F$ and the remainder of the input string is $*2$, there are two possible actions that can be taken. The first alternative is to *reduce* $T + F$ with the production $T \rightarrow T + F$, leaving T on the stack and $*2$ as remainder. The parser would then shift two times and perform one reduce, producing $T * N$ on the stack. However, the parser is now unable to proceed due to no matching right-hand side production exists. The other alternative, which yields the successful result, is to *shift*. The issue regarding whether to shift or reduce is called a *shift-reduce* conflict. This can sometimes be avoided with the use of

Stack	Input	Action
	9+5*2$	shift
9	**+5*2$**	reduce $N \rightarrow 0 \mid \cdots \mid 9$
N	**+5*2$**	reduce $F \rightarrow N$
F	**+5*2$**	reduce $T \rightarrow F$
T	**+5*2$**	shift
T+	**5*2$**	shift
T+5	***2$**	reduce $N \rightarrow 0 \mid \cdots \mid 9$
T+N	***2$**	reduce $F \rightarrow N$
T+F	***2$**	shift
T+F*	**2$**	shift
T+F*2	$	reduce $N \rightarrow 0 \mid \cdots \mid 9$
T+F*N	$	reduce $F \rightarrow F * N$
T+F	$	reduce $T \rightarrow T + F$
T	$	accept

Table 2.1: A shift-reduce parsing of the string `9+5*2` with the grammar from Section 2.3. For brevity, *Term*, *Factor* and *Number* are expressed as *T*, *F* and *N* respectively. $ is the symbol for the end of the string.

lookahead. In this particular example the parser would be able to perform the correct action given a lookahead of one symbol. The parser would then discover the $'*'$ symbol and the reduce action would be discarded.

2.4 Top-Down Parsing

Another parsing method is called *top-down* parsing. This technique does instead start at the high-level (nonterminals) and works its way downwards (to the terminals), hence the name top-down.

The produced parse tree from a top-down parser has the same structure as a bottom-up parse tree. However, the order of which the nodes are evaluated differs. A top-down parser performs a LL of the input string. For comparisons with the bottom-up approach, Figure 2.2 displays the top-down parse tree from parsing the string `9+5*2` with the grammar specified in Section 2.3.

There exists a general form of top-down parsing called *recursive descent* [1]. A useful characteristic of a recursive descent parser is that its parsing procedures closely match the grammar productions. This makes the implementation (with use of recursion) of a recursive descent parser fairly straightforward. This differs from shift-reduce parsers which may need cumbersome and non-trivial functions and state machines inside its parser. Therefore, shift-reduce parsers are usually created with the assistance of a *parser generator* [33], which is a tool that creates a parser automatically from a given grammar.

A recursive descent parser can use prediction or backtracking. These two techniques are described in Section 2.4.1 and 2.4.2.

2.4.1 Backtrack Parsing

A backtracking parser takes a substring of the input string (depending on how much of the input string that has already been parsed) and tries a legal production. If the production fails, the parser "backtracks" on the substring (which may now be longer than previously) and tries the next available production. In the worst case, this can lead to having the same production to

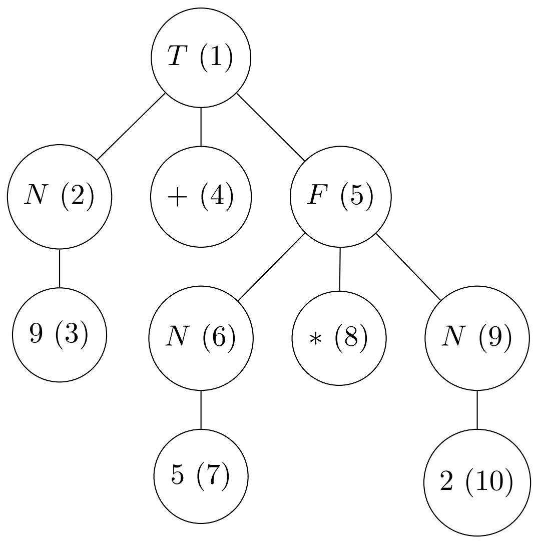

Figure 2.2: A top-down parse tree of parsing the string `9+5*2`. The numbers in parenthesis indicate in which order the nonterminal or terminal is evaluated. For brevity, $Term$, $Factor$ and $Number$ are expressed as T, F and N respectively.

be evaluated multiple times for the same substring of the input. Thus, the worst case parsing time can grow exponentially proportional to the length of the input string. Therefore, a naive backtracking parser is considered inefficient and are rarely used in practice [12].

2.4.2 Predictive Parsing

A more sophisticated strategy is to use predictions. A predictive parser is able to "look ahead" a fixed number of tokens from the input substring and can therefore more accurately guess what production that should be chosen. Consider the grammar defined in Section 2.3; for the parser to be able to know whether the input consists of a $Term$ or a $Factor$, the parser must be able to look ahead one token to check for a potential $'+'$ or $'*'$. In this particular case the grammar can be rewritten into an LL(1) grammar and thus parsed by a predictive parser with the use of two different steps.

The first step is to change the ordering of the right-hand side nonterminals (to avoid left recursion which is explained in Section 2.4.3). This would produce the following grammar:

$$\begin{array}{lcl} \text{Term} & \rightarrow & \text{Factor '+' Term} \mid \text{Factor} \\ \text{Factor} & \rightarrow & \text{Number '}*\text{' Factor} \mid \text{Number} \\ \text{Number} & \rightarrow & 0 \mid \cdots \mid 9 \end{array}$$

In this case the transformation does not change the accepting language of the grammar. However, the parser is still unable to correctly decide whether to choose the first or second alternative for the $Term$ production. The second step is thus to rewrite the grammar using *left factoring*:

$$\begin{array}{lcl} \text{Term} & \rightarrow & \text{Factor TermSuffix} \\ \text{TermSuffix} & \rightarrow & \text{'+' Term} \mid \epsilon \\ \text{Factor} & \rightarrow & \text{Number FactorSuffix} \\ \text{FactorSuffix} & \rightarrow & \text{'}*\text{' Factor} \mid \epsilon \\ \text{Number} & \rightarrow & 0 \mid \cdots \mid 9 \end{array}$$

Left factoring means that a common prefix of a production is factored out. In this case the nonterminal $Factor$ is a common prefix to both alternatives of the $Term$ production. The two alternatives could then be combined by adding one new production with an empty alternative. These transformations made the grammar parseable with a predictive parser that uses one token of lookahead.

Predictive parsers can only parse LL(k) grammars, where k is the amount of lookahead required to make a successful decision regarding which production to evaluate. If a predictive parser is used on such a grammar the parser runs in linear time proportional to the length of the input string [1, 10].

2.4.3 Left Recursion

One problem that conventional top-down parsers face is that they are unable to handle *left recursion*. A left-recursive production is of the form:

$$A \rightarrow A\alpha|\beta$$

For a recursive descent parser, the corresponding function call for A would look something like this:

```
function A()
{
  A();
  ...
}
```

This would lead to an infinite recursion and the parser would be unable to terminate. One approach to handle this problem will be described in Section 3.2.3.

2.5 Abstract Syntax Tree

The created parse tree will often times contain an unnecessary large amount of nodes. For instance, the grammar defined in Section 2.3 could be extended to support parenthesis in the following manner:

Term	$\rightarrow$	Term '+' Factor | Factor
Factor	$\rightarrow$	Factor '$*$' Primary | Primary
Primary	$\rightarrow$	'(' Term ')' | Number
Number	$\rightarrow$	0 | $\cdots$ | 9

Figure 2.3 illustrates how the parse tree of parsing the input string `(9+5)*2` would look like. However, some of this information is redundant. The parenthesis-nodes can be removed without altering the precedence of the expression. The precedence from the parenthesis comes implicitly from the structure of the graph. Removing such irrelevant information from the parse tree results in creating an *abstract syntax tree* (AST), which is a more compact representation of the input string.

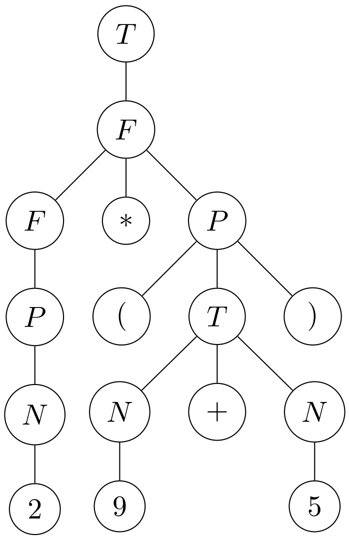

Figure 2.3: A parse tree of parsing the string `(9+5)*2`. For brevity, $Term$, $Factor$ and $Number$ are expressed as T, F and N respectively.

Chapter 3

Packrat Parsing

Packrat parsing is a top-down, recursive descent parsing technique that provides the user with unlimited lookahead and backtracking, and yet guarantees linear parse time [10, 12, 13]. Packrat parsing uses the backtracking approach as described in Section 2.4.1 but without suffering from a potential exponential parsing time. In addition, packrat parsing is a recursive descent parsing technique which makes it relatively straightforward to implement by hand.

3.1 Founding Work

The founding work for packrat parsing was done in 1970 by A. Birman et. al. [5]. Birman introduced a schema called the *TMG recognition schema* (TS). Birman's work was later refined by A. Aho and J. Ullman et. al. [2], and renamed into *generalized top-down parsing language* (GTDPL). This was the first top-down parsing algorithm that was deterministic and used backtracking. Due to the resulting grammar being deterministic they discovered that the parsing results could be saved in a table to avoid redundant computations. However, this approach was never put into practice, most likely due to the limited amount of main memory in computers at that time [10, 33]. Another characteristic of GTDPL is that it can express any LL(k) and LR(k) language, and even some non-context-free languages [2, 4].

3.2 Parsing Expression Grammars

As an extension to GTDPL and TS, Bryan Ford introduced PEGs [13]. CFGs (which were introduced mainly for usage with natural language [13]) may be ambiguous and thereby either (1) produce multiple parse tree's, which is not necessary due to only one is needed, and (2) produce a heuristically chosen one, which might not even be correct [15]. However, one of the characteristics of PEGs is that they are by definition *unambiguous* and thereby provides a good match for machine-oriented languages (since programming languages are generally designed to be deterministic). It is also shown that PEGs, similar to GTDPL and TS, can express all LL(k) and LR(k) languages, and that they can be parsed in linear time with the memoization technique [13].

3.2.1 Definition And Operators

PEGs, as defined in [13], are a set of productions of the form $A \leftarrow e$ where A is a nonterminal and e is a parsing expression. The parsing expressions denote how a string can be parsed. By matching a parsing expression e with a string s, e indicates success or failure. In case of a success, the matching part of s is consumed. If a failure is returned, s is matched against the next parsing expression. Together, all productions form the accepting language of the grammar.

The following operators are available in PEG productions: [13, 33]

- **Ordered choice:** $e_1/\dots/e_n$, apply expression $e_1,\dots,e_n$ in this order, to the text ahead, until one of them succeeds and possibly consumes some text. Indicate success if one of the expression succeeded. Otherwise do not consume any text and indicate failure.
- **Sequence:** $e_1,\dots,e_n$, apply expressions $e_1,\dots,e_n$, in this order, to consume consecutive portions of the text ahead, as long as they succeed. Indicate success if all succeeded. Otherwise do not consume any text and indicate failure.
- **And predicate:** $\&e$, indicate success if expression e matches the text ahead; otherwise indicate failure. Do not consume any text.
- **Not predicate:** $!e$, indicate failure if expression e matches the text ahead; otherwise indicate success. Do not consume any text.
- **One or more:** e^+, apply expression e repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any) and indicate success if there is at least one match. Otherwise indicate failure.
- **Zero or more:** e^*, apply expression e repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any). Always indicate success.
- **Zero or one:** $e?$, if expression e matches the text ahead, consume it. Always indicate success.
- **Character class:** $[s]$, if the character ahead appears in the sting s, consume it and indicate success. Otherwise indicate failure.
- **Character range:** $[c_1 - c_2]$, if the character ahead is one from the range c_1 through c_2, consume it and indicate success. Otherwise indicate failure.
- **String:** 's', if the text ahead is the string s, consume it and indicate success. Otherwise indicate failure.
- **Any character:** . (dot), if there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

The repetition operators (* and $^+$) does not have linear complexity, however, and should therefore be used with caution with packrat parsers to ensure linear parsing time, see Section 9.1.5.

3.2.2 Ambiguity

The unambiguousness of a PEG comes from the *ordered* choice property. The choices in a CFGs are *symmetric*, i.e., the choices need not be checked in any specific order. However, the choices in a PEGs are *asymmetric*, i.e., the ordering of the choices determine in which order they are tested. For a PEG the first expression that matches are always chosen. This means that a production such as $A \leftarrow a/aa$ is perfectly valid and unambiguous. However, it only accepts the language $\{a\}$. On the contrary, a CFG production $A \rightarrow a|aa$ is ambiguous but accepts the language $\{a, aa\}$.

A traditional example that is hard to express with the use of CFGs is the *dangling else* problem. Consider the following if-statement:

```
    if cond then if cond then statement else statement
```

This statement can be matched in the following two ways:

```
    if cond then (if cond then statement else statement)
    if cond then (if cond then statement) else statement
```

If the intended matching is the former of the two (in fact, this is how it is done in the programming language C [8]) then the following PEG production is sufficient:

Stmt	$\leftarrow$	'if' Cond 'then' Stmt 'else' Stmt
	/	'if' Cond 'then' Stmt
	/	...

Note: Matching the outermost if with the else-clause is believed *not* to be possible with a PEG. However, no source to either prove or contradict this statement was found.

Discovering if a CFG production is ambiguous is sometimes a non-trivial task. Similarly, choosing the ordering of two expressions in a PEG production without affecting the accepting language is not always straightforward [13].

3.2.3 Left Recursion

As discussed in Section 2.4.3 left recursion is when a grammar production refers to itself as its left-most element, either *directly* or *indirectly.* Similar to the conventional LL(k) parsing methods, left recursion proves to be an issue for PEGs, and therefore a problem also for packrat parsing [10, 13]. Consider the following alteration of the production described in Section 3.2.2:

A $\leftarrow$ Aa / a

For a CFG, this modification is not a problem. However, for a PEG, parsing of nonterminal A requires testing that A matches, which requires testing that A matches etc, producing an infinite recursion. However, it was early discovered that a left-recursive production can always be rewritten into an equivalent right-recursive production [1], and thus making it manageable for a packrat parser. However, if there is *indirect* left recursion involved, the rewriting process may become fairly complex.

