POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

PARALLEL SCALABLE PARSING WITH FLOYD GRAMMAR

Relatore: Prof. Stefano CRESPI REGHIZZI

Correlatore: Ing. Alessandro BARENGHI

Tesi di Laurea di:

Valerio PONTE, Matricola n. 755331

Ermes VIVIANI, Matricola n. 754853

Anno Accademico 2010–2011

Για την Παραλλαιόλου και την Παρσεφόνη.

Σας ευχαριστούμε παιδιά.

Acknowledgments

This thesis is the result of the work of about a year. It would not have been possible without the support and guidance of various persons. In particular, we would like to express our thanks to professor Crespi Reghizzi Stefano for his advice and willingness to help us whenever we needed; Ph.D. Barenghi Alessandro for his technical knowledge and revisions of our code and this thesis; professor Mandrioli Dino and Pradella Matteo for their contributions and interest given during several meetings and brainstorming sessions.

Finally we would like to thank our families for their support during all these years of study.

Abstract

Recent development in the hardware sector has made parallelized architectures almost pervasive. While parallel solutions have been found for many common problems, this is not the case for parsing, where traditional techniques are inherently sequential, except for specific ad-hoc solutions. From recent studies in the field of formal languages, and in particular Floyd grammars, the possibility for a parallel scalable language-independent algorithm has surfaced. In this thesis the study of the problem is dealt with and a possible solution is presented. As a proof of concept the development of the algorithm and experimental results comparing it with traditional sequential solutions accompany this work. This thesis is part of a Research Project that won a Google Research Award.

Contents

List of Figures

List of Tables

Sommario

Recenti sviluppi nel settore dell'hardware hanno portato ad una diffusione sempre più capillare di architetture dotate di unità di calcolo multiple. Processori multi-core si trovano ormai dai centri di ricerca ai dispositivi mobili, passando per i calcolatori general purpose. Questi sviluppi nel settore dell'hardware hanno portato ad una spinta anche nel campo del software perché esso si adatti a questi cambiamenti; in particolare questo ha portato a ricerche crescenti nel campo del calcolo parallelo. Mentre per molti problemi comuni sono già state trovate e vengono comunemente utilizzate delle soluzioni alternative a quelle tradizionali che sfruttino il parallelismo, questo non è vero nel campo della parsificazione. Infatti, poiché gli algoritmi tradizionalmente impiegati in questo campo sono inerentemente sequenziali, adattarne il loro utilizzo ad un contesto parallelo non sembra una scelta praticabile.

In questo lavoro di tesi è stato studiato un approccio differente al problema della parsificazione che sfrutta le proprietà teoriche delle grammatiche ad operatori (*grammatiche di Floyd*). Originariamente formalizzate da Robert W. Floyd (da cui il nome), per le quali aveva anche introdotto un algoritmo di analisi sintattica, queste grammatiche furono successivamente abbandonate in favore delle più espressive grammatiche libere dal contesto e dai loro parsificatori. A causa di lavori recenti relativi alle loro proprietà, sono state riprese in considerazione per un adattamento ad un contesto parallelo.

Il lavoro di questa tesi si è occupato di studiare queste proprietà per poterle sfruttare per lo sviluppo di un algoritmo di parsificazione parallela effettivamente impiegabile in contesti applicativi reali. In particolare, questa tesi ha visto come risultati la progettazione di un algoritmo di parsificazione parallela, la sua prototipazione avvenuta per mezzo di un programma realizzato in Python atto a dimostrare la correttezza dell'algoritmo e a fornire i primi risultati sperimentali, ed infine un'implementazione a più basso livello in C volta ad essere un punto di inizio per una possibile realizzazione che possa competere ed eventualmente superare gli algoritmi che al momento costituiscono lo stato dell'arte degli analizzatori sintattici. Conclude la tesi un capitolo contenente i risultati sperimentali ottenuti utilizzando gli strumenti software realizzati e delle considerazioni finali relative al lavoro svolto. Questa tesi si inserisce all'interno di un Progetto di Ricerca vincitore di un Google Research Award.

Chapter 1

Introduction

This work belongs to the research area of formal languages and compilers; in particular, it focuses on the study of parsing techniques. The purpose of this thesis is to design and implement a new parsing method, which can obtain substantial advantages over traditional parsers on parallel architectures. This has been done exploiting the formal properties of the so called *Floyd grammars* [1], also known as *operator precedence grammars.*

Recent development in the hardware sector has made parallelized architectures almost pervasive. Nearly every general purpose CPU produced nowadays is multi-core, GPUs are more and more used for heavily parallel computations and ARM is producing various multi-core embeddable processor models [2, 3]. In general, parallel hardware stands for hardware capable of parallel computing, *i.e.* a form of computation in which many different calculations are performed simultaneously. Parallel computation has the obvious advantage of improving performances, since executing the same computations sequentially is slower that executing them simultaneously on different processors. As it has been stated by Amdahl [4] the maximum achievable speedup on a parallel architecture can be defined as

$$\frac{1}{(1-P)+\frac{P}{N}}$$

where P is the portion of the program that may be parallelized and N is the number of processors employed to perform the computation. Theoretically, if P is 1, *i.e.* the entire problem is parallelized, the maximum speedup is $\frac{N}{P}$. Unfortunately, in practice the speedup drops rapidly as N is increased, since in general $P < 1$ and thus the maximum speedup has a fixed saturation level.

The drawbacks of parallelization is that not every problem can be solved in parallel (intrinsically sequential problems) and that, in general, finding an adequate parallel formulation of a problem is more difficult than finding the corresponding sequential one. This is particularly true in the field of parsing, where the most general and diffused algorithms are intrinsically sequential. For instance, as we will see in chapter 2, an LR(1) parser has the concept of state [5], which represents the rules that could possibly be reduced while reading a token; the state depends on the entire prefix of the string analyzed up to that moment. A parallel parser could possibly analyze a substring which may not be a valid prefix of the language and therefore the initial state (the state in which the parser is found when starting the analysis of the substring) is generally unknown. Therefore it is evident that a parallel parser could not easily use such a paradigm in order to work.

However, for some particular languages, it seems possible for the state not to depend on the entire prefix but only on a limited amount of preceding tokens; therefore, the analysis of a string belonging to such a language could possibly be parallelized using another method. Let us take for instance a generic *Dyck language* [5]; in order to correctly recognize a certain parentheses structure it is sufficient to know its starting and ending points. It is obvious that, splitting a string into different substrings, it is always possible to reduce the balanced parentheses structures, *i.e.* structures in the form '(s)' where '(' is the opening bracket, 's' is a generic substring and ')' is the closing bracket. Therefore a parallel parser may try to recognize all the balanced parentheses structures in a given substring and then try to reunite them with the whole string.

Such an approach (adequately extended and modified) may be applied to a wider range of languages. In fact, Dyck languages have little to no use in a real world context and therefore a more general method, applicable to more powerful languages, is needed. For instance, real world languages such as *XML* or *JSON* have an evident parenthesized structure, and thus could benefit a lot from a parallel analysis. In particular, in our thesis we focused on designing and developing a parallel parsing algorithm for the languages generated by *Floyd grammars* (from now on indicated as FGs), which can include languages such as the ones cited above. Even if *operator precedence parsing*, *i.e.* the parsing method used for FGs, is one of the oldest methods [6], it has not been researched or employed much since its introduction. In fact, in order to be used, it requires the target grammar to be expressed in a restrictive way. However, its reintroduction is justified by recent research [1] which have shown more opportunities for them to be employed in a parallel computation environment than traditional parsers, such as *LR* or *LL*, which have been thought to accept less restrictive grammars and optimized for sequential computations.

The reason behind the attempt to parallelize parsing, despite that current state of the art sequential parsers are very efficient, comes from the fact that there exist application fields in which parsing is done extensively. These may be identified, for instance, in semi-structured data analysis, web spiders, compilation and anti-virus applications. It is evident that, if parsing is one of the main activities of a particular application there is interest in doing it as efficiently as possible.

The developed method produced interesting results in terms of speedup, making it and its possible future developments a feasible and reasonable choice for a parser, both in a limited parallel environment, *e.g.* desktop or mobile devices with few cores used to browse the web, and in highly parallel environments, *e.g.* server machines that analyze large quantities of semi-structured data. For this reason, the Research Project of which this thesis constitutes the proof of concept, has won a Google Research Award [7].

Our work is presented in the following chapters; in chapter 2 we introduce current state of the art parsing methods; in particular we focus on general sequential techniques and ad-hoc parallel parsers, *i.e.* developed only for specific languages. In chapter 3 we discuss FGs and their theoretical properties which serve in order to explain why the languages generated by such grammars can be easily parsed in a parallel way. In chapter 4 we describe in detail the parallel parsing algorithm we developed. In chapter 5 we describe the optimizations we did or planned to do both on the algorithm and on its implementation. In chapter 6 we present a suite of experimental activities we performed and we discuss the results obtained. In chapter 7 we give a final evaluation of the implemented algorithm and its future developments.

Chapter 2

Main parsing methods

In this chapter we give an introduction to the lexical analysis and parsing processes, followed by a section in which we describe what are the current state of the art parsing techniques and we address issues that prevent their parallelization; in the last section we present some of the already existent parallel parsers that have been developed for a specific language.

Since, as we will see, the parsing process works with tokens, *i.e.* a string of characters categorized as a symbol, these have to be generated. This is done by what is called a *lexer*. Tokens are defined according to a lexical grammar, *i.e.* a set of rules which bind sequences of characters to a symbol. A typical lexer is composed of a *scanner* and an *evaluator*. The scanner is used to identify the different symbols inside the string, while the evaluator is used to give each of them their *semantic value*, *i.e.* the meaning that a certain token holds with respect to the grammar that will be used during the parsing phase.

For instance, if we consider the string $35 + 4$ and the following lexical grammar:

$$\begin{aligned} \{1-9\}\{0-9\}^* &\rightarrow NUMBER \\ \{+\} &\rightarrow TPLUS \\ \{\times\} &\rightarrow TTIMES \\ \{(\} &\rightarrow LPAR \\ \{)\} &\rightarrow RPAR \end{aligned}$$

the result of the scanning process will be $NUMBER_0, TPLUS_0, NUMBER_1$ and the evaluation phase will associate to $NUMBER_0$ the value 35 and to $NUMBER_1$ the value 4. We can observe that not every tokens has a semantic value, as in this example $TPLUS_0$ does not have one.

In general, for *parsing* we mean the analysis of a sequence of tokens s, one at a time, with respect to a given formal grammar G, determining the syntactical structure of the string, in order to define whether it belongs to the language $L(G)$ generated by G. Although the given definition is correct, nowadays most parsers, when recognizing a string, build a syntactic parse tree, *i.e.* a data structure representing the syntactic structure of the string, that holds the semantic values of the tokens in the string.

Let us consider a formal grammar G defined as

$$G =< V_N, \Sigma, P, S >$$

where V_N is the set of symbols called *nonterminal* symbols, *i.e.* the nonterminal alphabet, Σ is the set of symbols called *terminal* symbols which correspond to the possible tokens generated by the lexer, *i.e.* the terminal alphabet, P is the set of *syntactic production rules* and $S \in V_N$ is a particular nonterminal symbol called *axiom*. For instance, if G is used to express parenthesized arithmetic expressions, it can be defined as

$$
\begin{aligned}
V_N = &\ \{S,\ E,\ T,\ F\} \\
\Sigma = &\ \{a,\ +,\ \times,\ (,\ )\} \\
P = &\ \{ \\
&\quad S \to E \\
&\quad E \to E + T \ \mid \ T \\
&\quad T \to T \times F \ \mid \ F \\
&\quad F \to (\ \ E\ \ ) \ \mid \ a \\
&\ \}
\end{aligned}
$$

Consider the string $s = a + a \times (a \times a)$, its parsing will build the syntactic parse tree shown in figure 2.1. Suppose each token a represents a number, we define its semantic value as the value of the number. Therefore for each production we can define a semantic function that will associate the semantic values of its right hand side symbols to the semantic value of its left hand side.

For instance, if we consider the substring $a_{18}\ \ \times_{19}\ \ a_{21}$ with semantic values $v(a_{18}) = 5$ and $v(a_{21}) = 2$, the resulting semantic value for T_{15} will be 10 if the semantic function associated with production $T_0 \to T_1 \times F_2$ (where the subscripts are used to distinguish the individual symbols in the rule) is $v(T_0) = v(T_1) \times v(F_2)$.

In order to obtain the shown results, *i.e.* the syntactic parse tree decorated with the semantic values, different parsing methods have been researched and developed. In section 2.1 we give an overview of the most used ones, their performances and their limitations; in section 2.2 we describe some recent attempts at parallelizing parsing for particular languages.