3.2.4 Syntactic Predicates

As shown in Section 3.2.1 PEGs allow the use of the *syntactic predicates* ! and &. Consider the following grammar production:

A $\leftarrow$!B C

Every time this production is invoked it needs to establish if the input string matches a B and if it does, signal a failure. If there is no match the original input string is compared with the nonterminal C. This ability to "look ahead" an arbitrary amount of characters combined with the selective backtracking (described in Section 3.3) gives packrat parsers unlimited lookahead [10, 12, 13, 31].

3.3 Memoization

The introduction of memoization was done as a machine learning method in 1968 by D. Michie et. al. [23]. By storing calculated results, the machine "learned" it. The next time it was asked for the same result, the machine merely "remembered" it by looking up the previously stored result. The storage mechanism used was a stack. This makes the look-up process become linear. However, insertions of results are constant, they are merely pushed on top of the stack.

In packrat parsing, the results are instead stored in a matrix or similar data structure that provides constant time look-ups (when the location of the result is already known) and insertions [10]. For every encountered production this matrix is consulted; if the production has already occurred once the result is thereby already in the matrix and merely needs to be returned; if not, the production is evaluated and the result is both inserted into the matrix and returned.

Conventional recursive descent parsers that use backtracking may experience exponential parsing time in the worst case. This is due to redundant calculations of previously computed results caused by backtracking. However, memoization avoids this problem due to the fact that the result only needs to be evaluated once. This gives packrat parsing a linear parsing time in relation to the length of the input string (given that the access and insertion operations in the matrix are done in constant time).

Consider the following trivial PEG, taken from [10]:

Additive	←	Multitive '+' Additive / Multitive
Multitive	←	Primary '*' Multitive / Primary
Primary	←	'(' Additive ')' / Decimal
Decimal	←	[0-9]

With this grammar and the input string `2*(3+4)` the following memoization matrix can be produced:

`pAdditive`	7	-1	5	3	-1	1	-1	-1
`pMultitive`	7	-1	5	1	-1	1	-1	-1
`pPrimary`	1	-1	5	1	-1	1	-1	-1
`pDecimal`	1	-1	-1	1	-1	1	-1	-1
input	'2'	'*'	'('	'3'	'+'	'4'	')'	$

Table 3.1: A matrix containing the parsing results of the input string `2*(3+4)`

The columns correspond to each position of the input string, the rows correspond to each of the parsing procedures. To make it clear that the rows are in fact procedures they have been given a prefix '`p`'. Each cell contains either a number that represents how much of the input string that have been consumed by a previous call to the procedure, or the cell contains a '-1' which indicates a failed evaluation. For instance, if backtracking occurs at input position four (where the number '3' is present) and procedure `pAdditive` is called, the parser first checks if a previous result is stored in the matrix. In this case the number 3 is stored and thereby the parser immediately knows that it can advance three steps on the input string and end up at the seventh character of the input string. If the stored value is '-1' the parser knows that a previous call resulted in a failed parse and can thus avoid continuing with the procedure call and instead return a failure response to the calling function.

This illustrates how redundant computations and thereby also potential exponential parsing times are avoided with the assistance of memoization.

Calculating the whole matrix in Table 3.1 would be unneccesary since many of the cells are not needed. The idea behind packrat parsing is not to evaluate *all* of the cells in the parsing matrix, only the results that are needed [10]. This effectively reduces the amount of memory space required during parsing.

3.4 Scannerless

Conventional parsing methods are usually split into two phases: the lexical analysis phase and the parsing phase. The tokenization phase is called lexer, lexical analyser, tokenizer or scanner. The lexical analysis splits the input string into tokens which hopefully corresponds to the permitted terminals of the grammar. This lexical analysis is important for conventional parsers due to their inability to refer to nonterminals for lookahead decisions [12]. Thus, the parser treats the tokens acquired from the lexical analysis as if they were terminals.

Packrat parsers can on the other hand be *scannerless*, which means that it requires no lexical analysis. When a scannerless packrat parser evaluates different alternatives, it can rely on already evaluated results. This effectively makes a packrat parser able to use both terminals and nonterminals during lookahead [10, 12].

Performing Lexical Analysis

Large parts of the code base of programs may consist of white spaces, comments and other irrelevant information that is not needed for the semantic analysis. A lexical analyser can effectively disregard such information by simply opting not to create any tokens for them, thus no specific productions for white spaces and/or comments need to be included in the grammar specification of the parser.

For a packrat parser that does not use a lexical analyser, however, this is not the case. A packrat parser that uses no lexical analyser, the white spaces and comments need to be incorporated into the productions of the grammar.

As previously mentioned, the conventional parsers treat the created tokens given by the lexical analyser as if they were terminals, and between each token the lexical analyser disregards any white spaces or comments. To achieve this effect for a packrat parser, a production to manage white space or comments can be created and used after each terminal symbol of the grammar. For instance, the grammar specified in Section 3.3 can be altered in the following way to be able to correctly handle white spaces inside an arithmetic expression:

```
Additive    ←   Multitive ’+’ Spaces Additive / Multitive
Multitive   ←   Primary ’*’ Spaces Multitive / Primary
Primary     ←   ’(’ Spaces Additive ’)’ Spaces / Decimal
Decimal     ←   [0-9] Spaces
Spaces      ←   (’ ’ / ’ \t’ / ’ \n’ / ’ \r’)*
```

This grammar is now able to parse an arithmetic expression such as: `2 * (3+ 4 )`.

3.5 Parse Completion

The grammar from Section 3.3 would successfully parse the input string `2*(3+4)abc`. The `2*(3+4)` part of the string would be parsed as an *Additive* and the rest of the input would be unconsumed. Thus, the parser would falsely report success without even considering the `abc` part of the input string.

This behaviour can be avoided with two different approaches: One approach is to add the special symbol '.' from Section 3.2.1, the *any character*, combined with a new starting symbol of the grammar. Given this, the grammar from Section 3.3 can be transformed into the following grammar:

Start	$\leftarrow$	Additive !.
Additive	$\leftarrow$	...
Multitive	$\leftarrow$	...
Primary	$\leftarrow$	...
Decimal	$\leftarrow$	...

The use of the *not-predicate* (!) ensures that there are no trailing characters after an *Additive*. This means that the parser only reports success when the whole input string has been consumed.

Another solution to this problem is to simply compare the amount of consumed characters with the total size of the input string. This comparison can be done when the parser believes it has parsed the whole expression.

3.6 Practical Limitations

3.6.1 Memory Consumption

One of the drawbacks with using packrat parsing is the additional memory consumption when compared with conventional parsing techniques. A tabular approach where the whole $m \times n$ matrix is evaluated would require $\Theta(mn)$ space, where m is the amount of nonterminals, i.e., productions of the grammar, and n is the size of the input string. However, for a packrat parser, only the required cells in the matrix are evaluated, and these cells are directly related to the input string and not the nonterminals of the grammar [12]. In other words, adding more productions to the grammar may not necessarily increase the memory consumption whilst increasing the size of the input string will always increase the memory consumption. This makes the required size of the memoization matrix for a packrat parser be proportional only to the size of the input string, thus $\mathcal{O}(n)$.

Even if the space consumption is upper-bounded by the input string and can therefore be written as $\mathcal{O}(n)$ there is a "hidden constant multiple" of n [12, 24]. This is because there can be more than n elements in the produced memoization matrix.

Conventional LL(k) and LR(k) parsing algorithms only require storage space proportional to the *maximum recursion depth* that occurs for the given input. This causes these conventional algorithms to have the same worst case memory requirement as a packrat parser. However, a packrat parser is also lower bounded by n and this worst case behaviour for LL(k) and LR(k) parsers rarely occurs [10, 12]. In fact, the maximum recursion depth is usually significantly smaller than the size of the input string [12].

3.6.2 Maintaining States

A parser for the programming languages C and C++ requires that the parser is able to maintain a global state. This is due to the nature of typedef's for the two languages. The parser needs to be able to distinguish whether the input is a typedef symbol, an object, a function or an enum constant, and alter the global state accordingly if the meaning of a specific token changes. For instance, the following C code requires this feature: [14]

```
T(*b)[4]
```

By only looking at this snippet, the parser has no way of knowing whether `T` refers to a function call or a typedef name, it is *context-sensitive*. If it is a function call, the snippet corresponds to accessing the fifth element of the resulting call to function `T` with the pointer `b` as input parameter. If `T` instead is a typedef, the snippet corresponds to `b` pointing to an array consisting of four elements of type `T`.

C or C++, however, can still be parsed with a packrat parser that changes its state whenever a variable changes its type [15]. This is because of the requirement that the type of a variable needs to be declared *before* its usage and therefore no parsing information prior to a definition of a variable is lost. This way, a separate symbol table can be constructed during the parse which keeps track of the type for different tokens.

However, for the general case of context-sensitive grammars, packrat parsers may experience exponential parsing time *and* memory consumption. This is because during parsing a packrat parser assumes that an already evaluated cell of the result matrix is the correct result, and that this value will not have to change. But if a state change occurs the result matrix may have to be re-evaluated to ensure a correct result during backtracking. This can potentially break the guaranteed linear time characteristic due to cells being evaluated multiple times [12].

Chapter 4

Lex/Yacc

The Lex/Yacc parsing combination has been around since the 1970's [19, 20]. They were developed as separate tools but they were designed to easily integrate with each other [20]. Both Lex and Yacc are written in the programming language C and Yacc can produce an LALR(1) parser in either C or C++. An LALR(k) parser uses lookahead and works similarly to a predictive LL(k) parser. An LALR(k) parser translates the grammar into a deterministic finite state machine [1]. The transitions between the different states are determined by the *action* whether to *shift* or *reduce* which is the same actions as for a shift-reduce parser as was described in Section 2.3.1. LALR(k) grammars are a simplification of LR(k) grammars but decreases the amount of states in the finite state automaton in order to decrease the memory footprint. However, this decreases the recognition power for LALR(k) to be *strictly smaller* than LR(k) [9].

The Lex/Yacc combination has been replaced by newer versions continuously since their initial release. In the late 1980's the two tools Flex and Bison were created which was introduced as direct replacements to Lex and Yacc respectively [21]. Flex/Bison introduced several improvements and to ease any potential transitions they are both backwards-compatible with Lex/Yacc.

4.1 Lex

Lex is a lexical analyser which receives an input stream that it transforms into an output stream. The output stream consists of the tokens that are sent to the parser. The tokens are created with the use of a deterministic state automaton [20].

A Lex specification consists of three different parts: *definitions*, *rules* and *user subroutines.* The definition part is optional and is mainly used for including external source files, defining global variables, token types etc. The rules section is where the actual lexical analysis is defined. The Lex rules are defined as regular expression and support several operators such as repetitions, ranges, optionals and complements. The user subroutines section is optional and can contain C/C++ functions that will be pasted into the source code as is of the lexical analyser.

There can be an associated action to a rule which is executed each time the rule matches. This action needs to be specified in C or C++ depending on which of the two languages the lexical analyser will be generated in. For instance, if Lex is to be used together with Yacc, each rule needs to have a corresponding action which returns a value for the variable *yylval* which is a pre-defined variable that is used by Yacc.

The produced lexical analyser will create a function called *yylex()* which when called upon triggers the associated action for the next token of the input stream.

4.1.1 Ambiguity

Lex is able to tokenize an input stream even though its rules can be ambiguous. Consider the following two Lex rules:

```
lex      action1;
[a-z]*   action2;
```

If the current input is `'lex'` both of these rules will match, thus Lex needs to choose which of the two actions that will be executed. This is done with the following two disambiguation definitions: [20]

1. Choose the longest match.
2. If the match is for the same amount of characters, choose the first rule (top-most in the source file).

This particular case will trigger the second disambiguation definition since they both have the same length, thus the first (top-most) lexical rule is chosen and *action1* is executed. If the ordering of the two lexical rules were switched the `lex` rule would never match.

4.2 Yacc

Yacc is a parser generator written in C which produces a LALR(1) parser in C or C++ code, thus Yacc only accepts grammars that are LALR(1) [19]. Similarly to Lex, a Yacc specification consists of three different sections: *declarations*, *rules* and *programs*.

The declaration section is optional and can contain raw C/C++ code such as includes, defines and declaration of variables. All variables that are instantiated in the declarations section will be defined in a global scope and can therefore be accessed throughout the whole parser specification. The declaration part can also include the names of the tokens and the type that is returned from each rule, which is usually needed for the creation of the parse tree.

The rules section is where the productions of the grammar is declared. As in Lex, there can be an associated action to each rule specification. These action needs to be enclosed in angle brackets and are specified as C/C++ code.

The program part of the specification file is optional and can be used to declare functions. For instance, if the user does not wish to use Lex, a user-specified lexical analyser can be inserted into the program section.

A compiled Yacc program creates a function called *yyparse()* which repeatedly calls *yylex()* to receive input tokens. Fittingly, as mentioned in Section 4.1, a Lex program provides the *yylex()* function automatically.

4.2.1 Ambiguity

As mentioned in Section 2.3.1 a shift-reduce parser can experience shift-reduce conflicts when the choice of choosing whether to shift or reduce is ambiguous. There also exists a case called *reduce-reduce* conflicts which is similar but involves two different reduce actions. When either of those two conflicts are found, Yacc consults its two disambiguation rules: [19]

- For a shift-reduce conflict, the default action is to choose shift.
- For a reduce-reduce conflict, the default action it to reduce with the earliest defined grammar production (top-most in the source file).

4.2.2 Left recursion

As described in Section 2.4.3, left recursion for a top-down parser makes the parser enter an infinite loop. In contrast, Yacc *encourages* the user to write the productions left-recursively [19]. This is due to shift-reduce parsers use of a stack. If a production is left-recursive the option to reduce will become available sooner, thus the amount of elements on the stack will be smaller. If a production is right-recursive; the reduce action will be postponed until it has been reached through shifting and therefore the stack will need more space, thus increasing the overall memory consumption.

Chapter 5

Implementation

Implementing a parser generator can be a non-trivial task and may require several thousand lines of code. Therefore, to decrease the amount of time required for the implementation part of this thesis the choice was to alter an already existing parser generator instead of implementing one from scratch.

There exists several packrat parser generators, many of them listed at [11]. IAR Systems mostly use the programming languages C++ and Ruby in production, thus the choice of which packrat parser generator to choose was mainly based on the programming language criteria. Ruby was also seen as the favourable choice for the implementation of the parser generator for its high-level characteristic Due to its high-level characteristics, however, programs implemented in Ruby are most of the times slower when compared with C++ programs. For this reason the choice of using C++ as the language used by the actual generated parser came as the natural choice. The choice of ”re-implementing” the parser generation instead of taking an already existing packrat parser generator that generates C++ parsers and improve it was because IAR Systems wanted to gain a better understanding on how packrat parsers work, and thus decrease the time it would take if they choose to implement their own packrat parser generator in the future.

With these criteria in mind, the chosen packrat parser generator to alter is an open-source implementation called Treetop [18]. Treetop is a packrat parser generator written in the programming language Ruby which produces packrat parsers in Ruby. The implementation part of this thesis consisted of altering Treetop in such a way that it instead produces packrat parsers in C++, this altered version of Treetop will be referred to as Hilltop. To illustrate how a parser generated by Treetop and Hilltop may look like, see Appendix B and C.

5.1 Treetop

The accepting grammar for Treetop is based on PEGs and its grammar specification consists of roughly 500 lines of code and it supports all operators specified in Section 3.2.1. Each of these operators and some additional ones are specified in separate Ruby modules. Each module contains a `compile` method which is the method that performs the actual code generation.

Code Generation For a Nonterminal

An example from the file `nonterminal.rb` which generates code for a nonterminal:

Listing 5.1: Code generation for a nonterminal in Treetop

```ruby
class Nonterminal < AtomicExpression
  def compile(address, builder, parent_expression = nil)
    super
    use_vars :result
    assign_result "_nt_#{text_value}"
    extend_result_with_declared_module
    extend_result_with_inline_module
  end
end
```

For readers unfamiliar with the Ruby programming language, the author refers to the official Ruby documentation [36].

One important part of this snippet is the `builder` object. The `builder` object contains the string buffer that will be pasted into the source file, i.e., the string is the code that is to be generated.

`use_vars` is a method specifying which set of objects (everything is an object in Ruby, even integers, strings etc.) that is to be used and generated by the specific module. The available objects are `result`, `index` and `accumulator` (in the generated code they correspond to the `r(index)`, `i(index)` and `s(index)` respectively). To differentiate between different `result`, `index` and `accumulator` objects throughout the code, the objects are assigned specific addresses. The `result` and `accumulator` objects are of type `SyntaxNode` which is the type of the parse tree. The `index` object is an integer.

The `assign_result` method generates code which correspond to assigning the current result variable the object passed to the method, e.g., `assign_result 2` $\rightarrow$ `r(index) = 2`. In this particular case, the passed value is the identifier of the nonterminal (`text_value`) with the string `_nt_` in front of it.

The `extend` methods append a resulting variable to the current accumulator. Thus, the bottom-most accumulator variable (`s0`) will contain the parse tree of the result of each method. This result will both be returned by the method and inserted into the memoization matrix.

The snippet in Listing 5.1 may produce the following code inside the parser if a nonterminal named `op` is found:

```
r4 = _nt_op
s3 << r4
```

5.2 Hilltop

Hilltop uses the same structure as Treetop for the produced code. However, due to the differences between C++ and Ruby, several alterations needed to be done:

- Treetop uses *instance variables* for the index, memoization matrix and the input string objects. Instance variables are accessible within the class they are created (the parser in this case).

- Each method of a Treetop packrat parser returns a node for the syntax tree (or `nil` to signal failure).

Hilltop uses a different approach. To avoid having to use global variables or create a separate class containing the index variable, the memoization matrix and the input string; the memoization matrix is passed as a pointer to each function alongside the input string and the current value of the index. In addition, instead of having the methods return a tree node as in Treetop; Hilltop's functions return the new index (or -1 for failures). This means that the `result` variables will be integers and not tree nodes. To be able to alter the parse tree it is also passed as a pointer between the function calls.

So, contrary to the Treetop version which passes no arguments to its method calls, a basic call to the `Ht_op` function in Hilltop will have the following structure:

```
int Ht_op(const std::string input, int index, Ast::opNode *tree,
   MemoTable *memo_table);
```

Code Generation For a Nonterminal

The Hilltop version of the file `nonterminal.rb` looks as follows:

Listing 5.2: Code generation for a nonterminal in Hilltop

```
class Nonterminal < AtomicExpression
  def compile(address, builder, parent_expression = nil)
    super
    use_vars :result, :start_index
    obtain_new_non_term_address
    builder << "Ast::#{text_value}Node␣*#{accumulator_var}␣=␣new␣
       Ast::#{text_value}Node();"
    assign_result "Ht_#{text_value}(input,␣index,␣#{
       accumulator_var},␣memo_table)"
    builder << "#{builder.get_curr_class_ass}->#{text_value}#{
       non_term_address}␣=␣#{accumulator_var};"
  end
end
```

Hilltop's version might look a bit more complicated but basically this is only due to the static typing of C++, the `accumulator` variable need to be assigned the correct type before it can be used. More details regarding the structure of the parse tree will be discussed in Section 5.2.1.

The snippet in Listing 5.2 may produce the following code inside the parser if a nonterminal named `op` is found:

```
int r4 = -1;
const int i4 = index; // Used for backtracking
Ast::opNode *s4 = new Ast::opNode();
r4 = Ht_op(input, index, s4, memo_table);
tree->op0 = s4;
```

5.2.1 Parse Tree

The parse tree for Hilltop consists of a common base class named `AST` which is defined as follows:

```
class AST {
  public:
  virtual ~AST() {}
  virtual void print(int indent) = 0;
  virtual std::string getVal() = 0;
  virtual void updateVal(std::string acc) = 0;
  virtual void setIndex(int idx) = 0;
  virtual int getIndex() = 0;
};
```

All other nodes are derived classes which inherits the functions from the base class `AST`. For a production $A \leftarrow B/C$ one class is created for the specific production, namely A and is named `ANode`. Each of A's choices is created as children to `ANode`:

```
class ANode : public AST {
  std::string val;
  int index;
  public:
  ANode0 *A0;
  ANode1 *A1;
  // Constructor and methods for ANode
};
```

Each choice then creates a child node for each of its nonterminals:

```
class ANode0 : public AST {
  std::string val;
  int index;
  public:
  BNode *B0;
  // Constructor and methods for ANode0
};
class ANode1 : public AST {
  std::string val;
  int index;
  public:
  CNode *C0;
  // Constructor and methods for ANode1
};
```

All nodes also contain a string variable named `val` and an integer variable called `index`. `val` holds the result from potential evaluations of its production, i.e., the string values of potential terminals. `index` contains the index information that is used for the memoization matrix.

Repetitions

The classes created from productions containing repetition suffixes (* or $^+$) will need to be able to store an arbitrary amount of evaluations caused by the repetition. Consider the following production: $A \leftarrow Q(BC)^*$. This production means that the class `ANode0` that will be created will need to store a `QNode` followed by any amount of pairs of `BNode`'s and `CNode`'s. To achieve this, whenever a repetition suffix is found, a *nested* class is created that contains the expression surrounded by the repetition suffix. The parent class then creates an `std::vector` that can contain an arbitrary amount of objects of the nested class.

The following snippet illustrates how `ANode0` would be generated by the above production:

```
class ANode0 : public AST
{
  std::string val;
  int index;
  public:
  QNode *Q0;
  class nested_rep_class0 : public AST
  {
    std::string val;
    int index;
    public:
    BNode *B1;
    CNode *C2;
    // Constructor and methods for the nested class
  };
  std::vector< nested_rep_class0 > repetition0;
  // Constructor and methods for ANode0
};
```

For a more complete example of how the parse tree is generated, see Appendix D.

5.2.2 Memoization

Array. A number of different data structures were considered for the memoization matrix. One of them was a two-dimensional array. Array access and insertion operations are done in constant time which would maintain the guarantee of linear parse time for the packrat parser. However, using a two dimensional array for the memoization matrix would mean that the size needs to be allocated statically and thus require allocation of an $m \times n$ matrix, where m is the amount of productions of the grammar and n is the length of the input string. This is inefficient since most cells in the matrix will not be used and therefore most of the memory allocation will be wasted, as described in Section 3.3.

Map. Another data structure that was considered is called `map` [27]. How this data structure is defined is implementation-specific but it is usually implemented as a a binary search tree.

map provides the user with dynamic allocation which is more suited for this type of needed properties. Another beneficial feature of the map data structure is that it allows the indices to be of different types. This allows for converting the function names into strings and using them as row indices. To maintain the invariant of a binary search tree, the map data structure sorts its elements whenever an insertion occurs. This makes the insertion operation to be logarithmic, thus breaking the linear time guaranteed parse time performance if used in a packrat parser.

Unordered map. The chosen data structure for the packrat parser was unordered_map [28]. unordered_map is a hash map that inserts its elements into *buckets*. Which bucket that the inserted elements are to be placed in is based on a *hash function*. The insertion and lookup operations are done in constant time on the average case. However, if there is a hash collision or if the currently allowed bucket count is exceeded there will be a *rehashing* of the entire hash map. This rehashing procedure is done in linear time in relation to the amount of elements currently inside the data structure and should therefore be avoided as much as possible to ensure the linear time performance for the packrat parser. By increasing the amount of buckets the possibility of a hash collision effectively decreases. During the experimental evaluation it was found that the needed amount of buckets never exceeded the size of the input string. The hash map was thus statically allocated an amount of buckets equal to the size of the input string to avoid any rehashing.

To make the unordered_map a two-dimensional data structure similar to the one described in Section 3.1 the typedef for the memoization matrix looks as follows:

```
typedef std::unordered_map<std::string,Ast::AST*> InnerMemoTable;
typedef std::unordered_map<unsigned int,InnerMemoTable> MemoTable
    ;
```

Instead of storing an index in the memoization matrix as described in Section 3.3, an AST node is stored. Every AST node then has an integer variable named index which serves this purpose.

Inserting elements into this data structure is similar to how insertion would look like for a regular two dimensional array:

```
memo_table[row_index][column_index] = tree_node;
```

Since the allocation is dynamic, care needs to be taken when checking if an element already exists in the memoization matrix. The access operator [] will create an element if no current element exists in the corresponding location. Therefore, when checking for previous results in the memoization matrix, the access operator [] should not be used. Instead, the pre-defined function find() is preferable since it does not create any previously non-existing elements.

5.3 Implemented Improvements

5.3.1 Transient Productions

To reduce the amount of memory required by the memoization matrix, a prefix notation (∼) called *transient* was implemented. The idea behind this prefix is based on R. Grimms transient optimization [15]. If a nonterminal is flagged with this prefix the parser generator will not generate any code responsible for memoizing the result, and there will be no nodes created for the marked production on the parse tree.

Productions that can be well suited for the transient prefix notation is productions responsible for recognizing white spaces and comments, especially since such productions only populate the parse tree with a lot of unwanted information. To be able to mark a production with the ∼ prefix there needs to be assurance that no backtracking (or at least a very limited amount) for the marked production will be needed. One instance where the prefix can be used on white spaces and comments is for productions recognizing lexical syntax:

$$\text{LEFTPAREN} \quad \leftarrow \quad \text{'('} \sim\text{SPACE}$$

This production consumes a left parenthesis followed by any amount of white spaces. For this particular production the *SPACE* production does not need to be memoized. This is since knowing the amount of white spaces is irrelevant. The useful information is knowing the *total* amount of input characters that is consumed. However, great care needs to be taken when a production is marked as transient since a result from a transient production is not memoized and can thus violate the linear parse time guarantee for the packrat parser.

The insertion of the transient prefix is currently done manually in the grammar. The effect of this optimization will be shown in Chapter 6.

5.3.2 Hashing

During the analysis it was discovered that the hash function for strings in C++ was slower than for hashing integers. The function names were then instead defined as `enum`'s to allow them to be used as integers. This improved parsing times by roughly 15%. This also suggests that using a hash table as the data structure for the memoization matrix might not be an optimal solution due to the added complexity of hash functions and also due to the issue regarding rehashing.

5.4 General Comparison Between Hilltop and Lex/Yacc

5.4.1 LALR(1) versus LL(k)

Packrat parser generators produce recursive descent parsers that accepts any LL(k) grammar and the Lex/Yacc combination creates a LALR(1) parser that accepts LALR(1) grammars. It has been shown that *all* LL(k) grammars can be rewritten into an equivalent LR(k) grammar, where equivalent means that they accept the same language [32]. As mentioned in the first section of Chapter 4, LALR(k) grammars are a subset of LR(k) grammars. Thus, the recognition power of Lex/Yacc is strictly smaller than for a packrat parser.

5.4.2 Scannerless versus Separate Lexical Analyser and Parser

As described in Section 3.4 packrat parsers can be scannerless. In that case, in addition to the grammar productions, the grammar specification needs to include lexical analysis such as how to handle white spaces and comments. This approach has the beneficial advantage of having the same syntax for the lexical analysis and the parser.

A disadvantage with the separate lexical analyser and parser approach is that the tools will often be bound to work only with each other. This means that if a user only wants to use one of the tools the other tool will need to be manually implemented and correctly interfaced. In addition, if third-party tools are used and the lexical analyser and parser generator are developed independently of each other this can lead to version conflicts.

5.4.3 Modularity and Extendibility

PEGs are closed under composition, union and intersection [13] which makes them a good match for constructing modular grammars. This allows two modular PEGs to be joined and produce a proper union of the two grammars. This encourages grammar writers to group similar grammar productions in separate files which results in easier to read grammars. It also allows for reuse of the same modular grammar between different parsing languages.

Modularisation for Lex/Yacc is harder to achieve since a source file only containing grammar/lexical productions can not merely be included in another source file without having to manually tamper with, for example, the return type specifications. In other words, it is easier to have a single large source file for each required grammar.