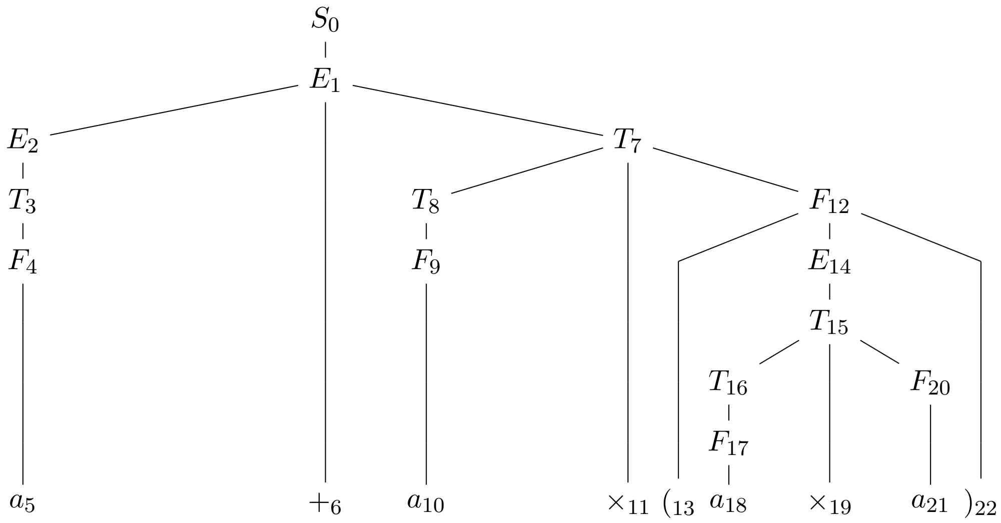


Figure 2.1: Syntactic parse tree

2.1 General parsing techniques

The presentation given in this section is part of [5]. In this section we give an overview of the most used parsing methods. The reason why we present these methods is to show how they give almost no chance to employ them in a parallel analysis of a generic string. By showing how they work this can be easily realized. First we present deterministic descent algorithms (subsection 2.1.1); after that we show deterministic ascent algorithms (subsection 2.1.2). Then we describe Earley parser algorithm that can treat any Context-free Grammar, even ambiguous and non deterministic ones (subsection 2.1.3). In order to give a short presentation here, many of the needed formal definitions are omitted and can be found in [5].

2.1.1 Deterministic descent parsing

A descent parser builds the *leftmost derivation* proceeding from the axiom towards the terminal symbols, where with leftmost derivation we mean the expansion of the leftmost nonterminal symbol. A descent analyzer expands the left hand sides of the production rules in their corresponding right hand side. It terminates either when an error is found or all the nonterminal symbols have been transformed in terminal symbols corresponding to the given string.

We present now the *LL* family used to build deterministic descent parsers. The first algorithm we see is called *LL(1)*. The LL(1) algorithm is based on the concept of *lookahead*. Since for these algorithms the grammar is expressed as a net of finite state machines, the lookahead serves to identify which edge to follow when performing a move. For each edge, its lookahead represents the first terminal symbol that could be read next. If the lookahead sets are disjoint for every outgoing distinct edge the move is unique: this is defined as the *LL(1) condition*. If the LL(1) condition holds for each state of each machine of a grammar, the grammar can be recognized by an LL(1) parser.

The automaton describing the LL(1) algorithm is defined as follows:

- x is the source string and cc is the current terminal symbol;
- the symbols of the stack are the disjoint union $Q = Q_A \cup Q_B \cup \ldots$ of the states of the machines;
- initially the stack contains only $q_{S,0}$;
- let $s \in Q_A$ be the symbol on top of the stack; the transitions are defined as follows:

  scan if the transition $s \xrightarrow{cc} s'$ is defined in machine M_A, then the current symbol is consumed and s' is pushed on the stack in place of s;

  call if the transition $s \xrightarrow{B} s'$ is defined for a nonterminal symbol B and $cc \in LA(s \xrightarrow{B} s')$ $q_{B,0}$ is pushed on the stack and M_B becomes the active machine;

  return if s is a final state of the machine M_B and $cc \in LA(s \xrightarrow{)} s$ is popped from the stack and the new state r on top of the stack is replaced by s' if $r \xrightarrow{B} s'$ is defined;

recognize if s is a final state of M_S (the machine of the axiom) and cc is the terminator symbol #, the string is accepted and the automaton terminates;

default in any other case there is an error, the string is refused and the automaton terminates.

An example of LL(1) grammar, expressed as a net of FSMs, is the simplified arithmetic expression grammar shown in figure 2.2. Notice that the grammar had to be modified since it cannot be left recursive. In fact, an LL(k) parser cannot handle ambiguous or left recursive grammars.

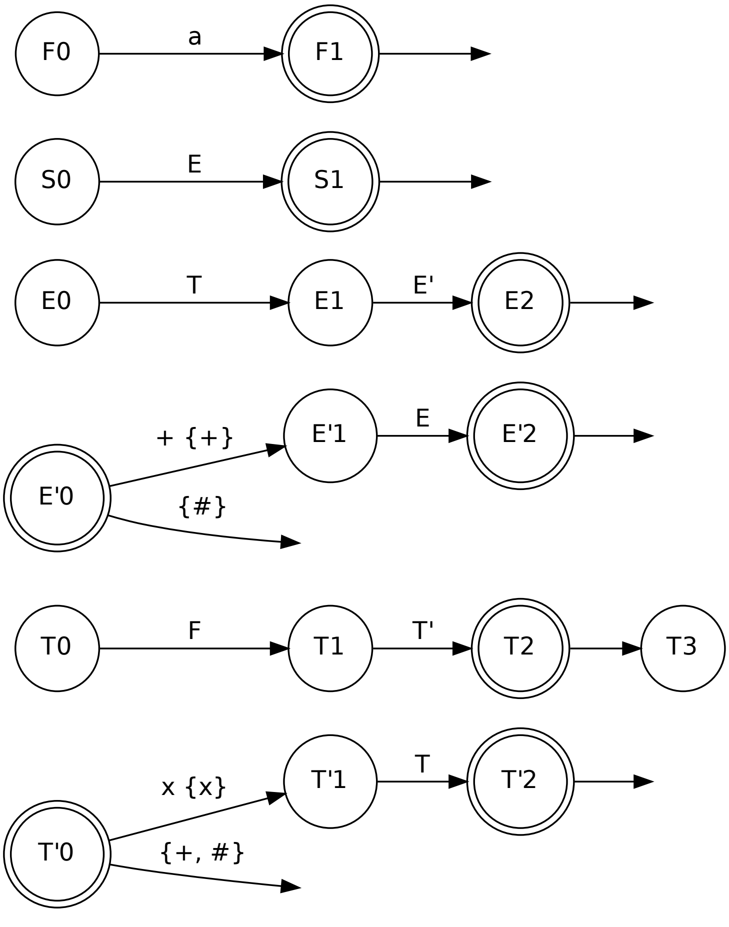
If the grammar does not respect the LL(1) condition it is possible to either try to modify the grammar and obtain an equivalent LL(1) grammar or increase the length of the lookahead, *i.e.* generate an *LL(k)* parser with $k > 1$. If the lookahead now result to be disjoint, the grammar is said to be respect the LL(k) condition. Any LL(k) parser, with $k \geq 1$, has linear complexity with regards to the length of the input string, since the automaton either reads and consumes a token or expands a nonterminal. Despite the fact that a nonterminal can be recursively expanded, this is limited by the number of nonterminal symbols present in the grammar, since there cannot be left recursive productions.

However, independently from the length k of the lookahead, not every grammar can be recognized by an LL(k) parser. Therefore in the next section we present *LR(1)*, a more powerful technique that can be used on a wider range of grammars.

2.1.2 Deterministic ascent parsing

As seen in section 2.1.1, the LL(1) method cannot be used if in a state of a machine there are conflicting lookahead sets. In general LL(k) tries

Figure 2.2: Arithmetic expressions grammar in FSM form with LA on forks.



to lengthen the lookahead needed to recognize which immediate move to choose; a different approach would try to delay the choice, and thus keep into consideration all possibilities, until it can be safely done. This is the idea on which the deterministic ascent parsing methods, or *LR* methods, are based. LR algorithms owe their name to the fact that they try to build the rightmost derivation of the input string, and they try to do so starting from the leaves of the tree and proceeding up to the root, *i.e.* when a complete right hand side is recognized, it is *reduced* into its corresponding left hand side.

In general, algorithms belonging to the LR family do not need a lookahead set, but require what is called a *pilot automaton*; intuitively, a state of the pilot automaton tracks all the possible reductions which were delayed. A state may be:

reduction if it represents only completely recognized production rules;

move if it represents only partially recognized production rules;

mixed if it represents both completely and partially recognized production rules.

If none of the states of the pilot automaton is mixed and if every reduction state represents only one rule, then the *LR(0) condition* is said to be satisfied and the language can be recognized by an *LR(0)* parser, *i.e.* an LR parser without lookahead.

Given the pilot automaton for a grammar, the pushdown automaton representing the corresponding LR(0) parser is directly obtained, and can be defined as follows:

- the symbol of the stacks are $R \cup \Sigma \cup V_N$ where $R = \mathcal{P}(Q)$, *i.e.* the powerset of Q;
- initially the stack contains only the pilot automaton initial state I_0;

- let I be the pilot automaton state on top of the stack and a the current token; the transitions are defined as follows:

  move if the transition $I \xrightarrow{a} I'$ is defined, the current token is consumed and the string aI' is pushed on the stack;

  reduction if the current pilot automaton state I_n contains the reduction state q, with $red(q) = \{B \rightarrow X_1X_2 \ldots X_n\}$, where $n \geq 0$ is the length of the right hand side the following action is preformed: the string $\beta' = X_1I_1X_2I_2 \ldots X_nI_n$, which is on top of the stack, will be deleted and replaced by BI'', where I'' is the pilot automaton state reached reading the nonterminal symbol B, *i.e.* the result of the transition $I' \xrightarrow{B} I''$.

- if the only symbol remaining on the stack is I_0 and the only remaining token is the terminator #, the string is recognized and the automaton terminates;

- if there is no possible move, the string contains an error, it is refused and the automaton terminates.

An example of LR(0) pilot automaton is shown in figure 2.4 and its grammar in the form of FSMs net is given in figure 2.3. They represent the usual arithmetic expressions grammar; the states with outgoing labelless arrows in the pilot automaton represent that a reduction is possible; it is easy to notice that there are several mixed states (*e.g.* $I1, I2, I9$) and therefore the grammar is not LR(0). In fact, if a language is LR(0) then it must not have prefixes, *i.e.* if a string belongs to the language then none of its prefixes can.

In order to cope with this problem, in a similar fashion to what has been done in LL, it is possible to introduce the idea of lookahead for LR as well. This results in the *LR(k)* methods, where $k \geq 1$, which combine the lookahead of LL(k) methods with the delay of the choices of LR(0). This

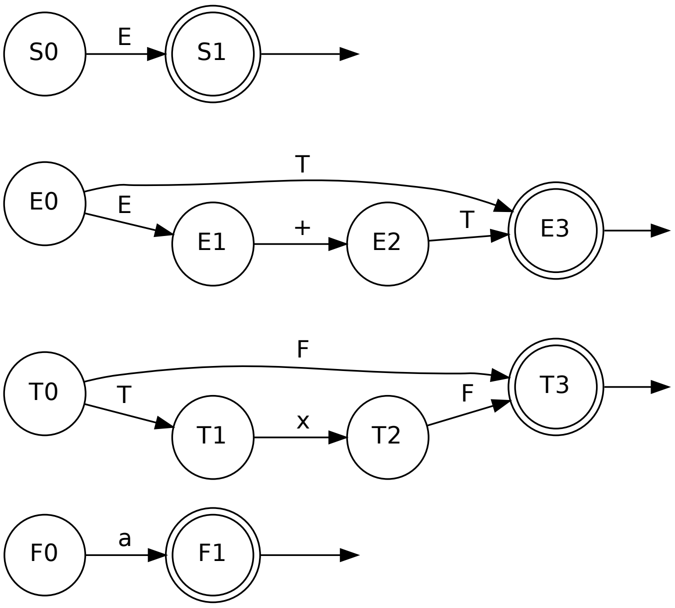


Figure 2.3: Left recursive arithmetic expressions grammar in FSM form.

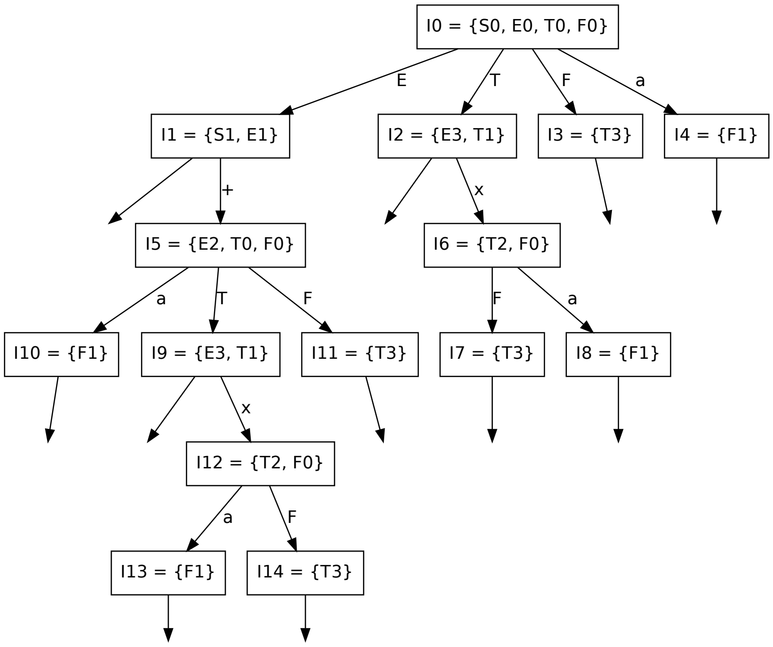


Figure 2.4: Left recursive arithmetic expressions grammar LR(0) pilot automaton.

introduction modifies only the construction of the pilot automaton of the grammar.