In addition, extensions to a Yacc specification file may introduce conflicts that are hard to debug. An issue regarding the alteration of the production `GuardingStatement` by adding an additional `Expression` for a grammar designed to parse Java is raised in [6]:

```
GuardingStatement :  SYNCHRONIZED '(' Expression Expression ')' Statement;
```

This extension results in 29 shift/reduce conflicts and 26 reduce/reduce conflicts. The troublesome part is that the conflicts are originating from parser states that seems to have no relation to the altered production. This makes the debugging of the grammar unnecessarily hard.

A packrat parser would not be bothered with such a change due to being unambiguous by definition and is therefore considered better suited for grammar extensions [14].

5.4.4 Error Reporting

When using Lex/Yacc the user needs to implement the error-handling functions manually. Yacc defines a special nonteminal called `error` that can be used inside a rule declaration where an error "is expected". If there is in fact a parse error in such a rule Yacc continues to parse and ignores the next three tokens hoping that the error is overcome and regular parsing can continue [19]. This allows Yacc to continue to parse even if an error occurred. Yacc also provides a function called *yyerror()* that is called whenever an error is encountered. By default this function does nothing but the user can override it to for instance implement what debugging information that should be displayed or if a specific action should occur. This solution is quite general and although it provides the user with great flexibility regarding error handling it also puts a large responsibility on the user.

Currently, the only error reporting in Hilltop consists of reporting the index for the last successful production, which may not be that helpful. However, general error reporting for packrat parser generators have been successfully implemented [12, 15], and a general error handling implementation is therefore left for future work.

5.4.5 Conflicts

Lex/Yacc creates a large amount of global variables, functions and macros, such as *yylex()*, *yyparse()* and *yylval* . This can cause conflicts if a program is produced to parse multiple languages. Lex/Yacc, however, supports a method of redefining the *yy*-prefix into something else. This allows for using a single program to contain multiple parsers.

Hilltop, however, uses a different approach. A parser produced by Hilltop is encapsulated inside a `namespace` which is named based on the name of the grammar. This allows for several parsers to exist in the same program without introducing conflicts. This, however, has not been fully integrated into the existing solution of Hilltop and is left for future work.

5.4.6 Parse Tree Creation

Similarly as for the error reporting, Yacc does not provide any default parse tree creation. This also needs to be implemented manually by the user. The parse tree creation is then inserted in each rule's own action and the node type need be declared as the return type for the rule.

In Hilltop, the parse tree and its specification is created automatically. Due to the lack of current support for code insertion inside grammar productions there can be no associated actions on the parse tree to create an AST. However, Treetop has built-in support for such a feature with syntax similar to Yacc where the actions are enclosed in curly brackets. This should make it possible to implement a similar solution for Hilltop.

Chapter 6

Experimental Results

To establish a result that is as realistic as possible the parsers have been tested on two different real-world grammars and inputs currently used at IAR Systems. To make the Lex/Yacc parser be on more equal terms with the Hilltop parsers large parts have been removed from the provided Lex/Yacc specifications; such as the AST generation in the Yacc source file and symbol table generation in the Lex source file. For one of the grammars, a manually written lexical analyser is used instead of Lex.

The first grammar, referred to as Grammar 1, is a DSL that mainly contains definition and assignment operations. Grammar 1 was chosen mainly because the tool that uses the grammar is old and in need of restructuring, and thus the grammar also needs to be redefined. Instead of using Lex, IAR Systems uses a manually written lexical analyser in combination with Yacc. The task of implementing the handwritten lexical analyser and constructing the grammar specification for Yacc has been a non-trivial task. IAR Systems are therefore interested in if packrat parsing can prove to be a suitable switch in parsing technique when a redesign of the tool used for Grammar 1 occurs.

The second grammar, referred to as Grammar 2, is an assembly language where most lines correspond to a set of different instruction calls, such as PUSH, LOAD, ADD etc. The input file is thus parsed line by line independently of the result from previous lines. The only production that is allowed to be parsed over multiple lines is multi-line comments. Grammar 2 was chosen because IAR Systems have a large amount of similar assembly languages that they are using. However, as described in Section 5.4.3, it is hard to modularise the grammar specifications of Lex/Yacc and therefore the grammars needs to be specified from scratch for every assembly language. Packrat parsing could therefore prove to be a good alternative because of the modular properties of using PEGs.

6.1 Testing Environment

There are three different parsers generated by Hilltop that have been tested:

- Ht-trans: creates a parse tree combined with the use of memoization and with the *transient* optimization, explained in Section 5.3, used on white space and comment productions.
- Ht-memo: creates a parse tree and uses memoization but *without* the transient optimization.
- Ht-tree: uses the transient optimization but does *not* use memoization and thus only creates a parse tree (conventional recursive descent parsing with backtracking).

For the Lex/Yacc combination, two different parsers have been generated:

- Manual/Yacc: a manually implemented lexical analyser in combination with Yacc. The parser is produced in C code. This parser is used for Grammar 1.
- Lex/Yacc: both Lex and Yacc is used for generating this parser. The parser is produced in C++ code. This parser is used for Grammar 2.

To make the five parsers produce a fair result as possible a parse tree creation was manually implemented for the Lex/Yacc and Manual/Yacc parsers.

Grammar 1 contains 43 productions in the Yacc specification file and the hand-written lexical analyser contains several hundreds of lines of code to effectively tokenize the input. The Hilltop grammar specification for Grammar 1 contains 57 productions, the increased amount of productions compared to the Yacc specification is due to the integrated lexical analysis.

Grammar 2 contains 154 rules for the Yacc specification file. The Lex specification file consists of 362 rules to handle all available tokens. The Hilltop grammar specification contains a total of 170 productions.

The versions used for benchmarking Lex/Yacc were flex-2.5.37 and bison-2.7.1 respectively. The parsing time experiments were run under Windows 7 (64 bit) on an Intel Quad Core Q9505 at 2.83 GHz with 4 GB of RAM. The memory consumption analysis was made on Ubuntu 12.04 (64 bit) using VirtualBox 4.3.6.

6.2 Parser Generation

The time it took the Hilltop parser generator to create the Ht-memo parsers (which produces the largest parser of the three Hilltop parsers) was roughly 800 ms for Grammar 1 and 2300 ms for Grammar 2. The Ht-memo parser for Grammar 1 was of size 443 kB and for Grammar 2 the size was 1806 kB. For the Manual/Yacc parser the lexical analyser was handwritten and thus not generated, the Yacc parser took 150 ms to generate for Grammar 1. For Grammar 2, the Lex program was generated in 140 ms and the Yacc parser in 380 ms. The total size of the Manual/Yacc parser was 140 kB and the Lex/Yacc parser was 583 kB.

6.3 Parsing Time

The parsing time for the Hilltop parsers and the Lex/Yacc parser was measured using a `QueryPerformanceCounter` [26]. The `QueryPerformanceCounter()` function returns the current value of the performance counter. The performance counter utilizes an internal hardware counter and can thus not be synchronized with any external timers or time zones [25]. To measure the total time spent parsing, the value of the performance counter before the parsing began and when it finished was compared. The resolution of the performance counter is in microseconds.

The Manual/Yacc parser, which was produced as C code, was measured using the `timeval` [17] struct. The `timeval` struct consists of two different parts: seconds and microseconds. To use this struct for timings, the `gettimeofday()` [16] function can be used to get the amount of seconds and microseconds since the Epoch, which is 1970-01-01 00:00:00. Similarly to how the `QueryPerformanceCounter()` was used, the `gettimeofday()` can be used before and after the parsing finishes to get the total amount of time spent parsing.

The parsers were fed with twelve input files with various sizes for each grammar. Each file was parsed a total of ten times and the parsing time was averaged.

6.3.1 Grammar 1

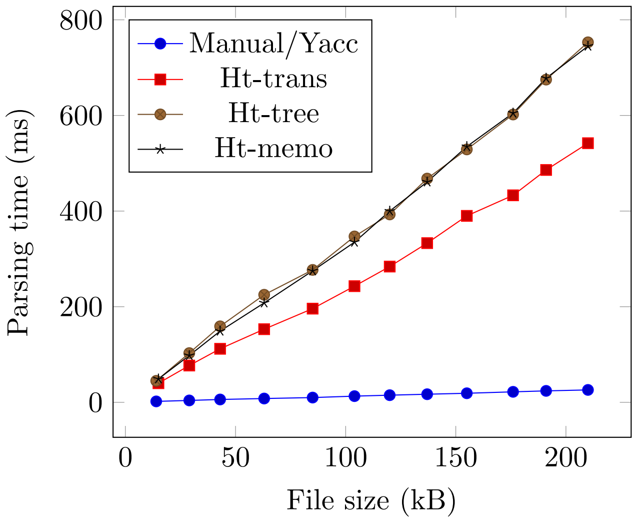

Figure 6.1: Parsing time for Grammar 1

Figure 6.1 illustrates the parsing time results of the four different parsers created for Grammar 1. It can be seen that the three Hilltop parsers all perform linearly, even Ht-tree which does not use memoization. The fastest of the three Hilltop parsers are Ht-trans. Ht-trans is fastest due to the high amount of backtracking performed during the parse. The largest input file produced a memoization matrix with 49227 elements and did 39810 lookups in the memoization matrix due to backtracking. Ht-tree thus suffered from having to recompute this high amount of redundant information and was roughly 30% slower compared to Ht-trans. Ht-memo is slower than Ht-trans due to the increased amount of memoized productions, the memoization matrix of Ht-memo consisted of 163021 elements. The Ht-memo and Ht-tree parser had roughly equal parsing time for the different input files. This suggests that for this particular grammar a trivial backtracking parser is as fast as a trivial packrat parser.

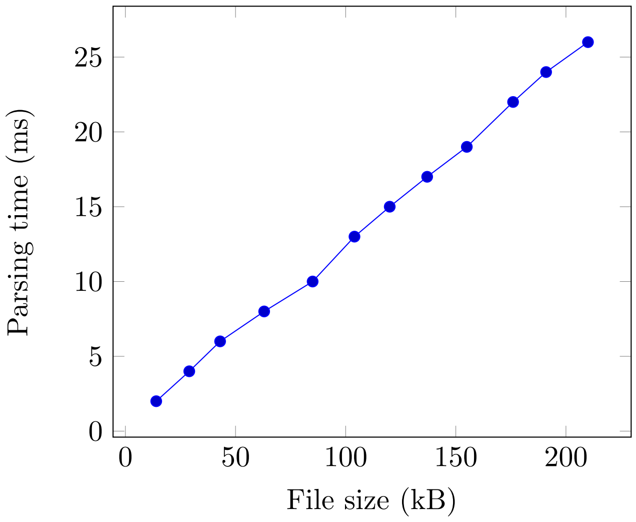

Figure 6.2: Manual/Yacc parse time for Grammar 1

As can be seen in Figure 6.1 the Manual/Yacc combination outperforms all three versions of Hilltop. Figure 6.2 is used to display that the parsing time grows linearly even for the Manual/Yacc combination.

6.3.2 Grammar 2

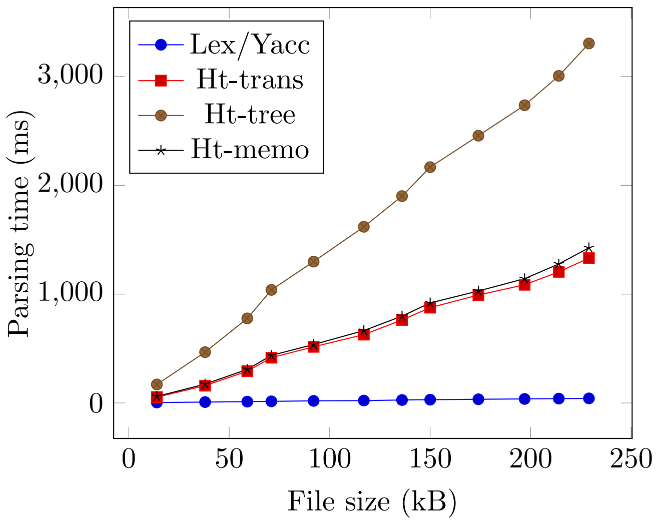

Figure 6.3: Parsing time for Grammar 2

Figure 6.3 illustrates the parsing time performance by the four different parsers produced for Grammar 2. As with Grammar 1, Ht-trans clearly outperforms Ht-tree. For this grammar and input files, there is an even heavier amount of backtracking occurring in relation to Grammar 1. For Ht-trans, the largest input file constructed a memoization matrix consisting of 85187 elements but was able to use the memoization matrix on 182730 backtracks. The Ht-memo parser comes relatively close to Ht-trans for Grammar 2, Ht-trans is only 7% faster. This is because the input files for Grammar 2 consists of less white spaces and comments in relation to the input files of Grammar 1, thereby the gain of the transient productions is not as significant for Ht-trans in Grammar 2.

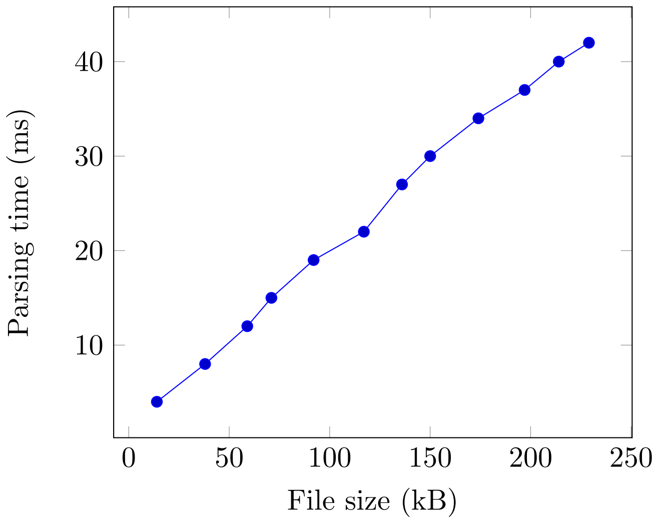

Figure 6.4: Lex/Yacc parse time for Grammar 2

Figure 6.4 shows that the parsing time grows linearly for the Lex/Yacc parser on the input files on Grammar 2.

Parser	**Grammar 1**	**Grammar 2**
Ht-trans	425.3	191.1
Ht-memo	294.5	178.6
Ht-tree	291.2	72.9
Manual/Yacc	7837.0	-
Lex/Yacc	-	4945.1

Table 6.1: Throughput, measured in kB/s, averaged over the twelve different input files for the five different parsers.