Similarly to the LR(0) condition, an *LR(1)* condition can be defined for a state in the pilot automaton as follows:

- no reduction-move conflicts, *i.e.* if a state is mixed then the lookahead sets of its reduction candidates and the sets of the labels of the outgoing edges of its move candidates must be disjoint;
- no reduction-reduction conflicts, *i.e.* if a state is a reduction state and there are multiple reduction candidates, their lookahead sets must be disjoint.

The LR(1) condition holds for a grammar if it holds for each state of its pilot automaton.

For instance, the pilot automaton for the grammar defined in figure 2.3 is shown in figure 2.5: it can be noted that there exist no reduction-move conflicts nor reduction-reduction conflicts, and thus the grammar respects the LR(1) condition and its language can be recognized by an LR(1) parser.

An intermediate approach, that was introduced when low memory capacities prohibited the use of the LR methods due to their large pilot automata, is the *LALR* method. This method is worth mentioning since it is still fairly used in modern parser generators such as *Bison* [8]. The LALR(1) method tries to simplify the LR(1) automaton by joining together different states which are indistinguishable in the LR(0) pilot automaton, *i.e.* those that differ only in the lookahead. When joining two states, the lookahead sets are joined as well for the candidates that correspond to the same state in the FSMs net. The *LALR(1) conditions* are the same as the LR(1) conditions.

As for LL(k), it is possible to lengthen the lookahead and generate an *LR(k)* parser, with $k \geq 1$. However, the family of languages generated by a grammar LR(k), with $k > 1$, is the same as the family of languages generated

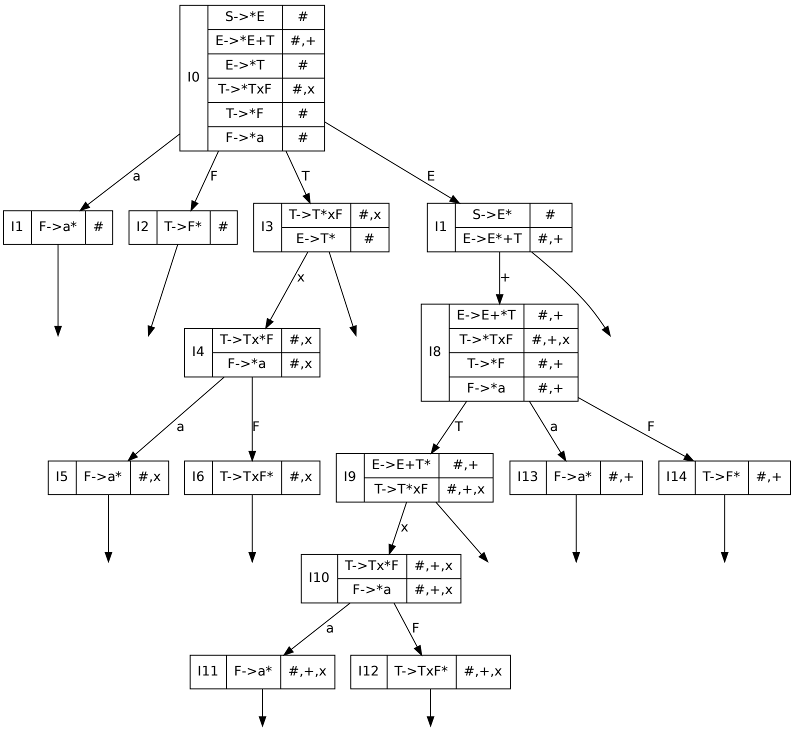


Figure 2.5: Left recursive arithmetic expressions grammar LR(1) pilot automaton.

by a grammar LR(1), which in turn coincides with the family *DCF*, *i.e.* the family of deterministic context-free languages. It is fairly obvious that not every grammar which generates a deterministic language is LR(1); but an equivalent LR(1) grammar can be generated.

As for LL(k), an LR(1) parser has linear complexity with regard to the length of the input string. The explanation is analogous to the one given for the LL(k) parser: at every iteration, the automaton either consumes one input token or performs a reduction. In the latter case, the number of reductions that can be performed is limited by the number of nonterminal symbols in the grammar, which is constant.

However, not every language can be analyzed with an LR parser; thus, we introduce in the next section the *Earley* method to parse every Context-free Grammar, even non deterministic ones.

2.1.3 Earley algorithm

The *Earley algorithm* owes its idea to LR, but, instead of the deterministic stack model, it uses a vector of sets, which represents efficiently many stacks with shared parts. In this way it simulates a non deterministic pushdown automaton without having its exponential complexity. In particular, instead of using a pilot automaton, the Earley algorithm uses a vector E as long as the input string, where it stores every state in which the recursive FSMs net may be found. Every element of E is defined as a set of pairs

$$< s, p >$$

where s is the state in the FSM and p is the position inside the string x where the recognition of the current production rule began.

The algorithm is divided into two parts. There is an initialization phase:

- the first position of the vector $E[0]$ is initialized with the pair $< q_{S,0}, 0 >$, where $q_{S,0}$ is the starting state of the FSM of the axiom;

- every other position of the vector is initialized as the empty set.

Then the algorithm iterates over each input token and performs the following actions:

prediction for each pair in $E[i]$ in the form $< q \equiv A \rightarrow \alpha \bullet B\gamma, j >$, where the transition $q \xrightarrow{B} s$ is defined, the pair $< r, i >$ is added to $E[i]$, where r is the initial state of the FSM M_B;

completion for each pair in $E[i]$ in the form $< q \equiv A \rightarrow \alpha\bullet, j >$, for each pair in $E[j]$ in the form $< r \equiv B \rightarrow \beta \bullet A\gamma, k >$ such that the transition $r \xrightarrow{A} s$ is defined, the pair $< s \equiv B \rightarrow \beta A \bullet \gamma, k >$ is added to $E[i]$;

scan for each pair in $E[i]$ in the form $< q \equiv A \rightarrow \alpha \bullet a\gamma, j >$, if $a = x_{i+1}$ then the pair $< r \equiv A \rightarrow \alpha a \bullet \gamma, j >$ is added to $E[i+1]$, where the transition $q \xrightarrow{a} r$ is defined.

The algorithm terminates when the construction of the set $E[n]$, where n is the length of x, is finished; it reports an error if it could not find a pair to add to $E[i+1]$ with $i < n$. If the final set contains at least one pair in the form $< q \equiv S \rightarrow \alpha\bullet, 0 >$, where q is the final state of the FSM M_S, the string is accepted.

The Earley algorithm can be used to recognized non deterministic languages, and therefore is the most general parsing algorithm. Its complexity with regards to a string of length n is at worst $\mathcal{O}(n^3)$, however, in practice it is much faster: $\mathcal{O}(n^2)$ for each unambiguous grammar and even less for deterministic grammars.

An example of the Earley algorithm is shown in figure 2.6. The analyzed string is $a + a \times a$, generated from the usual left recursive arithmetic expressions grammar. The Earley algorithm correctly builds the vector and recognizes the string, since the pair $< S \rightarrow E\bullet, 0 >$ is present in $E[5]$ after having consumed the entire input string.

E[0]	S->*E	0
	E->*E+T	0
	E->*T	0
	T->*TxF	0
	T->*F	0
	F->*a	0

E[1]	F->a*	0
	T->F*	0
	T->T*xF	0
	E->T*	0
	E->E*+T	0
	S->E*	0

E[2]	E->E+*T	0
	T->*TxF	2
	T->*F	2
	F->*a	2

E[3]	F->a*	2
	T->F*	2
	T->T*xF	2
	E->E+T*	0
	E->E*+T	0
	S->E*	0

E[4]	T->Tx*F	2
	F->*a	4

E[5]	F->a*	4
	T->TxF*	2
	E->E+T*	0
	T->T*xF	2
	S->E*	0
	E->E*+T	0

Figure 2.6: Earley vector built analyzing string $a + a \times a$.

As we have seen, all of the presented methods have the same basic problem for a parallelized approach, *i.e.* in order to recognize a substring they need to analyze all the previous tokens and therefore a simple way to use them in a parallel algorithm has not been found. This limitation falls using a parser which employs FGs since, as we will see in chapter 3, the recognition of an entire substring depends only on its previous and following token, *i.e.* the context.

2.2 Parallel parsing techniques

In this section we present some approaches that have been proposed to exploit parallel computing inside of the parsing process. A first real approach has been proposed in [9], where a parallel extension of the LR algorithm, the so called *piecewise LR*, is presented. What this algorithm does is divide the input at arbitrary points and assign a PLR parser to each of the obtained substrings. These proceed as normal LR parsers until they find a conflict; at this point the parser asks the previous parser to synchronize their stacks: the blocked one will give its stack to the previous one, that may have enough context to proceed, void its stack and continues with the rest of its input. This process is repeated until only the leftmost parser has input, point where it will finish parsing the input as a normal (sequential) LR parser. In the article there is no experimental evaluation of the algorithm's performances and there has been no further development or evaluation of this algorithm.

After this attempt, the attention of research has shifted to the analysis of the theoretical advantages of parallel parsing. A first attempt to determine the maximum possible speedup attainable with parallel parsing is proposed in [10]. In this paper a mathematical model is proposed for parallel parsing and, through analytical considerations, an upper bound for speedup is obtained. Following works [11] [12] are more practical; in these papers different possible models of parallel parsing are presented and evaluated. In [11] a simulator of such a model is proposed and through experiments the performances of the model are evaluated. In [12] the same model and a variant are estimated through mathematical analysis. In both of these papers good results in term of speedups are shown; however, in none of the cited papers there is any real implementation and therefore the performance evaluation does not have to deal with the limitations coming from real multi processor and multi core platforms; moreover, rather strong and unrealistic assumptions with respect to these architectures are made, making, in our opinion, the results far less meaningful when contextualized in relation to modern machines.

After these papers there have been no other attempts at developing general parallel parsing algorithms. Some ad-hoc parsers have, however, been developed. They are all targeted at parsing *XML*; the reason behind this is that XML is a widely accepted standard with very wide and different application fields. Moreover XML parsing is known to become a bottleneck in applications that use it extensively, due to its verbosity and text-based nature, and the fact that it includes metadata. This has led to several approaches to improve its parsing performance, including parallel approaches.

In our research all the methods at first split the input string (or document) into several substrings, one for each available core. The approaches they take in order to perform parsing on each of the substrings are two: the first one performs a preparsing phase in order to discover the general structure of the underlying tree, while the second one skips the preparsing phase, builds several incomplete subtrees and tries to link them together during a post processing phase.

2.2.1 Preparsing approach

The first approach is described in [13]. It fist makes an initial pass to determine the logical tree structure of the XML document. This preparsing phase is used to provide enough context so that each substring can be parsed starting from an unambiguous state. The preparsing phase is fast since it is not a complete parsing of the string: in fact, leaf nodes, such as attribute information items, comment information items and character information items, can be ignored; moreover element tag names are ignored since they do not affect the topology of the tree. It is to be noted that, despite the fact that it is fast, the preparsing phase is still the bottleneck of this approach, since it is sequential and it has to be done on the whole document. The preparsing phase produces what is called the *skeleton tree.*

Once the skeleton tree has been completed, it has to be partitioned and each element of the partition must be assigned to a thread. This is done either statically, *i.e.* the tree is split in subtrees that are analyzed by different threads, or dynamically, *i.e.* the tree is traversed starting from the root and different branches are analyzed by different threads. The difference between the two approaches lies in the fact that the dynamic partitioning needs synchronization among threads while the static one does not, but needs to be executed sequentially before the parsing. The authors of this approach further developed techniques to partition the skeleton tree. In particular, they developed methods to balance the load each threads has to process. This work can be found in [14].

2.2.2 Speculative parsing approach

The second approach is described in [15]. This parser is based on the *divide et impera* principle, which is to split the string at (almost) random points and analyze each resulting substring independently using what they they call a *speculative parser*. The output of the speculative parsers is a forest of incomplete trees, which are then linked together during a *post processing* phase.

In [15], each substring starts with an open bracket: this guarantees that each chunk can be treated as a new XML document, even though it may not be well-formed. Then the speculative parser behaves as a traditional XML parser, but has to handle the fact that it may find errors in its substring which are not actually errors, but are just the result of the division. When the speculative parsers are done, the trees must be linked together: this is done in the post processing phase. During this phase all the errors found during the speculative parsing phase are evaluated and, if possible, treated in such a way that all the subtrees can be joined in the tree that corresponds to the original document.

In [16] a similar approach is proposed. The only difference is that, during the parsing phase, a preorder numbering is done, which consists of assigning to each node a unique identifier. These identifiers are then used during the subsequent phase and permit a faster linking of the subtrees.

As we will see in chapter 4, the method we developed is similar to the second approach, in which it consists of no preparsing phase, splitting the string at random positions, parsing each one separately, and then treating the subtrees in successive phases. We will see that even if our approach is more general, *i.e.* it was not thought to be applied only on a particular language, requires the grammar to be expressed in a particular way: we will see how this can be done in chapter 3. This is, in our opinion, acceptable, since there is only more effort required by the designer of the parser.

Chapter 3

Floyd grammars

In this chapter we present the theoretical roots that led to the development of a parallel parser. In particular, we introduce and describe *Floyd Grammars* (FG), which is the fundamental formalism upon which the entire work is based, explaining both its formal properties and how these properties can be used in order to develop a parallel parsing algorithm. The rest of the chapter is structured as follows: in section 3.1 we provide the formal definitions needed to explain what a FG is; in section 3.2 we describe how FGs relate to other well known and used formal grammar families, and therefore why they really are a viable option in order to express commonly used formal languages; finally in section 3.3 we discuss some practical considerations and aspects needed to be taken into account when expressing FGs for the developed parser. This chapter, its definitions and theorems are based on [1] and [6].

3.1 Definition and properties

To define FGs we start from a generic *Context-Free* grammar $G = < V_N, \Sigma, P, S >$ where the symbols have the usual meaning already seen in chapter 2. A production is said to be in *operator form* if its right hand side has no adjacent nonterminals. Consequently, a grammar consisting only of productions in operator form is said to be an *operator grammar* (OG). If G is an operator grammar, we can define the concepts of *left* and *right terminal sets* for a nonterminal A as:

$$\mathcal{L}_G(A) = \{a \in \Sigma \ \mid \ A \overset{*}{\Rightarrow} Ba\alpha\}$$
$$\mathcal{R}_G(A) = \{a \in \Sigma \ \mid \ A \overset{*}{\Rightarrow} \alpha aB\}$$

where $B \in V_N \cup \{\varepsilon\}$ and $\alpha \in (V_N \cup \Sigma)^*$.

The notions of left and right terminal sets allow us to define the concept of *operator precedence* (OP) between two terminal symbols. Given an OG G, $a, b \in \Sigma$, $\alpha, \beta \in (V_N \cup \Sigma)^*$ and $A \in V_N$ we can define three relations:

$$\begin{aligned}
\text{equal precedence: } & a \doteq b \iff \exists A \to \alpha aBb\beta,\ B \in V_N \cup \{\varepsilon\} \\
\text{takes precedence: } & a \gtrdot b \iff \exists A \to \alpha Db\beta \text{ and } a \in \mathcal{R}_G(D),\ D \in V_N \\
\text{yields precedence: } & a \lessdot b \iff \exists A \to \alpha aD\beta \text{ and } b \in \mathcal{L}_G(D),\ D \in V_N
\end{aligned}$$

Following these definitions, it is possible for an OG G to define the precedence relations between each terminal pair. The *operator precedence matrix* (OPM) is therefore the function that associates every possible ordered terminal symbol pair with the precedence relations holding among them.

Given this, the definition of a Floyd Grammar is straightforward: G is a Floyd Grammar if and only if $M = OPM(G)$ is a *conflict-free matrix*, *i.e.* $\forall a, b, \ \ |M_{ab}| \leq 1$. Two OPMs are said to be *compatible* if their union is conflict-free. Similarly, two FGs are said to be *precedence-compatible* if their OPMs are compatible.

An example of FG has already been given in chapter 2, *i.e.* the grammar used to express parenthesized arithmetic expressions. In fact it is immediate to verify that all the productions are in operator form and that the left and

	a	(	)	+	×	#
a			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\gtrdot$
)			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
×	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
#	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

Table 3.1: Precedence matrix for the parenthesized arithmetic expressions grammar.

right terminal sets are as follows:

$$\begin{array}{ll}
\mathcal{L}_G(S) = \{+, \times, a, (\} & \mathcal{R}_G(S) = \{+, \times, a,)\} \\
\mathcal{L}_G(E) = \{+, \times, a, (\} & \mathcal{R}_G(E) = \{+, \times, a,)\} \\
\mathcal{L}_G(T) = \{\times, (, a\} & \mathcal{R}_G(T) = \{\times, a,)\} \\
\mathcal{L}_G(F) = \{a, (\} & \mathcal{R}_G(F) = \{a,)\}
\end{array}$$

from which we can obtain the OPM shown in 3.1. As it can be evinced from the table, each pair of terminals has at most one operator precedence relation and therefore the matrix is conflict-free; thus the grammar is a Floyd Grammar.

3.1.1 Properties

We will now show some formal properties which FLs (Floyd Languages) enjoy; these properties are shown here because they represent the basis upon which the entire work has roots. In fact, without many of them parallel parsing would not be possible. Moreover, some of these properties have been used in a first attempt to create a parallel parser (which has now been abandoned with respect to the objectives of this thesis), which is explained in appendix A.

In [1], it is shown how FLs enjoy closure properties with respect to common formal languages operations. The properties that are relevant for this thesis are the closure under prefix, suffix, concatenation and Kleene star. In short, this means that the prefix, suffix and applying the Kleene star operator to a FL produce a possibly different language which can still be generated by a precedence-compatible FG. As for concatenation, it means that two FLs concatenated produce a third FL. As a corollary, FLs are closed under Boolean and reversal operations as well, but these properties are of little interest with respect to our work.

The properties said to be relevant are used in the approach presented in appendix A. In fact, the basic idea of the approach is to define the grammars of the prefixes, suffixes and infixes of a given grammar, and use them to parse the different parts of an input string, resulting in several parse trees, which need to be linked together to reconstruct the parse tree of the original string.

Another interesting property of FLs, especially for the purpose of this thesis, is the *locality principle*, which means that given a string of a FL, different portions of it can be correctly parsed independently from other parts, considering only their previous and following contexts. In particular, a substring can be correctly parsed if the precedence relations with its previous and following tokens are known, and this is always possible given that the string belongs to the language generated by the grammar, since for FGs the precedence relation among two tokens is always univocally determined. This property stands as the backbone of the entire work; intuitively, since the recognition of a rule depends only on the previous and following tokens and not on the entire prefix, multiple parsers can work on different parts of the same input and reduce several subtrees at the same time. A subtree is defined as the partial result of a parsing process on a substring; in fact, since no parser can reach the axiom given that they do not work on the entire string, they just analyze a part of it, creating an incomplete parse

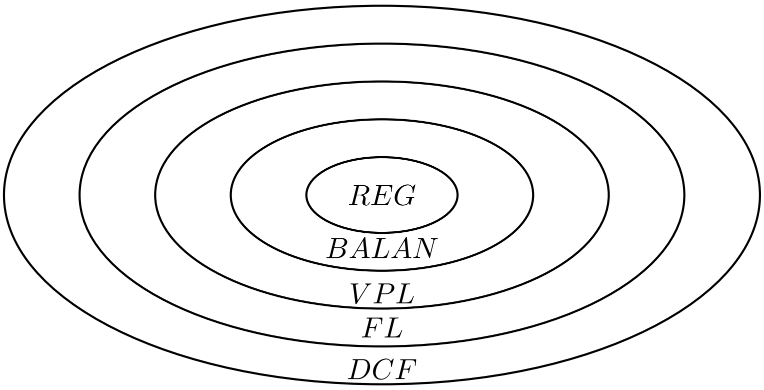


Figure 3.1: Relations between formal language families.

tree. However, thanks to the formal properties of FGs, these subtrees are valid parts of the tree associated with the entire string, and thus only need to be joined together in subsequent phases. At this point it can be guessed how a parallel parser can be built using this approach: as we will see in chapter 4, it consists of multiple phases which analyze the input string and the subtrees already produced in previous phases.

3.2 Relationship with other grammar families

In this section we give an overview of how FLs relate with other formal languages generated by known families. The purpose of this section is to show that FGs are a viable option in order to express commonly used formal languages, in the sense that they retain interesting properties that less powerful families have, while still being able to express a wider set of practical languages.

A depiction of the relations among the presented language families is shown in figure 3.1. The first relation to note is that regular languages are a special case of FLs. In fact every regular language can be generated by a *right-linear grammar*, *i.e.* a grammar whose rules are in the form $A \rightarrow aB,\ a \in \Sigma,\ b \in V_N$, therefore the right hand side length is bounded by 2 and the only precedence relations are $\lessdot$.

Since regular languages cannot be used to express more common practical languages, we show how FLs include other more powerful families. In particular, in [1] it is shown how FLs include the two families of *Visibly Pushdown languages* (VPL) and *balanced languages* (BALAN). For the purpose of this thesis the second result is especially relevant since BALAN includes languages such as XML and HTML, which are heavily employed in real world applications, and, as we have mentioned in section 2.2, these languages could gain great benefits from a parallel analysis.

The last thing to notice is that FLs are strictly included in the family of *Deterministic Context-Free languages* (DCF). A simple counterexample showing a DCF language which is not a FL is

$$L = \{a^n b a^n | n \geq 0\}$$

This language, no matter how its grammar is expressed, will produce a conflict in the precedence relation in the pair (a, a).

The DCF family comprehends all the commonly used computer languages, since as we have seen in section 2.1.2 LR(1) parsers recognize DCF languages and those are very common parsers. This implies that FGs and therefore FL parsers cannot be used to express and analyze every language that DCF grammars and LR parsers do. Nonetheless FGs are powerful enough to express real world languages. In fact, as we will show in chapter 6, the realized parallel parser has been successfully used to analyze JSON strings, which is a well known and widely used subset of the JavaScript language.

3.3 Practical considerations

In this section we will discuss some of the more practical aspects involving the use of FGs in a parsing process, in particular we will see how to express a grammar in order to be used with our parser and how these limitations have influenced its development. These limitations are not theoretical in a sense that they do not restrict the family of FGs which can be effectively employed, but have only been introduced in order to simplify the development of the parser and therefore every FG can be rewritten in the required form. However, these transformations often enlarge the number of rules and nonterminal symbols in the grammar and can make it less intuitive to read.

Aside from the usual limitations deriving from the definition of a Context-Free grammar and in particular a FG (*i.e.* all production rules in operator form), there are two situations which can cause problems when building a parser. The first one is the presence of repeated right hand sides, *i.e.* the existence of at least two rules in the form $A \to \alpha$ and $B \to \alpha$. This causes difficulties during the recognition of the correct reduction to apply to a substring representing the right hand side: in fact, as we will see in section 4.1.1, we developed a *reduction tree* that serves to correctly identify a reduction while exploiting the locality principle, *i.e.* employing only the right hand side. In presence of repeated right hand sides, the reduction tree cannot decide which reduction is the correct one. This problem could be solved by developing a *speculative reduction tree*, which could keep trace of the multiple reductions to be performed and delay the final decision.

A second problem is the presence of *immediate rewrite rules*, *i.e.* rules in the form $A \to B,\ B \in V_N$. The presence of such rules creates the necessity for the parser to be able to discern whether to perform another immediate reduction after a normal reduction or not. Since by exploiting the locality principle, this cannot be done just by looking at the right hand side, as we will see in section 4.1.2, we decided to employ *reduction sets*, that enable the parser to delay the decision while allowing it to perform the normal reduction. This solves the problem and allows the use of immediate rewrite rules while expressing the grammar.

Chapter 4

Parallel parsing algorithm

As seen in section 2.1, in the current state of the art general parsing techniques are inherently sequential. Using these parsing methods makes parallelization very difficult, due to the following reasons:

- traditional parsers require knowledge of the whole parsing history, *i.e.* to parse an infix of a string, the state reached by the parser after having analyzed the entire prefix is needed. The string cannot therefore be split into different substrings which are then parsed independently from each other;
- in general, the properties required to parallelize parsing do not hold for languages that are recognized by traditional parsers.

Since traditional parsing schemes are not a valid option in order to achieve parallelization a new algorithm was developed exploiting the properties of FGs.

In this chapter we present the parallel parsing algorithm we developed, *i.e.* a multi-phase parser operating on different parts of the input strings. In particular we will show a basic operator precedence parser (section 4.1, which performs parsing of strings of a language expressed by a FG; then we move towards a real parallel parser by modifying the presented algorithm in order to work on substrings of an input string (section 4.2; finally we adapt the algorithm to work in a multiple subsequent phases environment (section 4.3) and we describe the final parallel operator precedence parser (section 4.4).

4.1 Naive operator precedence parser

A first approach to a parallelizable parser is shown in algorithm 1. It is based on [6, 17] and employs the concept of *handle*, *i.e.* a series of tokens with precedence relations in a form such as

$$\lessdot \doteq \cdots \doteq \gtrdot$$

which identify a right hand side of a grammar production and may be reduced independently from the rest of the string. This is a consequence of the *locality principle* of FG described in chapter 3. The algorithm analyzes the string one token at a time, by calling the lexer which is implemented by a third-party library, and sets its precedence relation with the previous token; based on the precedence relation it then either goes on to the next token (precedence relation is $\lessdot$ or $\doteq$) or tries to perform a reduction (precedence relation is $\gtrdot$). The reduction will fail if there is an error in the string. This proceeds until either an error is found or the string is completely recognized.