The overall throughput for the different parsers is shown in Table 6.1. It can be seen that the fastest Hilltop parser (Ht-trans) for parsing with Grammar 1 is 18.4 times slower than the Manual/Yacc parser, and for Grammar 2 the difference between Ht-trans and Lex/Yacc is by a factor of 25.9.

6.3.3 Discussion

From these results it can be seen that the parsing time performance for a recursive descent parser that uses backtracking is largely linked to the grammar. The difference in performance between Grammar 1 and Grammar 2 is due to how the accepting language of the two grammars are specified. For instance, for Grammar 2, at the beginning of each line for an input file there are 148 different keywords that can appear as the left-most character sequence. This means that in the worst case the parser will successfully match the character sequence on the 148'th try, and on average on the 74'th. These keywords are spread out over 29 different productions of the grammar which means that there are as many different procedures in the parser that may be called when searching for one of these keywords. This both affects parsing time performance and also memory consumption due to context-switching, string comparisons and integer and parse tree node allocations.

The Lex/Yacc parser effectively handles the keyword issue more sophisticated by using lookahead. Whenever the parser enters the state corresponding to when a new line occurs, the parser merely asks for the next token and can then accurately choose what transition to make if there are no shift-reduce or reduce-reduce conflicts. Retrieving the correct token, however, is performed by the lexical analyser.

As for the Hilltop parsers, the Lex/Yacc parser also gets decreased throughput for Grammar 2. However, the decrease is merely 36.9% as opposed to 55.1-75.0% for the Hilltop parsers. The decrease in throughput for the Lex/Yacc parser is most likely due to the increased amount of lexical analysis. For both the Manual/Yacc and the Lex/Yacc parser the lexical analysers are the procedures that consumes most of the parsing time with 17.4% and 26.7% respectively.

For Grammar 1, the reason for the large difference in parsing time for the Ht-memo parser in relation Ht-trans is that a large part of the source files for Grammar 1 evaluate productions that are marked as transient (white spaces and comments). This not only increases the amount of parse tree nodes for the Ht-memo parser, and thus also heap consumption, but it also has a considerable effect on parsing time performance. The amount of inserted elements in the memoization matrix (i.e., the amount of parse tree nodes) created for the Ht-memo parser is 163021 compared with Ht-trans' 49227. This increase in node allocations and memoization matrix operations have a significant effect on the performance for Ht-memo on Grammar 1. This illustrates how effective the transient optimization can be. In addition, this also suggests that optimizing the memoization procedure in Hilltop may result in significant decreases in parse time. Currently, the operations related to the `unordered_map` data structure consumed roughly 15% of the total parse time.

For Grammar 2, the transient optimization shows less effect. This is because the matching of the large amount of keywords consume most of the parse time. In addition, the input files for Grammar 2 consist of less white spaces and comments that are considered as transient.

Two of the most time-consuming tasks for the Hilltop parsers are the allocation of integers and the operations related to string comparisons. In every parsing function there is a number of newly created integers that keeps track of the current index and the results of the evaluation for the different choices. For instance, for the Ht-trans parser on Grammar 2 with the largest input file, the number of integer allocations are roughly four million. For the largest input file for Grammar 1 the number is roughly one and a half million integer allocations. The time spent on performing all of these allocations and freeing them takes up between 35-45% of the parsing time for the Hilltop parsers. For the Manual/Yacc and the Lex/Yacc parser operations related to integer allocations consumes 11-14% of the parsing time.

The second most time-consuming task for the Hilltop parsers is string related operations. The different string operations such as comparison, creation, length checking etc. consumes around 25-30% of the parsing time for the Hilltop parsers.

One possible way of decreasing the amount of string operations performed by the Hilltop parser is to add a lexical analyser. By using a lexical analyser the parser can compare the length of the provided token with the next available terminal symbol in the grammar, and based on the length comparison choose whether to to a string comparison or not. In addition, if a lexical analyser is used the parser does not need to handle white spaces or comments any more. Depending on the grammar, using a lexical analyser could have a beneficial effect on the parsing time. For instance with Grammar 2, where most of the parsing time is spent comparing a large amount of keywords, using a lexical analyser might be an improvement even though most of the keywords are of length three or four since they are assembly instructions. However, the decrease in parsing time needs to be smaller than the overhead of using the lexical analyser.

An interesting note is that Ht-memo is faster than Ht-tree on Grammar 2 and they are both relatively close on parsing time for Grammar 1. This illustrates that packrat parsing is a good choice for these particular grammars as opposed to conventional recursive descent parsing with backtracking.

6.4 Memory Consumption

The memory consumption was measured using the heap profiler `massif` which is part of the memory analysis tool `valgrind` [34] that is built for Linux-based operating systems.

6.4.1 Grammar 1

Figure 6.5 illustrates the total heap consumption by the four parsers used on Grammar 1. As expected, the parsers using memoization consumes the highest amount of heap. This is since the parsers using memoization needs to store results that fail to avoid redundant computations, thus creating a larger parse tree than necessary. The Lex/Yacc and Ht-tree parser only creates a node in the parse tree for successful results, thus creating a ”normal” parse tree.

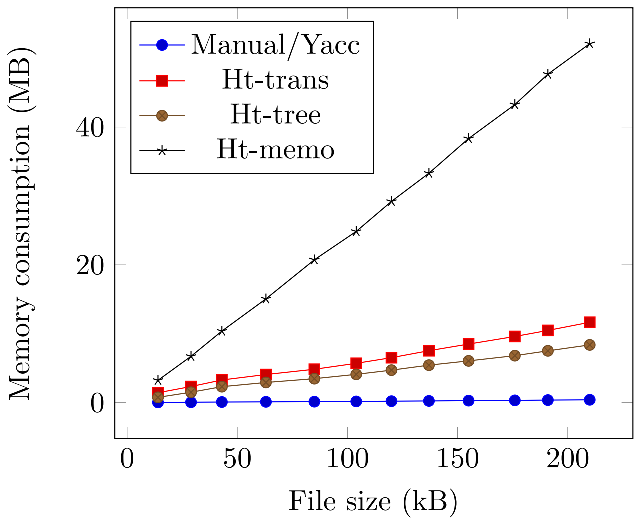

Figure 6.5: Heap consumption for Grammar 1

The increased heap consumption for the Ht-tree parser in relation to the Manual/Yacc parser comes from recursive productions in the Hilltop grammar. For instance, the production that recognize a valid identifier looks as follows:

```
rule T_IDENT
    [_/?/\\/</>] ID
  / [a-zA-Z] ID
end
rule ID
    [a-zA-Z0-9] ID
  / [_/?/\\/</>] ID
  / SPACE
end
```

This will create an `IDNode` containing the matched character for each consecutive character that matches the `ID` production.

The `ID` production could be altered in the following manner:

```
rule ID
  ( [a-zA-Z] / [_/?/\\/</>] ) ( [a-zA-Z0-9] / [_/?/\\/</>] )*
     SPACE
end
```

This production accepts the same language as the previous production but due to how repetition is implemented for the parse tree generation as explained in Section 5.2.1, the same amount of memory will still be consumed. In addition, using repetition suffixes on grammar productions may result in exponential parse times [12]. The reason for this will be explained in Section 9.1.5.

Lex/Yacc on the other hand only creates one node for the entire identifier and thereby reduces its heap consumption in relation to Ht-tree.

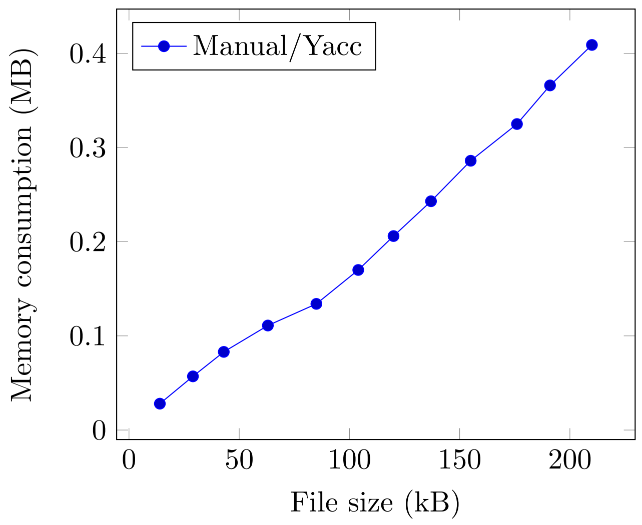

Figure 6.6: Manual/Yacc parser heap consumption for Grammar 1

Figure 6.6 displays how the heap usage scale for the Manual/Yacc parser. The creation of the parse tree is the main source of heap consumption for the Manual/Yacc parser. The parse tree nodes are created as `structs` since the parser is generated as C code. These `structs` are smaller in size than the classes used for the parse tree nodes for the Hilltop parser which makes the difference between the parsers even larger. All four parsers gain a linear increase in heap consumption.

6.4.2 Grammar 2

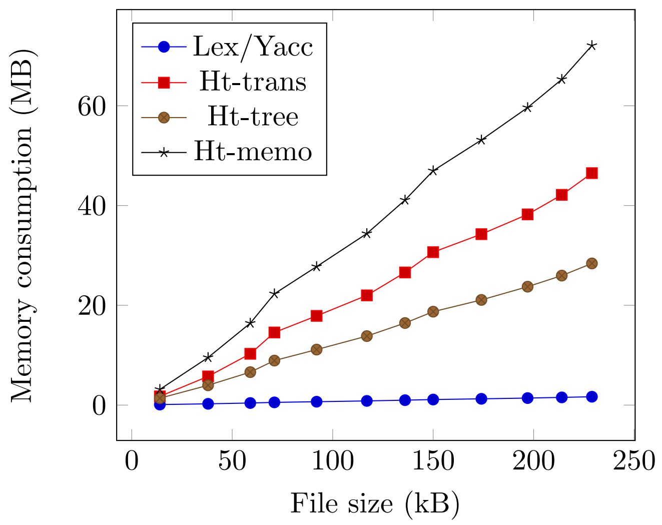

Figure 6.7: Heap consumption for Grammar 2

Figure 6.7 shows the heap consumption for all of the parsers used for Grammar 2. Compared to the heap consumption for Grammar 1, both Ht-trans and Ht-tree has an increased heap usage by roughly a factor of 3. This is related to the increased complexity of the grammar since the amount of created nodes in the parse tree for a production is related to how many nested calls that are required to retrieve a successful result. Another reason for the difference in heap consumption is due to the input files. The input files for Grammar 2 has a smaller ratio of white spaces and comments which are marked as transient. Thus, the amount of parse tree nodes created for the input files of Grammar 2 will be higher and therefore increase the heap consumption. This also applies to the Lex/Yacc parser.

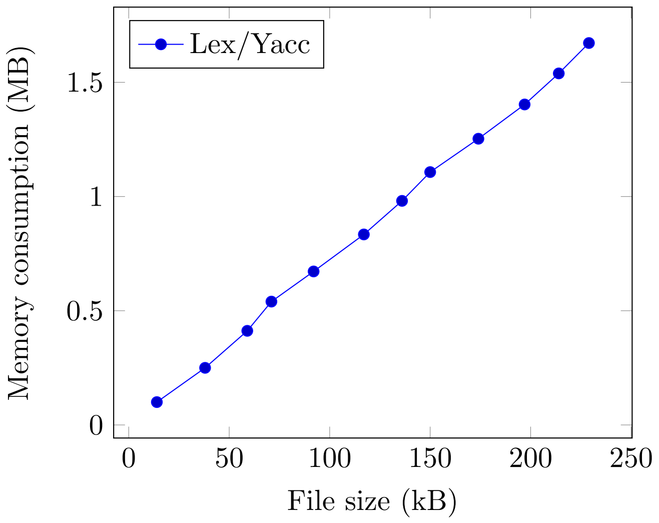

Figure 6.8: Lex/Yacc heap consumption for Grammar 2

The heap consumption for the Lex/Yacc parser for Grammar 2 is illustrated in Figure 6.8. When comparing the heap consumption to Grammar 1, the Lex/Yacc parser has gained the largest increase in heap consumption compared to the other Hilltop parsers. Since the Lex/Yacc produced parser was written in C++ the parse tree creation was more closely mirrored to the Hilltop parse tree creation. This increased the memory footprint since the classes used were larger in size compared to the `structs` used for the C parser. However, the heap consumption for the Lex/Yacc parser was still low when compared with the Hilltop parsers.

Parser	**Grammar 1**	**Grammar 2**
Ht-trans	63.7	185.7
Ht-memo	241.9	292.2
Ht-tree	43.6	117.7
Manual/Yacc	1.8	-
Lex/Yacc	-	7.2

Table 6.2: Byte on the heap per byte of input

By examining Table 6.2 it can be seen that the difference in heap consumption between Grammar 1 and Grammar 2 for the Ht-trans and Ht-tree parser is by a factor of 2.9 and 2.7 respectively. For Ht-memo the same factor is 1.2. Between the Manual/Yacc and Lex/Yacc parsers the difference is by a factor of 4.

6.4.3 Memoization Matrix Size

Figure 6.9 and 6.10 illustrates the size of the memoization matrices for both the Ht-trans and Ht-memo parser for Grammar 1 and Grammar 2. The size of the memoization matrices increase linearly for both parsers. In addition, the memoization matrix size for the Ht-memo parser for the largest input file is 652.1 kB for Grammar 1 and 595.2 kB for Grammar 2. For the Ht-trans parser the same numbers are 196.9 kB and 340.7 kB.

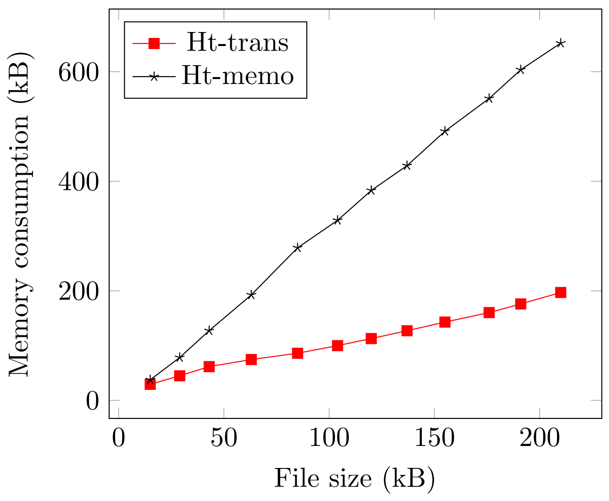

Figure 6.9: Size of the memoization matrices for Grammar 1

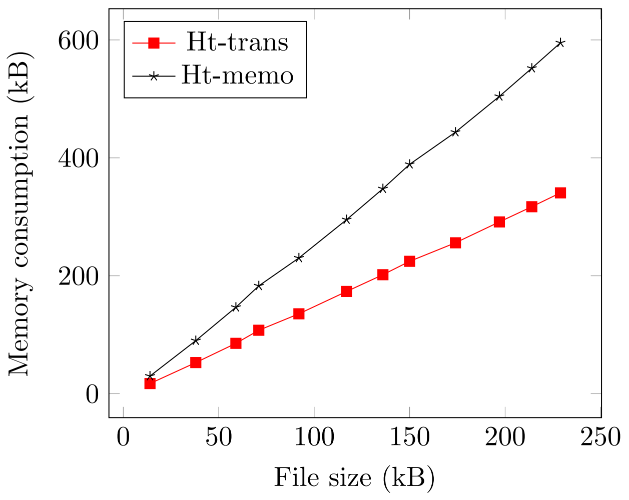

Figure 6.10: Size of the memoization matrices for Grammar 2

The decreased size of the memoization matrix of Ht-memo for Grammar 2 in relation to Grammar 1 is due to that larger part of the input files of Grammar 2 consists of keywords such as XOR, LOAD, ADD etc. For these keywords there are only one parse tree node created and not one node for each character. This decreases the amount of parse tree nodes for the Ht-memo parser for Grammar 2 and thus also the size of the memoization matrix.

The increased size of the memoization matrix for the Ht-trans parser for Grammar 2 in relation to Grammar 1 is due to a decreased amount of white spaces and comments which are marked as transient.

6.4.4 Discussion

As for the parsing time performance, the heap consumption for a packrat parser is related to the grammar. The increased memory footprint for the Lex/Yacc parser compared with the Manual/Yacc is not only related to the increased complexity of the grammar but also due to the implementation of the parse tree generation. If the Manual/Yacc parser was using C++ it could have more closely mirrored the parse tree constructors as used in Hilltop and thus be on more equal terms when compared with the Hilltop parsers. Since the heap consumption for Ht-tree and Lex/Yacc parsers are more or less only related to the size of the resulting parse tree, a change for the Manual/Yacc parser into a C++ version would most likely bring down the factor increase and bring it more closely to the growth for Ht-tree which is increased by a factor of 2.7 instead of 4 as is the case for the Lex/Yacc parser.

The effect of creating one parse tree node for every character part of an identifier as explained in Section 6.4.1 is clearly influencing the heap utilization performance for the Hilltop parsers.

In addition, the increased heap consumption for Ht-memo in relation to Ht-tree and Ht-trans is due to the comments and white spaces in the source files. Whenever a comment expression is found, being it multi-line or single line, the parser uses the *any character* to proceed until the end of the comment production is found, for multi-line comments it is the '*/' symbol and for single lines it is the line break '\n' symbol. For each character inside a comment a parse tree node is created and allocated on the heap. These nodes are not created in the Ht-trans parser and thus gets a decreased heap consumption.

The largest part of the heap consumption for the Ht-trans parser is not the actual memoization matrix. Since the memoization matrix only stores pointers to the parse tree nodes, where the pointers have a size of four bytes, the main source of the increased memory consumption comes from the increased size of the parse tree since a node is created even though it is created from a failed evaluation. This creates a parse tree that contains a lot of unnecessary information since many of the nodes in the tree should not be a part of the resulting parse tree. This could be implemented differently by only storing a `NULL` value for every failed evaluation in the memoization matrix and avoid creating a parse tree node for it, and whenever a `NULL` value is found inside the memoization matrix the parser knows that it results from an unsuccessful evaluation. This would decrease the heap usage of Ht-trans to probably be more on par with Ht-tree, with the only increase being from the size of the memoization matrix. However, the largest input file for Grammar 2 produced a memoization matrix consisting of 85187 elements which only results in a memoization matrix of size 341 kB. An attempt to implement this was made but could not be finalized in this thesis but may still be interesting for future work.

6.5 Result Evaluation

It is hard to pin-point an exact reason for the differences in performance between the Hilltop parsers and the Manual/Yacc and Lex/Yacc parsers. The large difference related to the amount of string operations for the different parsers suggest that it is implementation specific; while the amount of required integer allocations for the Hilltop parsers leans to be caused by the difference of the two parsing techniques. In addition, although Yacc can generate C++ parsers those parsers are mostly C-programs in C++ syntax. If Hilltop was to generate parsers in C instead of C++ there might be a decrease in the performance-gap between the parsers.

However, the difference in parsing time and heap consumption between the Hilltop parsers and the Manual/Yacc and Lex/Yacc parser was expected. Not only due to Hilltop being a packrat parser but also when taking the amount of time spent implementing it into account. There exists several optimizations that can be performed on Hilltop to further improve its parsing time performance [10, 15, 24], some of which will be discussed in the Chapter 9. If some or all of the improvements were to be implemented, and the amount of integer allocations and required string related operations were to be decreased, the Hilltop parser could most likely gain around a factor of ten in decreased parsing time. However, for grammars similar to those used in this thesis work it is believed that a packrat parser will most likely be unable to catch up performance-wise with Lex/Yacc due to its ability to use predictive lookahead.

Chapter 7

Related Work

7.1 Pappy

In [10, 12], Ford presents the first packrat parser generator named *Pappy* which is written in the functional programming language Haskell.

Ford implements a pre-processing stage that is executed on the input grammar. Amongst these alterations are the elimination of direct left recursion by left-factoring the production. However, indirect left recursion is not taken care of, Pappy merely reports an error if such a production is found. Another part of the pre-processing stage is the transformation of iterative ('*' and '+') operators by making the production right-recursive instead.

To reduce the memory consumption of the memoization matrix, Pappy analyses the nonterminals of the grammar and decides whether the memoization for some nonteminals can be discarded. If a nonterminal includes recursive calls inside its alternatives the nonterminal needs to be memoized to ensure the linear parse time. However, if no recursion is involved the time to evaluate the nonterminal will be constant and thus it need not be memoized without breaking the linear parsing time characteristic. Due to the parser having to evaluate such a production every time it is called instead of merely retrieving its memoized value the constant factor of such a production will increase and therefore the overall parse time may increase but still be linear. However, due to the decreased size of the memoization matrix the heap consumption will be smaller and the overall parse time might still gain a beneficial effect due to lower garbage collection overhead and better utilization of the cache.

7.2 *Rats!*

Rats! [14] is a parser generator implemented by R. Grimm in the object-oriented programming language Java, which generates packrat parsers in Java. In [15], Grimm introduces seventeen optimizations to *Rats!*, where thirteen of them are new in relation to Ford's implementation. These optimizations are intended to both decrease the parse time and the amount of memory required for the memoization table. The most significant of the new optimizations is the introduction of the *transient* optimization. This optimization is based on the fact that many productions do not require any backtracking. Thus, memoizing such a production is not necessary. The transient optimization decreased the amount of used memory and parsing time by roughly a factor of two.

Rats! is tested and compared against four different parser generators, namely SDF2 [7], Elkhound [22], ANTLR [30] and JavaCC [29]. A grammar for the Java language is created for all five different parser generators and they are tested on 38 different Java source files. *Rats!* is the parser generator that required the least amount of lines of code for its grammar specification. ANTLR and JavaCC require 50% more, SDF2 need roughly two times more lines of code than *Rats!* and Elkhound requires almost three times the amount.

The performance by the five parser generators is illustrated in Table 5 in [15]. JavaCC consumes more heap memory than *Rats!* even though *Rats!* uses a memoization table. The high heap usage for JavaCC is claimed to be due to its poor AST generation [15]. JavaCC's generated AST bears more resemblance to a basic parse tree and thus the generated AST for JavaCC is a lot larger than the AST generated by *Rats!*. The heap consumption by ANTLR is roughly half the size of *Rats!* usage. The SDF2 and Elkhound parser generators does not report any usage of the heap.

SDF2 and Elkhound produces the slowest parsers considering parse time. ANTLR and JavaCC produces the fastest. The parsers generated by *Rats!* is roughly 1.3 times slower than the corresponding parsers for ANTLR and JavaCC, but the *Rats!* parser is 2.2 times faster than the SDF2 and Elkhound parsers. However, it is also noted that the parsing time is related to the size of the input file. When the size decreases the throughput for a *Rats!*-generated parser increases. This is related to the increased size of the memoization table which increase the amount of needed garbage collection, increased miss rates in the cache etc. In contrast, an ANTLR-generated parser gets increased throughput when the input size gets increased. This suggests that ANTLR-generated parsers have a high startup cost in relation to the other parser generators [15].

7.3 The Cost of Using Memoization

In [3], R. Becket and Z. Somogoyi argues against the use of packrat parsing. The key issue being the use of memoization, which increases the parsing time and memory consumption. To illustrate this, the authors implement a recursive descent parser with backtracking capability defined in a *definite clause grammar* (DCG) in the programming language Mercury that is able to parse the Java language. This parser is then compared with *Rats!* on a set of different Java source files. The tests are performed with and without memoization of the authors own implementation, and with the optimizations of *Rats!* turned on and off. The authors implementation when memoization is used is slower than *Rats!* with optimizations, but faster when the optimizations on *Rats!* are switched off. Similar to the results from Chapter 6, it is also discovered that memoizing specific productions (compared to memoizing *all* productions) generates parsing times that is faster than not memoizing anything at all.

7.4 Packrat Parsing versus Primitive Recursive Descent Parsing

In [33], a primitive recursive descent parser with backtracking is implemented with the use of PEGs to parse Java 1.5. It is discovered that the use of rather straightforward PEG productions can introduce an unwanted behaviour regarding performance. To illustrate this, the author provides an example regarding the parsing of the production *Identifier*:

Identifier = !Keyword Letter LetterOrDigit* Spacing?

Every time this production is invoked the input string is compared with *all* of the keywords (53 in the case for Java 1.5). This is an unnecessary costly operation and can be circumvented by first checking if its a valid *Identifier* by using only *Letter LetterOrDigit** and *then* checking if its a keyword. This avoids the problem of looking through all of the keywords even when the input is not a valid *Identifier*. However, discovering these kinds of productions can be a non-trivial task for more complex productions and PEGs is therefore questioned if it is indeed an advantageous grammar compared with more conventional grammars such as CFGs.

The author states that a large part of programming languages are LL(1) [33], and therefore the unlimited lookahead and backtracking features of packrat parsing is mostly unnecessary, especially since it has a large memory consumption in relation to other parsing techniques.

Chapter 8

Conclusion

The contribution of this thesis has been to compare and evaluate packrat parsing in relation to the lexical analyser Lex and the parser generator Yacc that produces shift-reduce parsers. A packrat parser generator named Hilltop has been implemented for this specific purpose. The parser generators have been used on two different real-world DSL and assembly grammars and received differently sized real-world inputs used in production at IAR Systems. The packrat parsers produced by Hilltop proved inferior to the parsers produced by Lex/Yacc both regarding parsing time (a factor of 18.4-25.9) and memory consumption (by a factor of 25.9-34.9). This comes as no surprise due to the limited amount of time spent on the implementation of Hilltop and thus the lack of optimizations are evident. An improvement to the Hilltop implementation could thus decrease the gap between the two parser generators. However, the results indicate that the performance of the Lex/Yacc combination, for similar grammars as were tested in this thesis work, will never be reached by a packrat parser.

The objective of this thesis, however, has been to test the hypothesis regarding if the performance difference between packrat parsing and Lex/Yacc was *acceptably* small. There has been substantial research on optimizations for packrat parsers [10, 15, 24], some of which were discussed in Chapter 7. It is believed that if some or all of these optimizations are implemented, and if the amount of string related operations and large amount of integer allocations can be reduced, the parsing time could be decreased with a factor up to ten. This would decrease the performance gap between the two parser generators and make the difference in performance acceptably small.

In addition to analysing the performance difference between the two parser generators there were several flaws in Lex/Yacc that was investigated if they could be addressed with the use of packrat parsing:

- *Yacc only accepts LALR(1) grammars.*

  Packrat parser accepts LL(k) grammars which are a larger set of grammars than LALR(1) [9]. However, there was no found practical real-world grammars that were LL(k) but not LALR(1).

- *The Lex and Yacc source files need to be specified separately and follow different syntax.*

  Packrat parsers can be scannerless which enables the grammar writer to include both the lexical analysis and the parsing productions in the same grammar, thus also in the same syntax.

- *In Yacc; the user is required to perform error handling manually, this can make it hard for users inexperienced with shift-reduce parsers to create useful parse error messages.*

The error reporting implementation in Hilltop was not finalized. However, Treetop, the parser generator Hilltop is based on, automatically generates error messages during parse failures. The messages contain information regarding which choices that failed (what the parser expected), what row the error occurred on and what "column" in the text. This should be able to be similarly implemented in Hilltop and thus allow it to also have an automatic error reporting feature instead of having to implement it manually as for the Lex/Yacc solution. This effectively removes the responsibility from the user to have to implement a useful error reporting procedure manually.

- *Lex/Yacc create global identifiers/functions/macros that can cause conflicts if a produced program is used for parsing multiple languages.*

  Hilltop does not support multiple parsers in the same program at the time of the writing of this thesis. However, Treetop supports this feature and is able to handle conflicts if the same production is added from multiple modules. It should be possible to implement a similar solution in Hilltop and thus allow it to support modular grammars.

- *Lex/Yacc specifications are hard to extend and modularise.*

  Packrat parser uses PEGs which have the benefit of being closed under composition, union and intersection. This encourages grammar writers to modularise their grammars and reuse the same modules in several different grammars and thus reduce the time spent on rewriting redundant grammar productions. In addition, this enables the grammars to easily be extended without introducing errors that may seem unrelated, since a PEG is unambiguous by definition.

These results suggest that the question regarding which type of parser generator to choose is connected to different use cases. If the user regards extendibility and modularisation as an important part of a parser generator, then packrat parsing provides a viable solution. If on the other hand performance is of critical interest and grammars are not regularly redefined then Lex/Yacc is most likely the best choice.