The algorithm, during the parsing process, builds a stack, which holds information about the analyzed tokens and is needed to correctly identify the elements of a handle. A stack element is a tuple defined as

$$< T, p, v >$$

where $T \subset V_N \cup \Sigma$ is the token list, which is composed of either multiple nonterminal symbols or one terminal symbol, $p \in \{\lessdot, \doteq, \gtrdot\}$ is the precedence relation to the next token, and v is the subtree associated with the parsing of the token. When analyzing a new token, in order to set the precedence, the algorithm goes back to the first terminal symbol from the top of the stack, finds its precedence relation with the new token and then sets it.

When a $\lessdot$ or $\doteq$ precedence relation is found, the algorithm simply pushes the new terminal symbol on the stack and proceeds to analyze the next token.

When a $\gtrdot$ precedence relation is found, the algorithm proceeds (algorithm 2) as follows:

- it iterates on the stack starting from the top until an error or a $\lessdot$ relation is found;
- for each stack element it verifies if there exists a rhs which agrees with the current element and the part of handle that has been analyzed up to now. This is done by navigating a tree called *reduction tree*, which is described in section 4.1.1;
- if the current precedence relation is $\lessdot$ and only one possible rhs which agrees with the handle remains, the reduction is found: all the elements of the handle are removed from the stack and replaced by a new stack element which represents the reduced nonterminal symbol;
- else an error has been found and the algorithm terminates.

An example of parsing using the operator precedence parser is shown in figure 4.1. The string $a + (a \times a)$ is expressed with the parenthesized arithmetic expressions grammar with precedence matrix shown in table 3.1 on page 26.

In order to complete the overview of the naive algorithm the tree used to perform reductions and the mechanism used to create the reduced tokens need to be presented. This is done respectively in sections 4.1.1 and 4.1.2. Since this algorithm is not suitable to be used in a parallel context we will later see (sections 4.2 and 4.3) how it has to be modified.

	Stack	Rest of input
(1)	$\#$	a+(a×a)#
(2)	$\# \boxed{\lessdot\ \mathrm{a} \gtrdot} +$	(a×a)#
(3)	$\# \overset{\lessdot}{E} + \lessdot (\boxed{\lessdot\ \mathrm{a} \gtrdot} \times$	)#
(4)	$\# \overset{\lessdot}{E} + \lessdot (\overset{\lessdot}{T} \times \boxed{\lessdot\ \mathrm{a} \gtrdot})$	#
(5)	$\# \overset{\lessdot}{E} + \lessdot (\boxed{\overset{\lessdot}{T} \times \overset{\gtrdot}{F}})$	#
(6)	$\# \overset{\lessdot}{E} + \boxed{\lessdot (\overset{\doteq}{T}) \gtrdot} \#$	
(7)	$\# \boxed{\overset{\lessdot}{E} + \overset{\gtrdot}{T}} \#$	
(8)	$\# \overset{\doteq}{S} \#$	

Figure 4.1: Example of OPP applied to a string.

Algorithm 1 Naive Operator Precedence Parser

```
token := first_token
init_stack
while there are tokens do
  prec := get_precedence_relation(token, stack)
  if prec = ‘$\lessdot$’ $\vee$ prec = ‘$\doteq$’ then
    push(stack, token)
  else if prec = ‘$\gtrdot$’ then
    handle := identify_handle
    if $\exists$ handle then
      if reduction is possible then
        perform_reduction(handle)
        if axiom is reached then
          return 0
        end if
      else
        string does not belong to language
        return 1
      end if
    else
      string does not belong to language
      return 1
    end if
  end if
  get_next_token
end while
return 1
```

Algorithm 2 Handle identification

```
cur_item := last(stack)
handle := $\emptyset$
while cur_item.p $\neq \lessdot \wedge$ cur_item $\in$ stack $\wedge \neg$ error do
  if (rhs which agree with cur_item and handle) $\neq \emptyset$ then
    handle := handle $\cup$ cur_item
  else
    error := true
  end if
  cur_item := prev(stack)
end while
```

4.1.1 Reduction tree

We saw that when a $\gtrdot$ precedence relation is found, the algorithm tries to perform a reduction. This is done with the help of a reduction tree, which can be defined as an usual tree (directed acyclic graph). The basic idea of the reduction tree is similar to the *suffix trees* often employed in dictionaries or compression algorithms. A complete path on the tree, from the root to a leaf, stands for a rhs of a production read in inverse order, where the nodes of the tree represent the position inside the rhs, and the edges express the reading of a token from right to left. For a given grammar G, its reduction tree represents all and only its production rules right hand sides read in reverse order.

An example of the reduction tree used for the arithmetic expressions grammar is shown in figure 4.2. Take, for instance, the production

$$E \rightarrow E + T$$

we can see that, starting from the root node, one can follow the series of edges $(T, +, E)$ until the leaf node, which represents that the rule is completely recognized and therefore a reduction can be correctly performed.

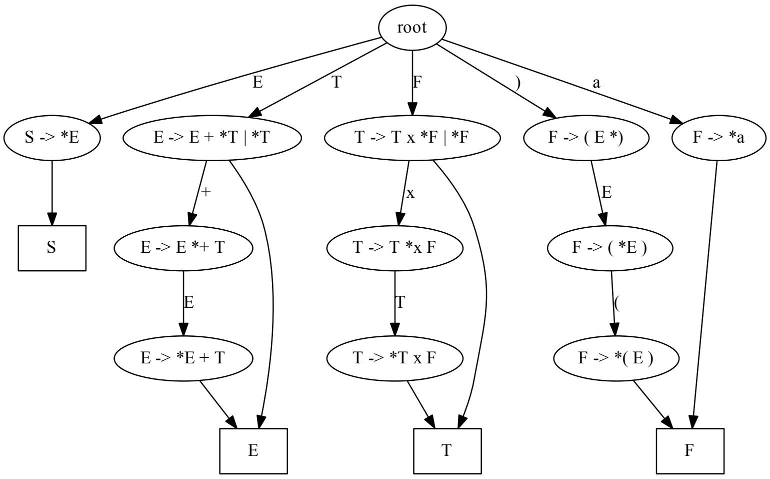


Figure 4.2: Reduction tree used for the arithmetic expressions grammar.

One last thing to notice is that the building of the reduction tree is done offline, *i.e.* before the parsing phase and therefore is not a burden on the parsing process. A detailed description of the algorithm employed to build a reduction tree starting from the definition of the grammar is given in appendix B.1.

4.1.2 Usage of reduction sets

When a reduction, *i.e.* a right hand side, has been correctly identified using the reduction tree, it has to be performed. This corresponds to translating the right hand side into the respective nonterminal symbol. Due to the fact that, as mentioned in section 3.3, there may be immediate rewrite rules in the grammar, these have to be handled correctly. This is done using reduction sets; intuitively, these represent for any nonterminal symbol A all the nonterminal symbols that can be immediately rewritten into A.

Therefore, the reduction sets are used to create the new element which replaces the handle on the stack following a reduction. In fact, the new element, which follows the same structure of a stack element defined in section 4.1, is composed as follows:

T is a list of nonterminal symbols. If L is the lhs of the recognized production and its reduction set is $R(L)$, then $T = \{L\} \cup R(L)$;

p corresponds, as usual, to the precedence relation, but, since the result of a reduction is a nonterminal symbol, the precedence relationship is undefined;

v is the usual parse subtree, computed by linking together the different subtrees associated with the tokens of the handle during the reduction process using the semantic function corresponding to the production.

In the following example we show both why reduction sets are necessary in presence of immediate rewrite rules and how a reduction set is formed. Consider the rules

$$E \rightarrow E + T \mid T$$

$$T \rightarrow a$$

If we do not include both T and E in the list of possible nonterminal symbols when we reduce a later on we may incur in an error. This can happen, for example, when the string is $a + a$, we reduced the first a to only T, and cannot find a rule to reduce $T + T$. However, reducing the first a as described above we obtain $\{E, T\}$, which guarantees a correct parse in the following reduction.

An in-depth description of the algorithm that computes the reduction sets is given in appendix B.2.

4.2 Substring operator precedence parser

The algorithm, as described in the previous section and considering a possible parallel context, has to work with relaxed assumptions, *i.e.* it needs to analyze a substring of the language, which may be either a prefix, infix or suffix of the global string. This leads to the following consequences:

- the context may differ from the terminator symbol couple;
- parsing of the substring may finish without recognizing the axiom yet without error;
- the substring may begin or end with incomplete handles, that cannot be reduced;

For instance, if we consider the usual arithmetic expressions grammar and the string

$$a + a \times (a \underbrace{\times (a + a) + a \times a + (a \times}_{\text{sub}} a)) + a \times a$$

if the parser analyzes substring *sub* the context is (a, a) and it is obvious that *sub* will not be reduced to the axiom. Yet the parser should not report an error code: in fact the string considered in its entirety does belong to the language, even if *sub* does not. Moreover the first '×' in *sub* will not be reduced since its handle is incomplete. The algorithm should skip it and try to reduce the rest of the substring.

It is therefore evident that algorithm 1 on page 35 has to be modified to cope with these new assumptions. A scheme of the modified algorithm is shown in algorithm 3. The modifications done can be summarized as follows:

row 2 The initialization of the stack needs to push at the bottom the previous context of the substring; this is done in order to guarantee a correct identification of the precedence relation of the incoming terminal tokens;

row 20 This is to cope with the presence of incomplete handles at the beginning of the substring; in the case of an incomplete handle, the algorithm does not report an error but moves on with the parsing;

row 23 When considering the next token, if the end of the substring is reached, the algorithm needs to consider the following context;

row 25 If the algorithm did not reach the axiom without encountering any error it should return only the forest of recognized subtrees;

With these modifications the algorithm is suitable to be used on a generic substring of the input string.

4.3 Partially parsed substrings

What we have described up to now is a parser that can analyze a generic substring of a language and build all its complete parse subtrees, even without reaching the axiom. This was the first step towards a parallel parsing algorithm. One further modification is needed in order for the parser to work in a parallel context, *i.e.* a parser that can work on partially recognized substrings. This is because the parser works on multiple subsequent phases with decreasing degree of parallelism and phases following the first need to be able to manage nonterminals produced by the previous phases.

In order to do so, we have to modify algorithm 3 on page 41 as follows: it is necessary to handle incoming nonterminal symbols. When one is found, the algorithm will create a new stack element that will be pushed on top. The list of tokens is composed of the reduction set of the given nonterminal symbol; the precedence relation is undefined because there is no precedence relation between a nonterminal symbol and any other token; the parse subtree is the result of a previous phase;

This is the only modification needed and the rest of the algorithm does not have to be modified.

Algorithm 3 Substring Operator Precedence Parser

```
token := first_token
init_stack
while there are tokens do
  prec := get_precedence_relation(token, stack)
  if prec = '$\lessdot$' $\vee$ prec = '$\doteq$' then
    push(stack, token)
  else if prec = '$\gtrdot$' then
    handle := identify_handle
    if $\exists$ handle then
      if reduction is possible then
        perform_reduction(handle)
        if axiom is reached then
          return 0
        end if
      else
        string does not belong to language
        return 2
      end if
    else
      push(stack, token)
    end if
  end if
  get_next_token
end while
return 1
```

4.4 Parallel operator precedence parser

Up to now we have described how a sequential operator precedence parser works. Then we saw how to adapt such a parser in order to analyze substrings and already partially parsed substrings. In this section we present how we decided to manage parsing of a string by dividing it into several substrings which are analyzed by different operator precedence parsers.

As shown in algorithm 4, there is an initialization phase, during which common data structures (*i.e.* the reduction tree, the precedence matrix and the copy rules) and the lexer are generated; the string is read and divided into n substrings that will be mapped on a different thread each. In section 2.2 we saw that some parsers execute a preparsing phase before the actual parsing in order to find suitable cutoff points; this approach though has very limited benefits when using FGs since their locality principle makes so that recognizing one handle only requires the knowledge of the precedence relation with the previous token. Therefore analyzing the entire string in order to find a suitable cutoff point is not justifiable. Thus, when designing the algorithm, our choice was to cut the string into n balanced substrings, *i.e.* each string has the same length. This has been done in order to try to balance the job that each thread has to do; since without preparsing phase there is no idea of how the parse tree of the input string is formed, it is not possible to actually balance the job of each parser, and therefore balancing the length of the input of each thread is the only possible way. A different and possibly more efficient approach involves analyzing a limited neighborhood of the already chosen cutoff points and using some heuristics on the precedence relation to find better points; this will be explained more in depth in chapter 5.

After having divided the string, the algorithm iterates until either it is completely parsed or an error is found. During each iteration, the algorithm determines the context in which every substring is to be analyzed. The context corresponds to a couple of terminal symbols: one is the first terminal

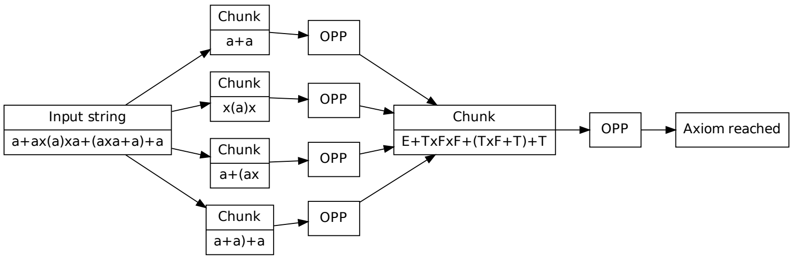


Figure 4.3: Sequential recombination parsing scheme.

preceding the beginning of the substring and the other is the first terminal following its end. If the substring does not have a prefix (*e.g.* the first substring) its preceding context is the terminator symbol #, and this applies dually if the substring has no suffix. The context is needed to determine the precedence relations of the first and the last tokens of each substring.