8.1 Thesis Evaluation

The goal of the thesis has been to implement a packrat parser generator and compare it with the Lex/Yacc parser generator combination, and also to perform a general comparison between the two different parsing techniques. This goal has been met such that a packrat parser generator has been implemented (although it lacks certain features such as AST support), a general comparison has been conducted and a conclusion has been reached.

The originating time plan was to spend roughly a third of the time for research, a third for implementation and the last third for analysis. This distribution was kept throughout the thesis work with no alterations required. However, during the implementation of Hilltop the parse tree generation took longer than foreseen which lead to a decreased amount of implemented optimizations. The reason for the large amount of time spent on the parse tree generation was because it needed to be implemented from scratch due to the differences between Ruby and C++.

Chapter 9

Future Work

9.1 Future Work

9.1.1 Testing of More Grammars

As shown in Chapter 6, packrat parsers are more sensitive to the grammar specification regarding performance than Lex/Yacc. This implies that not only is it important to generate effective packrat parsers but also that "optimizations" can be done on the grammar specification level. Due to the rather large difference in performance between the two tested grammars it would have been interesting to compare the different parser generators on additional grammars. However, the grammar writing and parser debugging process required a large amount of time and did not make it possible to perform a more empirical testing.

9.1.2 Array as Memoization Data Structure

It would be interesting to test using an array as a data structure instead of a hash table. This would increase the heap consumption but it could still have a positive effect on the parsing time. As explained in Section 5.3.2 the hash function for strings in C++ was found to be ineffective and if arrays were used as a replacement there would be no need for hashing in the first place. In addition, the issue regarding rehashing can be avoided altogether with the use of an array.

9.1.3 Hilltop with a Lexical Analyser

An interesting change to test for the implementation would be to use a lexical analyser, instead of parsing character-by-character. It is unclear if adding a lexical analyser would have any signification effect on the performance, being it positive or negative. For instance, the increased overhead of using a lexical analyser might be higher than the performance gain from using it.

By using a lexical analyser the parser could be able to more effectively decide if a choice should be evaluated at all. For example, if the following production is used:

```
Type  ←  'int' ...
      /  'double' ...
      /  ...
```

If the parser now received a token with five characters; the parser could avoid evaluating the first choice altogether since the first string in the sequence is three characters long and can therefore not match the token. This could reduce the amount of string comparisons done by the parser if there are variations in the length of the sent tokens, which could have a positive effect on the parsing time.

Also, with the addition of a lexical analyser, information such as white spaces and comments can be discarded and not sent to the parser at all. This means that the parser does not need to have productions responsible for handling white spaces and comments. This could decrease the parsing time significantly since the parser would get a reduced amount of memoization and parse tree node allocations, similar to using the transient optimizations on such productions. In addition, the amount of string related operations for the parser would decrease which, as discussed in Section 6.3.3, is responsible for a large portion of the parsing time. However, this decrease in parsing time needs to be larger than the overhead gained from using the lexical analyser.

In addition, it should be possible to decrease the memory consumption by using a lexical analyser. For instance, with a lexical analyser the parser would not need to create one parse tree node for each consecutive character in a self-recursive production (productions identifying label names, variable names etc.), with a lexical analyser only one parse tree node would be needed for the whole token. This behaviour can most likely be implemented in Hilltop without the addition of a lexical analyser. However, it might be an easier transition if a lexical analyser is used.

9.1.4 Merging Choices with Common Prefixes

To reduce the amount of string comparisons needed during parsing merging of choices containing common prefixes could be introduced. Consider the following production:

```
Oper  ←  'ADD' Reg ',' Reg ',' Immediate
      /  'ADD' Reg ',' Reg ',' Reg
      /  'SUB' Reg ',' Reg ',' Immediate
      /  'SUB' Reg ',' Reg ',' Reg
```

This production can be altered in the following manner without affecting the language:

```
Oper  ←  'ADD' Reg ',' Reg ',' (Immediate / Reg)
      /  'SUB' Reg ',' Reg ',' (Immediate / Reg)
```

By doing this transformation the parser now only need to perform a string comparison on 'ADD' and 'SUB' at most once. The production can of course be written like this from the beginning by the grammar writer. However, this produces grammar productions that are sometimes harder to read and understand. It is advisable to instead perform such optimizations automatically at a pre-processing stage with the parser generator.

9.1.5 Iterative Operators

The guaranteed linear parse time for packrat parsers can be violated with iterative ('*' and '$^+$') productions. Consider the following production, taken from [12]: $A \rightarrow x^*$. Such a production may violate the assumption that every cell in the parsing table can be computed in constant time if all of its depending results are already available. If this production is fed with an input string consisting of n x's, the corresponding cells for nonterminal A and this input string will each iterate over the remaining part of the string without taking into account that a result have already been computed. This will make each cell take linear time to compute, and thereby in the worst case, cause the computation of the whole table to run in exponential time [12].

In [12, 15], this problem is addressed by altering such productions into equivalent right-recursive productions. The above example would be altered into the equivalent production: $A \rightarrow xA \mid \epsilon$. Now, the result of each cell is dependent on the result of the cells farther right in the table, and therefore the computation of each cell will take previous results into account. This will preserve the linear time guarantee.

This is currently not implemented in Hilltop. But it could be implemented as a pre-processing stage. The conversion into right-recursive calls for the repetition suffixes were done manually for the experimental results shown in Chapter 6.

9.1.6 Left Recursion

In [37], a modification to the memoziation technique is introduced. The modification is used to make a packrat parser able to parse productions that is left-recursive, both direct and indirect. This effectively removes the issue of having to rewrite the grammar productions that are left-recursive (which is a time-consuming matter from a programmers view). The rewriting can of course be done automatically. However, this may produce code that is hard to read, especially for indirect left recursion, and there are no certainties that the accepted language stays the same [37]. With this modification to support left recursion it is possible to experience exponential parse times. This eliminates one of the beneficial characteristics of packrat parsers which is the guarantee of linear parsing time. However, it is argued that this exponential parsing time is not likely to occur for ordinary grammars [37].

Implementing such a solution as part of the pre-processing stage for Hilltop and analysing if the linear parsing time is not violated could prove to be beneficial. Especially since discovering and rewriting indirect left recursion automatically is a non-trivial matter [10, 15].

9.1.7 *Cut* Operator

In [24], the issue of high memory consumption for packrat parsers is analysed. A *cut* (↑) operator is introduced to signal that no backtracking will occur prior to the current index position. The author uses the following production to illustrate how and why this can work:

```
S  ←  'if' ↑ '(' E ')' S ('else' S)?
   /  'while' ↑ '(' E ')' S
   /  'print' ↑ '(' E ')' S
   /  'set' ↑ I '=' E ';'
```

Due to the nature of ordered choices in PEGs, if the 'if' of the first choice is successfully matched the remaining choices are known to fail. This means that if the E expression of the first choice fails a naive packrat parser would signal a failure and try to match 'while', 'print' and lastly 'set'. By using the *cut* operator the parser avoids performing unnecessary evaluations. This has a positive effect on both the parsing time and memory consumption due to a smaller amount of function calls and a smaller amount of inserted elements into the memoization matrix. In addition, whenever a *cut* operator is found, the memoization matrix can be flushed since there will be no need for backtracking prior to this index position.

This modification turned out to make the memory consumption requiring "almost constant space" no matter how large the input file is [24]. An automated method for inserting cut operators was implemented and tested. The automatic insertion was tested alongside *Rats!* [15] (a heavily optimized packrat parser generator) and the automatic *cut* was superior in both parse time and heap consumption for parsing Java and JSON files. However, at the time of the publication, the automated method used the same amount of memory space as a conventional packrat parser when parsing XML-files, manual insertion of *cut* operators was needed. For the Java and JSON grammars, when the *cut* optimization is switched off, the difference in parsing performance drops by a factor between two and three, and the heap consumption goes from constant to linear.

Among the found optimizations for packrat parser, the *cut* operator is the one with the largest impact on both parsing time and heap consumption. A successful implementation of this optimization into Hilltop would possibly improve its performance in the same manner and therefore make it closer to the performance of the Lex/Yacc combination.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman. The Theory of Parsing, Translation, and Compiling. Upper Saddle River, NJ, USA, 1972. Prentice-Hall, Inc.

[3] Ralph Becket and Zoltan Somogyi. DCGs + Memoing = Packrat Parsing But is it worth it? In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] A. Birman and J. D. Ullman. Parsing Algorithms with Backtrack. *Journal of Information and Control*, 23(1):1–34, 1973.

[5] Alexander Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, Department of Electrical Engineering, Princeton, NJ, USA, 1970.

[6] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The `metafront` System: Extensible Parsing and Transformation. *Journal of Electronic Notes in Theoretical Computer Science*, 82(3):592–611, 2003.

[7] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation Without Restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 365–383, New York, NY, USA, 2004. ACM.

[8] ISO/IEC/JTC1/SC22/WG14 C. WG14/N1256 Committee Draft. `http://www.open-std.org/jtc1/sc22/wg14/`. [Online; accessed 2014-06-16].

[9] F. L. Deremer. *Practial Translators For LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, Cambridge, MA, USA, 1969.

[10] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time - Functional Pearl. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 36–47, New York, NY, USA, 2002. ACM.

[11] Bryan Ford. Packrat Parsing and Parsing Expression Grammars. `http://bford.info/packrat/`. [Online; accessed 2014-06-22].

[12] Bryan Ford. Packrat Parsing: A Practical Linear-time Algorithm with Backtracking. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engeneering and Computer Science, Cambridge, MA, USA, 2002.

[13] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[14] Robert Grimm. Practical Packrat Parsing. Technical Report TR2004-854, New York University, Department of Computer Science, 2004.

[15] Robert Grimm. Better Extensibility Through Modular Syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[16] The Open Group. The Single UNIX Specification, Version 3. `http://www.unix.org/version3/online.html`. [Online; accessed 2014-06-24].

[17] The Open Group. The Single UNIX Specification, Version 4. `http://www.unix.org/version4/`. [Online; accessed 2014-06-23].

[18] Clifford Heath. Treetop: Bringing the syntactic simplicity of Ruby to syntactic analysis. `http://cjheath.github.io/treetop/index.html`. [Online; accessed 2014-06-24].

[19] Stephen C Johnson. Yacc: Yet Another Compiler-Compiler. Bell Laboratories Murray Hill, NJ, 1975.

[20] M. E. Lesk and E. Schmidt. UNIX Vol. II. chapter Lex; a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[21] John Levine. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[22] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. Berkeley, CA, USA, 2003. University of California at Berkeley.

[23] D. Michie. Memo Functions and Machine Learning. *Journal of Nature*, 218:19–22, 1968.

[24] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 29–36, New York, NY, USA, 2010. ACM.

[25] Microsoft Developer Network. Aquiring high-resolution time stamps (Windows). `http://msdn.microsoft.com/en-us/library/dn553408%28v=vs.85%29.aspx`. [Online; accessed 2014-08-15].

[26] Microsoft Developer Network. QeuryPerformanceCounter function (Windows). `http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904%28v=vs.85%29.aspx`. [Online; accessed 2014-06-23].

[27] The C++ Resources Network. map - C++ Reference. `http://www.cplusplus.com/reference/map/map/`. [Online; accessed 2014-02-20].

[28] The C++ Resources Network. unordered_map - C++ Reference. `http://www.cplusplus.com/reference/unordered_map/unordered_map/`. [Online; accessed 2014-02-16].

[29] Cognisync Oracle, Project Kenai. Javacc is a parser/scanner generator... - Project Kenai. `https://java.net/projects/javacc`. [Online; accessed 2014-06-16].

[30] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Journal of Software - Practice and Experience*, 25(7):789–810, 1995.

[31] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates To LL(k): Pred-LL(k). In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, pages 263–277, London, UK, UK, 1994. Springer-Verlag.

[32] Peter Pepper. LR Parsing = Grammar Transformation + LL Parsing - Making LR Parsing More Understandable and More Efficient. Technical Report 99-5, TU Berlin, 1999.

[33] R. R. Redziejowski. Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking. *Journal of Fundamenta Informaticae*, 79(3-4):513–524, 2007.

[34] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., UK, 2008.

[35] IAR Systems. IAR Systems - World leading provider of software development tools for embedded systems. `http://www.iar.com/`. [Online; accessed 2014-06-25].

[36] Dave Thomas, Andy Hunt, and Chad Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2013.

[37] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers Can Support Left Recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 103–110, New York, NY, USA, 2008. ACM.