At this point, n threads are created and a different substring is assigned to each one of them. Each thread runs a modified version of the naive operator precedence parsing algorithm (which we will describe in section 4.2); the threads return one of the following:

- an error code if they are sure that an error was present in their substring;
- a success code and the parse tree associated with the whole string;
- a work in progress code if they finished analyzing the string but could not reach the axiom. If this is the case, all the reductions performed by the thread are kept in a suitable data structure in order to be reused in successive iterations of the algorithm. This behavior is described in section 4.4.1.

When all threads finish, the algorithm checks all return codes: if it finds neither error nor success codes it prepares the strings for the next iteration. We have studied two alternative ways to do this:

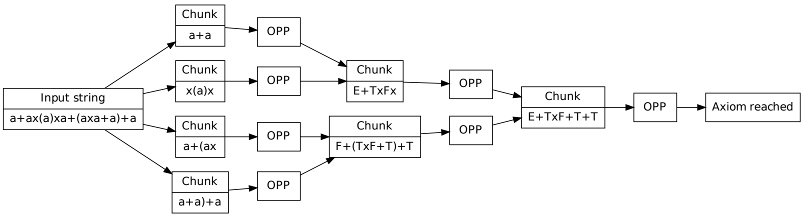


Figure 4.4: Logarithmic cascade of parallel recombinations parsing scheme.

sequential recombination After the first parallel phase all the substrings are joined and the next iteration proceeds with only one thread (figure 4.3).

logarithmic cascade of parallel recombinations The algorithm joins substrings two by two, thus reducing the number of threads in the next iteration by a factor of 2 (figure 4.4).

It is rather obvious that, independently from which mode is chosen, the last iteration is inherently sequential. How much work is done by the first phase or left for the following ones may also depend on the form of the grammar. This aspects is left for future investigations. The hope, however, is that most of the parsing process is done by the first iterations, and that the last one will only have few reductions left. This is backed up by the results we obtained with the experimentation that we did, which are presented in chapter 6.

4.4.1 Data management between iterations

As said in section 4.4 the information about the reductions performed by each thread needs to be stored in a suitable data structure in order to be used during the next iterations. This structure links an interval of tokens in the string to a nonterminal symbol and the parse subtree associated with their reduction, *i.e.* it is a list of elements defined as

$$< s, e, N, v >$$

Algorithm 4 Parallel Operator Precedence Parser

```
init data structures
divide string in $n$ substrings
while $\neg$ finished $\wedge$ $\neg$ error do
    determine context
    create and launch $n$ threads
    wait for threads
    if $n = 1$ $\wedge$ string is recognized then
        finished = true
    else if $\neg$ error then
        if mode = log then
            n := n/2
            stick two subsequent strings together
        else
            n := 1
            stick every substring together
        end if
    end if
end while
```

where s is the starting position in the string, e is the final position, N is the nonterminal symbol and v is the usual parse tree produced by a parser as described in chapter 2.

This data structure is modified by each thread when reducing a handle. Moreover this structure is used whenever the parser tries to identify a new token from the string: the algorithm has to check whether there is a saved nonterminal token in the current position or not; if it finds one, it returns it with the associated parse subtree and skips ahead in the string up to its final position; otherwise it simply reads the next terminal token.

Using this kind of structure it is possible to save all the reductions performed by a thread so that, in the following iterations, they will not have to be computed again. In general, a nonterminal saved in this structure will be used in the next iterations to identify wider handles and thus lead to further reductions, up to the axiom or an error.

Chapter 5

Implementation and optimizations

Up to now we have presented how the parallel parsing algorithm works. In this chapter we will present two things: how it has been prototyped (section 5.1) and implemented (section 5.2), and possible future optimizations that have been identified and left as a foreseeable development of this work.

5.1 Python prototype

In this section we will outline the approach we took in order to create a first working prototype of the parallel algorithm. Due to the fact that the idea of the algorithm grew while the prototype was being developed, it does not present a particularly optimized structure and some parts of it have indeed been changed in the final implementation.

The language chosen to code the prototype is *Python*, due to its ease of use and flexibility, which enabled us to continually make rather radical changes. The only shortcoming of Python is the *Global Interpreter Lock* (GIL) [18], *i.e.* the mutual exclusion lock held by the interpreter thread that makes it impossible to execute more than one thread at the same time, independently from the number of processors and cores available. Therefore, in order to achieve parallelism we decide to use *Jython*, *i.e.* an implementation of Python written in Java that actually supports executing multiple threads at the same time, because it compiles Python code to Java bytecode and therefore maps Python threads to Java threads.

As it has been said in previous chapters, the first step that needs to be done in order to parse a string is the lexing phase. In the prototype this is done *concurrently* with the parsing. This means that, whenever the parser needs a new token it has to invoke the lexer which will read characters from the input string until it recognizes one token and then returns it to the parser. This also implies that the lexer is used during the entire parallel parsing process. This practically translates to lexing done multiple times over the input, the first phase on the string and subsequent phases on pieces of string not yet recognized and on already parsed subtrees. It can be evinced that the lexer needs to be able to manage such nonterminals, expanding them into the adequate token with its associated parse subtree if necessary. However, the lexing may proceed in such a manner only if each token is composed of at most one character of the input string, since having tokens longer than one character implies that a token may have been cut in half during the division of the string and the algorithm would have to be able to handle this occurrence. This limitation was introduced mainly to simplify development of the prototype. In order to handle lexing, we used an external library called *Python Lex-Yacc* (PLY) [19], *i.e.* an implementation of lex and yacc in python.

In order to manage data between iterations (parse subtrees) as we have seen in section 4.4.1, we need a data structure that contains the subtrees and information regarding their position inside the input string. In our prototype this is done by means of the *marker structure*, *i.e.* a structure defined exclusively for this purpose. This data structure needs to be known by the lexer

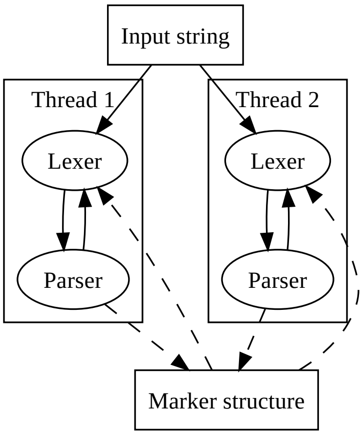


Figure 5.1: Scheme of a generic prototype parsing phase.

which will consult it. In fact, whenever the lexer is invoked by the parser, it has to verify the presence of a nonterminal symbol and, in case it finds one, it has to return it to the parser and skip the input string for the nonterminal's length. Moreover, when the parser recognizes a handle, it needs to interact with the marker structure in order to save the reduced nonterminal and its parse subtree for future iterations.

A scheme of how a generic parsing phase works in the prototype can be seen in figure 5.1. It is an example of two threads parsing an input string and it can be seen how the components interact with each other; the lexer reads from the input string and the marker structure to produce a token for the parser, which asks the input from the lexer and writes its output on the marker structure. We can see how the lexer and parsers are unique for each thread while the marker structure is global for the entire program.

5.2 C implementation

In this section we will see how the shortcomings of the prototype are solved in the actual implementation of the algorithm and some obvious optimizations have been implemented. The choice of the programming language to be used for the implementation fell on C; this was because being a less abstract language than Python it gives a finer lower level access to the machine and therefore allows a better control over memory and processor use and finer optimizations. Moreover, it has native support for threads (pthreads) and since it is compiled, it is generally faster than Python (see chapter 6 for benchmark results on both C and Python implementations).

As usual the first step is lexing; since the one character token limitation had to be removed in order to work in a real context and with real world languages, the lexing process had to be modified accordingly. In particular, lexing is no longer done concurrently with the parsing, but it is performed only once before the parsing phase on the entire string. The lexing process done in this way produces from the input string a token list that will be split among the threads. It can be seen how having lexing done in such a manner allows us to maintain unaltered the process of job balancement of the parsing phase, since the token list can be split in multiple parts exactly how the input string was in the prototype. Moreover the lexer now does not need to know anything about the nonterminal symbols, since it only has to deal with the input string which is entirely composed of only terminal symbols. Lastly it makes more sense to decouple two components that have to do jobs that are different and bound by a direct data dependence, in such a way that performing them concurrently is not possible. Similarly to what has been done on the prototype, we did not write the lexer, but we employed a scanner generated via the *Flex* [20], the mainstream free lexer generator.

As for what concerns data management between iterations, we decided to remove the dedicated marker structure and use already existent structures instead. Since the structure built by the lexer makes no distinction between terminal and nonterminal symbols because both are composed of a token and the corresponding parse subtree, it seemed natural to use the token list to store information for subsequent iterations. In fact, when a handle has

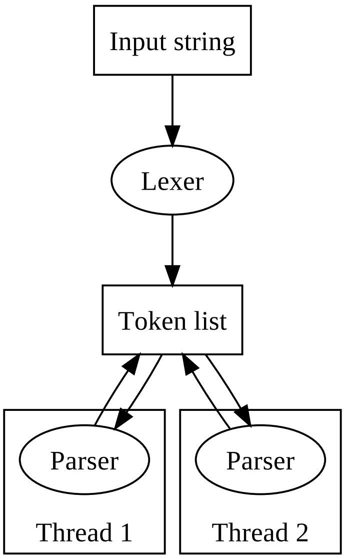


Figure 5.2: Scheme of the first parsing phase of the C implementation.

been reduced its corresponding nonterminal is created, the symbols forming the handle that were on the list are removed and the new nonterminal is inserted instead. This simplifies both the reading and writing phases of the parser, which are done directly by the parser that interacts with the token list without the use of external support structure or functions.

A scheme of how the first parsing phase works in the C implementation can be seen in figure 5.2. It can be seen how the lexer interacts with the input string and builds the token list, which is in turn directly accessed and modified by the different threads of the parser. What cannot be seen in the figure is that after the first phase the components above the token list disappear and only the parsers remain which interact with the token list modified by the parsers of the previous phase.

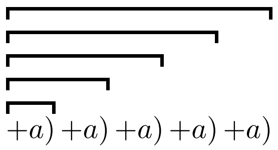
Another aspect to be discussed regarding the C implementation are data structures. Unlike Python, C does not implement data structures like trees and lists in its standard library. Therefore, in order to use these structures, we both implemented some of them and employed an external library for others.

In particular, we implemented basic double linked lists and trees, and we employed *Judy arrays* [21], *i.e.* data structures with high performance, low memory usage which implement dynamic associative arrays. In particular, Judy arrays are similar to 256-ary tries which guarantee for every level 256 possible keys and therefore, even when the number of indexed elements grows a lot, the number of levels of the tree remains limited; this means that very few operations are required to access a generic element of the trie, achieving a speed comparable to a direct access array.

The last point that needs to be discussed is the basic optimization of the reduction phase we introduced in the C implementation. In particular, this optimization concerns the beginning of the substrings and the research of handles when the precedence relation chain is in the form

$$\doteq \cdots \doteq \gtrdot$$

In fact, it is evident that with such precedence relations a complete handle will never be found, but the handle identification algorithm as seen in algorithm 2 on page 36 always tries to perform a reduction when a $\gtrdot$ precedence relation is found. This is a problem because, whenever the parser analyzes a string with such a relation chain it wastes time with useless operations. In order to avoid this, a *reduction counter mechanism* has been implemented; basically, this counter starts at 0 when the parser begins the analysis of a substring, it is incremented every time a $\lessdot$ precedence relation is found and decremented when a handle is successfully reduced. The meaning of this is that a reduction is possible only if the counter has a value greater than 0, and therefore the algorithm will search for a handle only if this condition is verified. An example of a pathological substring expressed with the usual parenthesized arithmetic expressions grammar that forces the parser to continually waste time in searching for impossible handles could be the following:



In fact, the only possible reductions here are to reduce all the 'a' to $\{S, E, T, F\}$. Without the optimization we just described, every time the parser finds a ')' after having reduced the preceding 'a', since $+ \gtrdot$), it would try to perform a reduction without success and analyze all the tokens which are below the corresponding bracket shown above; with the optimization in place this behavior is avoided because there are no $\lessdot$ precedence relations matching the $\gtrdot$ ones.

5.3 Possible future optimizations

In this section we will present the optimizations that during the development of the thesis and the algorithm have been identified as possible but have not yet been implemented both because of time constraints and because we wanted to have a basic working implementation of the algorithm first and only then think about optimizing it. Therefore in this section we will see what is being worked on at the time of writing this thesis and other optimizations that have been more or less defined. It has to be specified that since these optimizations are only proposed and not yet implemented, there still exist no experimental data in favor or against their introduction.