Appendix A

A trivial grammar

Listing A.1: Grammar specification for a trivial language which accepts simple arithmetic expressions.

```
grammar Arithmetic
  rule term
    number (op term)*
  end

  rule op
    '*' / '/' / '-' / '+'
  end

  rule number
    '-'? [1-9] [0-9]*
  end
end
```

Appendix B

Treetop parser

Listing B.1: Packrat parser for the grammar specified in Listing A.1, generated by Treetop. Irrelevant parts have been removed for brevity.

```
# Autogenerated from a Treetop grammar. Edits may be lost.


module Arithmetic
  include Treetop::Runtime

  def _nt_term
    start_index = index
    if node_cache[:term].has_key?(index)
      cached = node_cache[:term][index]
      if cached
        cached = SyntaxNode.new(input, index...(index + 1)) if
           cached == true
        @index = cached.interval.end
      end
      return cached
    end

    i0, s0 = index, []
    r1 = _nt_number
    s0 << r1
    if r1
      s2, i2 = [], index
      loop do
        i3, s3 = index, []
        r4 = _nt_op
        s3 << r4
        if r4
          r5 = _nt_term
          s3 << r5
        end
        if s3.last
          r3 = instantiate_node(SyntaxNode,input, i3...index, s3)
        r3.extend(Term0)
      else
        @index = i3
        r3 = nil
      end
      if r3
        s2 << r3
      else
        break
      end
    end
    r2 = instantiate_node(SyntaxNode,input, i2...index, s2)
    s0 << r2
  end
  if s0.last
    r0 = instantiate_node(SyntaxNode,input, i0...index, s0)
    r0.extend(Term1)
  else
    @index = i0
    r0 = nil
  end

  node_cache[:term][start_index] = r0

  r0
end

def _nt_op
  start_index = index
  if node_cache[:op].has_key?(index)
    cached = node_cache[:op][index]
    if cached
      cached = SyntaxNode.new(input, index...(index + 1)) if cached == true
      @index = cached.interval.end
    end
    return cached
  end

  i0 = index
  if has_terminal?('*', false, index)
    r1 = instantiate_node(SyntaxNode,input, index...(index + 1))
    @index += 1
  else
    terminal_parse_failure('*')
    r1 = nil
  end
  if r1
```

```
      r0 = r1
    else
      if has_terminal?('/', false, index)
        r2 = instantiate_node(SyntaxNode,input, index...(index +
           1))
        @index += 1
      else
        terminal_parse_failure('/')
        r2 = nil
      end
      if r2
        r0 = r2
      else
        if has_terminal?('-', false, index)
          r3 = instantiate_node(SyntaxNode,input, index...(index
             + 1))
          @index += 1
        else
          terminal_parse_failure('-')
          r3 = nil
        end
        if r3
          r0 = r3
        else
          if has_terminal?('+', false, index)
            r4 = instantiate_node(SyntaxNode,input, index...(
               index + 1))
            @index += 1
          else
            terminal_parse_failure('+')
            r4 = nil
          end
          if r4
            r0 = r4
          else
            @index = i0
            r0 = nil
          end
        end
      end
    end

    node_cache[:op][start_index] = r0

    r0
  end

  def _nt_number
    start_index = index
    if node_cache[:number].has_key?(index)
      cached = node_cache[:number][index]
      if cached
        cached = SyntaxNode.new(input, index...(index + 1)) if
           cached == true
        @index = cached.interval.end
      end
      return cached
    end

    i0, s0 = index, []
    if has_terminal?('-', false, index)
      r2 = instantiate_node(SyntaxNode,input, index...(index + 1)
         )
      @index += 1
    else
      terminal_parse_failure('-')
      r2 = nil
    end
    if r2
      r1 = r2
    else
      r1 = instantiate_node(SyntaxNode,input, index...index)
    end
    s0 << r1
    if r1
      if has_terminal?('\G[1-9]', true, index)
        r3 = true
        @index += 1
      else
        r3 = nil
      end
      s0 << r3
      if r3
        s4, i4 = [], index
        loop do
          if has_terminal?('\G[0-9]', true, index)
            r5 = true
            @index += 1
          else
            r5 = nil
          end
          if r5
            s4 << r5
          else
            break
          end
        end
        r4 = instantiate_node(SyntaxNode,input, i4...index, s4)
        s0 << r4
      end
    end
    if s0.last
      r0 = instantiate_node(SyntaxNode,input, i0...index, s0)
      r0.extend(Number0)
    else
      @index = i0
      r0 = nil
    end

    node_cache[:number][start_index] = r0

    r0
  end

end

class ArithmeticParser < Treetop::Runtime::CompiledParser
  include Arithmetic
end
```

Appendix C

Hilltop parser

Listing C.1: Packrat parser for the grammar specified in Listing A.1, generated by Hilltop. Irrelevant parts have been removed for brevity.

```cpp
#include "ast_gen.h"
/* A C++ generated parser from a Treetop grammar.*/

namespace Arithmetic
{
  // Forward declarations
  int Ht_term(const std::string& input, int index, Ast::termNode
     *tree, MemoTable *memo_table);
  int Ht_op(const std::string& input, int index, Ast::opNode *
     tree, MemoTable *memo_table);
  int Ht_number(const std::string& input, int index, Ast::
     numberNode *tree, MemoTable *memo_table);

  enum {
  enum_term,
  enum_op,
  enum_number
  };

  int
  Ht_term(const std::string& input, int index, Ast::termNode *
     tree, MemoTable *memo_table)
  {
    const int start_index = index;
    MemoTable::iterator it = (*memo_table).find(index);
    if (it != (*memo_table).end())
    {
      InnerMemoTable::iterator it2 = it->second.find(enum_term);
      if (it2 != it->second.end())
      {
        tree = dynamic_cast<Ast::termNode*>((*memo_table)[index][
           enum_term]);
        return tree->getIndex();
      }
    }
    tree->term0 = new Ast::termNode0();
    const int i0 = index;
    int r0 = -1;
    int r1 = -1;
    const int i1 = index;
    Ast::numberNode *s1 = new Ast::numberNode();
    r1 = Ht_number(input, index, s1, memo_table);
    tree->term0->number0 = s1;
    if (r1 >= 0)
    {
      index = r1;
      r0 = r1;
      int r2 = -1;
      while (1)
      {
        const int i2 = index;
        Ast::termNode0::nested_rep_class0 *acc0 = new Ast::
           termNode0::nested_rep_class0();
        const int i3 = index;
        int r3 = -1;
        int r4 = -1;
        const int i4 = index;
        Ast::opNode *s4 = new Ast::opNode();
        r4 = Ht_op(input, index, s4, memo_table);
        acc0->op1 = s4;
        if (r4 >= 0)
        {
          index = r4;
          r3 = r4;
          int r5 = -1;
          const int i5 = index;
          Ast::termNode *s5 = new Ast::termNode();
          r5 = Ht_term(input, index, s5, memo_table);
          acc0->term2 = s5;
          if (r5 >= 0)
          {
            index = r5;
            r3 = r5;
          }
          else
          {
            index = -1;
            r3 = -1;
          }
        }
        else
        {
```

```
        index = -1;
        r3 = -1;
      }
      if (r3 >= 0)
      {
        index = r3;
      }
      else
      {
        index = i2;
        break;
      }
      tree->term0->repetition0.push_back(*acc0);
    }
    r2 = index;
    if (r2 >= 0)
    {
      index = r2;
      r0 = r2;
    }
    else
    {
      index = -1;
      r0 = -1;
    }
  }
  else
  {
    index = -1;
    r0 = -1;
  }
  if (index >= start_index)
  {
    index = r0;
  }
  else
  {
    index = -1;
  }
  tree->setIndex(index);
  (*memo_table)[start_index][enum_term] = tree;
  return index;
}
int
Ht_op(const std::string& input, int index, Ast::opNode *tree, MemoTable *memo_table)
{
  const int start_index = index;
  MemoTable::iterator it = (*memo_table).find(index);
```

```
    if (it != (*memo_table).end())
    {
      InnerMemoTable::iterator it2 = it->second.find(enum_op);
      if (it2 != it->second.end())
      {
        tree = dynamic_cast<Ast::opNode*>((*memo_table)[index][enum_op]);
        return tree->getIndex();
      }
    }
    const int i0 = index;
    int r0 = -1;
    tree->op0 = new Ast::opNode0();
    int r1 = -1;
    const int i1 = index;
    r1 = Packrat::TerminalCheck(input, '*', index);
    if (r1 >= 0)
    {
      tree->op0->updateVal(input.substr(index, std::string('*').length()));
      index = r1;
    }
    if (r1 >= 0)
    {
      r0 = r1;
      index = r1;
    }
    else
    {
      index = i1;
      tree->op1 = new Ast::opNode1();
      int r2 = -1;
      const int i2 = index;
      r2 = Packrat::TerminalCheck(input, '/', index);
      if (r2 >= 0)
      {
        tree->op1->updateVal(input.substr(index, std::string('/').length()));
        index = r2;
      }
      if (r2 >= 0)
      {
        r0 = r2;
        index = r2;
      }
      else
      {
        index = i2;
        tree->op2 = new Ast::opNode2();
        int r3 = -1;
        const int i3 = index;
        r3 = Packrat::TerminalCheck(input, '-', index);
        if (r3 >= 0)
        {
          tree->op2->updateVal(input.substr(index, std::string('-
             ').length()));
          index = r3;
        }
        if (r3 >= 0)
        {
          r0 = r3;
          index = r3;
        }
        else
        {
          index = i3;
          tree->op3 = new Ast::opNode3();
          int r4 = -1;
          const int i4 = index;
          r4 = Packrat::TerminalCheck(input, '+', index);
          if (r4 >= 0)
          {
            tree->op3->updateVal(input.substr(index, std::string(
               '+').length()));
            index = r4;
          }
          if (r4 >= 0)
          {
            r0 = r4;
            index = r4;
          }
          else
          {
            index = i0;
          }
        }
      }
    }
    if (index >= start_index)
    {
      index = r0;
    }
    else
    {
      index = -1;
    }
    tree->setIndex(index);
    (*memo_table)[start_index][enum_op] = tree;
    return index;
  }

  int
  Ht_number(const std::string& input, int index, Ast::numberNode
     *tree, MemoTable *memo_table)
  {
    const int start_index = index;
    MemoTable::iterator it = (*memo_table).find(index);
    if (it != (*memo_table).end())
    {
      InnerMemoTable::iterator it2 = it->second.find(enum_number)
         ;
      if (it2 != it->second.end())
      {
        tree = dynamic_cast<Ast::numberNode*>((*memo_table)[index
           ][enum_number]);
        return tree->getIndex();
      }
    }
    tree->number0 = new Ast::numberNode0();
    const int i0 = index;
    int r0 = -1;
    int r1 = -1;
    int r2 = -1;
    const int i2 = index;
    r2 = Packrat::TerminalCheck(input, '-', index);
    if (r2 >= 0)
    {
      tree->number0->updateVal(input.substr(index, std::string('-
         ').length()));
      index = r2;
    }
    if (r2 >= 0)
    {
      r1 = r2;
    }
    else
    {
      r1 = index;
    }
    if (r1 >= 0)
    {
      index = r1;
      r0 = r1;
      int r3 = -1;
      const int i3 = index;
      r3 = Packrat::ChoiceOrRangeCheck(input, "1-9", index);
      if (r3 >= 0)
      {
        tree->number0->updateVal(std::string(1,input[index]));
      }
      if (r3 >= 0)
      {
        index = r3;
        r0 = r3;
        int r4 = -1;
        while (1)
        {
          const int i4 = index;
          int r5 = -1;
          const int i5 = index;
          r5 = Packrat::ChoiceOrRangeCheck(input, "0-9", index);
          if (r5 >= 0)
          {
            tree->number0->updateVal(std::string(1,input[index]));
          }
          if (r5 >= 0)
          {
            index = r5;
          }
          else
          {
            index = i4;
            break;
          }
        }
        r4 = index;
        if (r4 >= 0)
        {
          index = r4;
          r0 = r4;
        }
        else
        {
          index = -1;
          r0 = -1;
        }
      }
      else
      {
        index = -1;
        r0 = -1;
      }
    }
    else
    {
        index = -1;
        r0 = -1;
      }
      if (index >= start_index)
      {
        index = r0;
      }
      else
      {
        index = -1;
      }
      tree->setIndex(index);
      (*memo_table)[start_index][enum_number] = tree;
      return index;
    }
}
```

Appendix D

Hilltop parse tree generation

Listing D.1: The parse tree constructed for the grammar specified in Listing A.1, generated by Hilltop. Irrelevant parts have been removed for brevity.

```
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

#include <vector>

namespace Ast
{
  class termNode;
  class termNode0;
  class opNode;
  class opNode0;
  class opNode1;
  class opNode2;
  class opNode3;
  class numberNode;
  class numberNode0;

  class AST
  {
    public:
    virtual ~AST() {}
    virtual std::string getVal() = 0;
    virtual void updateVal(std::string acc) = 0;
  };

  class termNode : public AST
  {
    std::string val;
    int index;
    public:
    termNode0 *term0;

    void print(int indent);
```

```cpp
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }

    termNode() :
    term0() {}
  };

  class termNode0 : public AST
  {
    std::string val;
    int index;
    public:
    numberNode *number0;

    class nested_rep_class0 : public AST
    {
      std::string val;
      int index;
      public:
      opNode *op1;
      termNode *term2;

      nested_rep_class0() :
      op1(),
      term2() {}

      void print(int indent);
      std::string getVal() { return val; }
      void updateVal(std::string acc) { val += acc; }
      void setIndex(int idx) { index = idx; }
      int getIndex() { return index; }
    };
    std::vector< nested_rep_class0 > repetition0;

    termNode0() :
    number0(),
    repetition0() {}

    void print(int indent);
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }
  };

  class opNode : public AST
  {
```

```
  std::string val;
  int index;
  public:

  opNode0 *op0;
  opNode1 *op1;
  opNode2 *op2;
  opNode3 *op3;

  void print(int indent);
  std::string getVal() { return val; }
  void updateVal(std::string acc) { val += acc; }
  void setIndex(int idx) { index = idx; }
  int getIndex() { return index; }

  opNode() :
  op0(),
  op1(),
  op2(),
  op3() {}
};

class opNode0 : public AST
{
  std::string val;
  int index;
  public:

  opNode0() {}

  void print(int indent);
  std::string getVal() { return val; }
  void updateVal(std::string acc) { val += acc; }
  void setIndex(int idx) { index = idx; }
  int getIndex() { return index; }
};

class opNode1 : public AST
{
  std::string val;
  int index;
  public:

  opNode1() {}

  void print(int indent);
  std::string getVal() { return val; }
  void updateVal(std::string acc) { val += acc; }
  void setIndex(int idx) { index = idx; }
```

```
    int getIndex() { return index; }
  };

  class opNode2 : public AST
  {
    std::string val;
    int index;
    public:

    opNode2() {}

    void print(int indent);
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }
  };

  class opNode3 : public AST
  {
    std::string val;
    int index;
    public:

    opNode3() {}

    void print(int indent);
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }
  };

  class numberNode : public AST
  {
    std::string val;
    int index;
    public:
    numberNode0 *number0;

    void print(int indent);
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }

    numberNode() :
    number0() {}
  };

  class numberNode0 : public AST
  {
    std::string val;
    int index;
    public:

    class nested_rep_class0 : public AST
    {
      std::string val;
      int index;
      public:

      nested_rep_class0() {}

      void print(int indent);
      std::string getVal() { return val; }
      void updateVal(std::string acc) { val += acc; }
      void setIndex(int idx) { index = idx; }
      int getIndex() { return index; }
    };
    std::vector< nested_rep_class0 > repetition0;

    numberNode0() :
    repetition0() {}

    void print(int indent);
    std::string getVal() { return val; }
    void updateVal(std::string acc) { val += acc; }
    void setIndex(int idx) { index = idx; }
    int getIndex() { return index; }
  };
}
#endif
```