5.3.1 Optimized cutoff points

We have seen in section 4.4 that before the parsing phase, the input is divided into n parts and each part is associated with a thread that will parse it. We have seen that this subdivision is done by just balancing the substrings from the length point of view. A possible and rather immediate way of improving the subdivision of the input is to look for suitable cutoff points, *i.e.* patterns in the form

$$\cdots \gtrdot \lessdot \cdots$$

If all the cutoff points are chosen in this way, the input will be split as follows:

$$\# \lessdot \cdots \gtrdot | \lessdot \cdots \gtrdot | \lessdot \cdots \gtrdot \#$$

The advantage of such a pattern is that it guarantees for both the preceding and subsequent substrings of a cutoff point at least one reduction. While one could argue that this is not a big improvement, it is also true that it can be easily done by searching locally for such a pattern from the cutoff points obtained by dividing the input into equally long parts.

The cost of such a local research is the one of scanning the list of tokens into one or another direction, *i.e.* reading a token and a precedence relation for each step, since the other precedence relation is kept from the previous iteration of such a research. From our point of view this research is probably justified since it opens up for one thread possibly many reductions, and having one thread that does nothing is nevertheless expensive since it consumes resources in order to be instantiated and to build support data structures such as the stack.

5.3.2 Optimized logarithmic cascade of parallel recombinations

We have seen in section 4.4 that one of the two ways we designed to prepare iterations that follow the first one is the logarithmic cascade of parallel recombinations. As it is implemented now this is done simply by taking two subsequent parts of the token list starting from the first two and linking them together to obtain a new one. This does not introduce any form of rebalancing of the workload between iterations: in the precedent iteration one thread might have reduced all its part into very few tokens while the others might have reduced almost nothing; therefore, in the next iteration, the thread that will work on the part of the token list that contains the part analyzed by the first thread will have to perform much less work than the other threads. An example of this problem expressed with the usual parenthesized arithmetic expressions grammar is the following:

$$a + a + (\quad \mid \; a + a + a) \quad \mid \; \times (((a + a \quad \mid \;) \times a) \times a) \tag{5.1}$$

$$E + (\quad \mid \; E) \quad \mid \; \times (((E \quad \mid \;) \times F) \times F) \tag{5.2}$$

In (5.1) we can see the input string and how it has been divided among threads. It can be observed that all the threads obtain inputs of about the same length. In (5.2) the result of the first iteration is displayed; it can be noticed that the first two threads reduced their parts respectively into 3 and 2 tokens, while the third and fourth threads have left their parts almost untouched. In the next iteration the first thread will analyze a part 5 tokens long while the second will analyze a part 12 tokens long; moreover the first thread will have to perform less reductions than the second one since its part is less nested than the other one.

This problem could be solved with an accurate choice of subsequent cutoff points. Instead of just joining two parts together, there is a rebalancing of the jobs on each iteration performed by dividing the resulting token list into parts with equal length. In addition to this, after having obtained the new cutoff points, a local research on the precedence relations can be performed in the same way that we saw in section 5.3.1. Continuing the previous example, we can see how using this policy produces the following:

$$E + (E) \times (((\;\mid\; E) \times F) \times F) \tag{5.3}$$

$$E + (E) \times \;\mid\; (((E) \times F) \times F) \tag{5.4}$$

In (5.3) the token list has been rebalanced only according to its length; despite this, the resulting cutoff points are suboptimal, since the second thread will not perform any reduction. Using also the local research on the precedence relations the cutoff point shown in (5.4) is obtained; in this case the first thread performs the same reductions that it would have done in the first case, but the second is able to completely reduce the parenthesized structures. The results of the two choices are shown respectively in (5.5) and (5.6).

$$T \times (((\mid E) \times F) \times F) \tag{5.5}$$

$$T\times \mid F \tag{5.6}$$

The performance overhead brought in by rebalancing the jobs for each iteration is expected not to be a particular burden. This is because it only requires to know the length of the token list, which can be computed by the different threads while they perform their analysis by reducing the length of their part each time they perform a reduction by its length. As for what concerns the local research, the same considerations expressed in section 5.3.1 hold.

5.3.3 Optimized handle identification and reduction

What we introduce in this section is a rather radical change on the parsing algorithm. As it is now, the soundness of this optimization has not been practically evaluated yet. The basic idea is to replace the mechanism used to recognize the production rules; as we have seen in chapter 4 this is done with a handle identification phase: once a $\gtrdot$ is found, the handle identification algorithm tries to recognize its corresponding production rule. This obviously wastes time as, for each reduction, first we read it from left to right and then from right to left. Our idea involves changing the handle identification phase and starting the recognition of production rules while the handles are being read. Intuitively this can be done by replacing the reduction tree with another analogous data structure that has the same purpose but can work on productions read from left to right. With such a structure it would be possible to perform the recognition of productions while identifying the corresponding handle.

While the complexity of the current algorithm is concentrated in the handle identification phase, *i.e.* most of the work is performed in the iteration during which a $\gtrdot$ relation is recognized, the complexity in the new idea is spread over all iterations: part of the reduction job is performed starting from the first $\lessdot$, throughout all the $\doteq$ up to the $\gtrdot$. Intuitively, whenever a $\lessdot$ is found, a new reduction recognition context is opened; whenever a $\doteq$ is found, the algorithm proceeds with the recognition within the same context and when a $\gtrdot$ is found, the algorithm closes the context and proceeds with the reduction.

What is expected to be obtained by introducing this optimization is mainly a gain in execution time; in fact, by analyzing each handle only once instead of twice we expect the execution time of each parsing phase to be significantly lower. However we also expect increased memory usage due to the fact that the algorithm needs to manage the reduction recognition context of each handle.

Chapter 6

Experimental evaluation and comparison

The work of this thesis is accompanied by a proof of concept, which we already presented in chapter 5, that demonstrates how the parsing process using FGs is actually parallelizable, and scales reasonably well raising the number of computing units. In this chapter we will discuss the results obtained by running benchmarks on both the Python prototype (section 5.1) and the C implementation (section 5.2). In particular, in section 6.1 we will present the experiments done with the Python prototype and we will see the obtained results, both in terms of absolute execution times and speedup over the sequential operator precedence parser; in section 6.2 we will present in a similar way the results obtained with the C implementation over a different set of experiments and we will briefly discuss the difference in performance between the two implementations; finally, in section 6.3, we will present a comparison between our parser and common state of the art traditional parsers.

For all the experiments we used the same platform, which is a multiprocessor machine that allowed us to perform tests on a number of different thread configurations. In particular, the platform is equipped with four

Quad-Core AMD Opteron 8378 clocked at 2.4 GHz, 32 GB of RAM and running GNU/Linux (Ubuntu 10.04 LTS), targeted for a 64bit x86 architecture.

Since the machine is equipped with four quad-core processors and therefore enabled us to perform benchmarks with a variable number of threads up to 16. The GNU/Linux operating system which the machine is running gave us access to the native implementation of POSIX compliant threads:, the NPTL and to the standard implementation of the glibc malloc function.

The main result of the experimentation was the measurement of the execution times of the parsing process applied to different inputs, from which followed the computation of speedup data, which is expressed as

$$\text{Speedup}_n = \frac{T_1}{T_n}$$

where n is the number of employed threads and T_i is the execution time of the algorithm using i threads.

By execution time of the parsing process we mean not only the time that the different threads spend in order to parse their substrings, but also the input preparation (*i.e.* time used to divide the string into substrings), thread preparation (*i.e.* time used to create, launch and synchronize the different threads) and, in the case of the Python prototype, even the lexing time. As for the prototype, the instrument used to measure times was a call to the Python *time* function, while for the C implementation we employed the *clock_gettime* function, which uses the system-wide real-time clock and therefore gives a high resolution. Given the nature of this functions and the operating system we decided that only times in the range of seconds or more are a reliable measure of the algorithm speed.

Finally, in order to reduce the effect of other system programs running on the machine during the benchmarks each test has been performed at least 10 times and the obtained values have been averaged. By doing this, differences between values of various test samples are averaged out, giving a more reliable result.

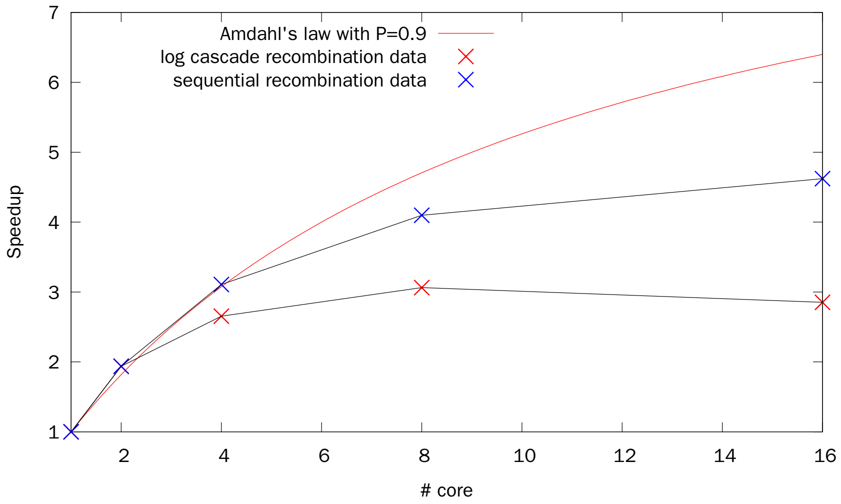


Figure 6.1: Python: Speedup with respect to the number of cores.

6.1 Python prototype

Since the Python prototype was developed with the purpose of proving the proper operation of the parallel operator precedence parsing algorithm, we deemed sufficient to test the implementation with a synthetic language as a benchmark. Therefore we opted to execute a series of tests with the usual parenthesized arithmetic expressions grammar described in chapter 2. The input text for the tests were all generated by an algorithm we developed that builds random strings belonging to the language defined by a given grammar. We used this algorithm to build valid strings of different lengths that we used for the Python prototype. The strings that we generated with the string generation algorithm and that we used in order to test the prototype have lengths respectively of 100, 500, 1000, 3000, 5000, 10000, 20000, 50000, 100000 and 250000 characters. Since, as we said, for the parenthesized arithmetic expressions grammar the token are all of length one, the string length in character corresponds to the input size measured in tokens.

The first result is shown in figure 6.1 on page 60. In the figure we see how the two recombination techniques presented in section 4.4 perform with

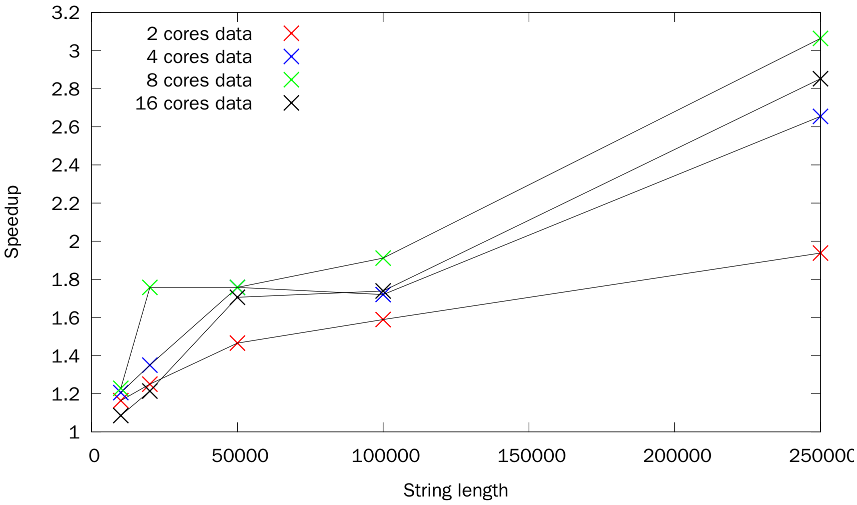


Figure 6.2: Python: Speedup with respect to the length of the string with logarithmic cascade of recombinations.

respect to number of threads employed. The speedups are calculated on the longest string, *i.e.* the one with 250000 tokens. We can observe that both the versions scale well up to 4 cores, where the sequential one has a value of ~ 3.10 whereas the logarithmic cascade one has a value of ~ 2.65. We can see that beyond that number of cores the speedup values tend to saturate, with a maximum of ~ 4.62 obtained by the sequential version with 16 cores. In this figure we see what is a general trend in the algorithm, *i.e.* that the sequential recombination technique performs generally better than the logarithmic cascade. This might sound strange, since the latter uses more parallelism throughout its execution. However, we noticed during the execution of the tests that the iterations following the first one perform little to no reduction and therefore the cost of launching more threads impacts negatively on the execution time. We expect this behavior to change with the implementation of the optimization described in section 5.3.2 and hopefully see the logarithmic cascade outperform the sequential one.

The next two figures show how the two recombination techniques perform with respect to the length of the input. In particular, figure 6.2 on page 61

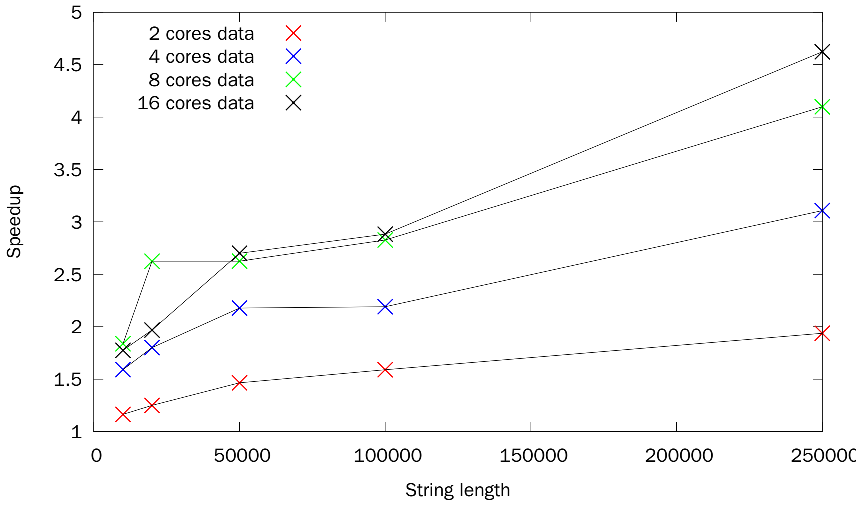


Figure 6.3: Python: Speedup with respect to the length of the string with sequential recombination.

shows the behavior of the logarithmic cascade version while figure 6.3 on page 62 exhibits the conduct of the sequential one. We can notice that the speedups change depending on the input. This effect can be explained by the two following causes:

- the speedup on the shorter strings (length < 50000) is generally lower since the times are rather small and therefore the creation and management of threads impacts a lot;
- the speedup observed on the last string is higher. We have already seen in chapter 5 that depending on the cutoff point the parsing process is more or less parallelizable; in general we can say that the input structure impacts the degree of parallelism that the algorithm is able to exploit. Therefore, strings with a particular structure tend to give better performance and this is the case of the last string.

Another question that we wanted to answer was whether the execution time of the implementation of the algorithm was as theoretically expected, *i.e.* asymptotically linear with respect to the input length. The answer is

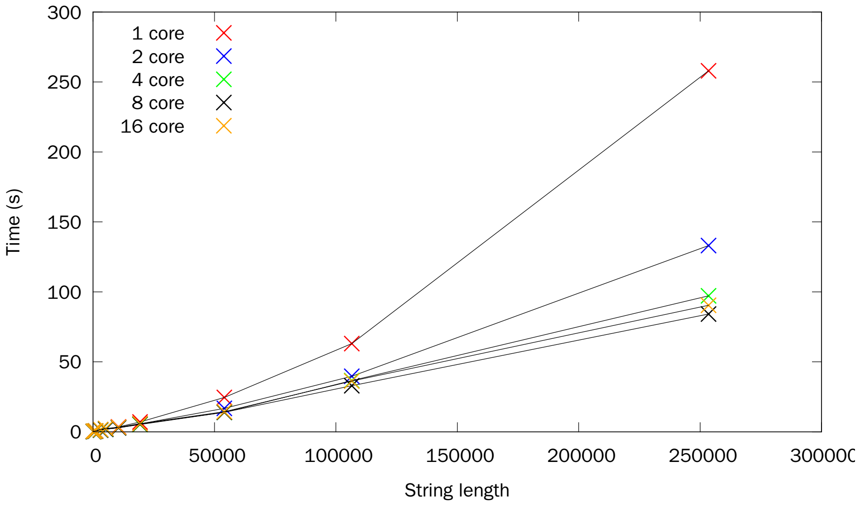


Figure 6.4: Python: Execution time with respect to the length of the string with logarithmic cascade of recombinations.

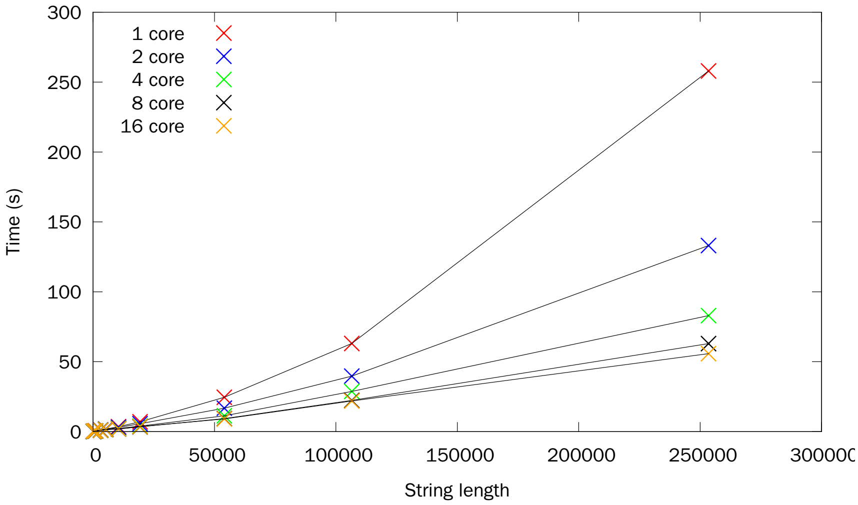


Figure 6.5: Python: Execution time with respect to the length of the string with sequential recombination.

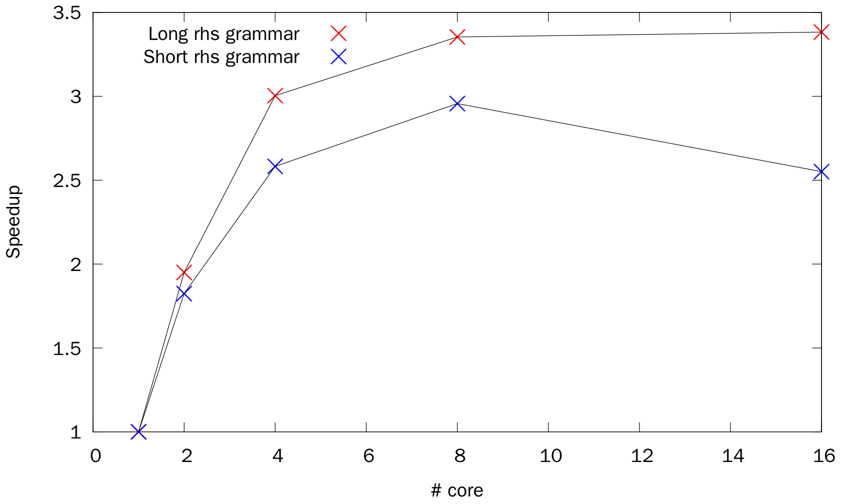


Figure 6.6: Python: Comparison of speedup with respect to the number of cores between grammars with long and short rhs with logarithmic cascade of recombinations.

shown in figures 6.4 on page 63 and 6.5 on page 63, where the absolute parsing time of different thread configurations with respect to the length of the input can be seen. What can be noticed in the two figures is that the parsing time for the 1 core configuration seems superlinear and also for the other configurations it can be noticed that the points do not follow a straight line. This would lead to the conclusion that the algorithm complexity is not linear but, as we will see in section 6.2, this is not the case; in fact, non linearities displayed by the prototype can be explained by the fact that it is implemented in a very high level language which is in turn compiled into another high level language and therefore the multiple levels of abstraction justify such a behavior.

The last thing we wanted to investigate was the behavior and performance of the algorithm with respect to different grammars. In particular, we wanted to inspect whether or not the length of the rhs of production rules influenced the performance of the parallel parser. In order to do so we defined two grammars, G_{long} and G_{short} such that the rhs of the rules in G_{long} were composed of more symbols that the rhs of the rules in G_{short} and that

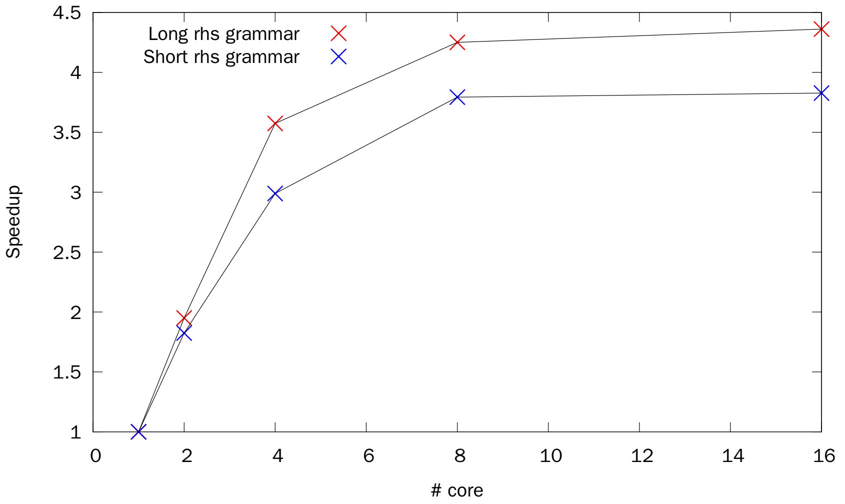


Figure 6.7: Python: Comparison of speedup with respect to the number of cores between grammars with long and short rhs with sequential recombination.

$L(G_{long}) \subset L(G_{short})$. We then generated a string that belonged to $L(G_{long})$ and parsed it with both grammars.

The reason why we did this benchmark was that we feared the longer production rules to hinder the performance of the parallel parser: having longer rules implies that cutoff points could more easily be placed in a way that would prevent the recognition of entire rules and thus long parts of the substrings would be left unparsed. As it can be seen in the figures, however, not only this is not the case, but the grammar with longer rhs performs generally better. This happens because the number of cutoff points is generally low with respect to the length of the string and even if the cutoff points are misplaced and they cause a problem that will propagate for several tokens, it is improbable that this will spread throughout thousands of tokens.

This was everything that we wanted to investigate on the Python prototype and we can conclude this section by saying that the first prototype of the parallel operator precedence parser is definitely a proof of concept of the theoretical parser described in chapter 4: despite the fact that it is implemented in a very high level language and the algorithm itself was developed while writing the prototype, this shows what we believe to be a good level of scalability and gain in performance with its parallelization. In section 6.3 we will also see how it performs compared to a traditional LR parser implemented in Python.

6.2 C implementation

In this section we will present the results obtained from the benchmarks by employing the C implementation of the parallel parsing algorithm. Since the C implementation is a first step towards a state of art parser, it seemed logical to test it with real world languages. In particular, as mentioned in section 3.2, we chose JSON as benchmark language; this has been done for the following reasons:

- it is widely used (inspired from data structure representation in *JavaScript* but it is also used as a format to interchange data in a human-readable way) and different computer languages have parsers, parsing libraries and export libraries that support it;
- it is a first step towards the definition in FG form of JavaScript and XHTML/HTML5, which we intend to obtain in order to further test our parser with the standards of the web languages;
- it is defined by a simple grammar which was easy to put in FG form; since usage is limited to representing data, it has very few rules and they have short rhs.

Since our parser works with FGs, we had to transform the JSON grammar taken from [22] into a suitable form. In particular, the obtained grammar is defined as $JSON_{FG} = \{V_N, \Sigma, P, S\}$ where the nonterminal alphabet is

$$\begin{aligned} V_N = \{&S,\ OBJECT,\ MEMBERS,\ PAIR,\ STRING,\\ &VALUE,\ ARRAY,\ CHARS,\ ELEMENTS\} \end{aligned}$$

the terminal alphabet is

$$\Sigma \;=\; \{\boldsymbol{\{},\ \boldsymbol{\}},\ \boldsymbol{,},\ \boldsymbol{:},\ \boldsymbol{number},\ \boldsymbol{bool},\ \boldsymbol{''},\ \boldsymbol{char},\ \boldsymbol{[},\ \boldsymbol{]}\}$$

and the production rules are the following:

$$\begin{aligned}
S &\rightarrow OBJECT \\
OBJECT &\rightarrow \boldsymbol{\{}\,\boldsymbol{\}} \mid \boldsymbol{\{}\, MEMBERS\, \boldsymbol{\}} \\
MEMBERS &\rightarrow PAIR \mid PAIR\,\boldsymbol{,}\, MEMBERS \\
PAIR &\rightarrow STRING\ \boldsymbol{:}\ VALUE \\
VALUE &\rightarrow STRING \mid \boldsymbol{number} \mid OBJECT \mid ARRAY \mid \boldsymbol{bool} \\
STRING &\rightarrow \boldsymbol{''}\,\boldsymbol{''} \mid \boldsymbol{''}\, CHARS\, \boldsymbol{''} \\
CHARS &\rightarrow \boldsymbol{char} \mid \boldsymbol{char}\, CHARS \qquad (6.1) \\
ARRAY &\rightarrow [\,] \mid [\, ELEMENTS\,] \\
ELEMENTS &\rightarrow VALUE \mid VALUE\,\boldsymbol{,}\, ELEMENTS
\end{aligned}$$

The only difference in the production rules from the original grammar is in rule (6.1), which originally were the two rules

$$\begin{aligned}
CHARS &\rightarrow CHAR \mid CHAR\ CHARS \\
CHAR &\rightarrow \boldsymbol{char}
\end{aligned}$$

that obviously make the grammar not in operator form, since there are two nonterminals close to each other. The only other difference is the terminal alphabet which in our version has symbols like ***number*** and ***bool*** that are generated by the lexer, whereas in the original version of the grammar they were rules that transformed a series of digits into a number and bool was the alternative of either ***true***, ***false*** or ***null***.

Despite the fact that we had an automated string generator and that therefore we could have generated random input, we wanted to stay true to the purpose of the benchmarks which were to test the algorithm in a real world environment. Therefore, the input that we used is the following series of strings, listed in order of length:

2.7K response of a Google search in JSON;

30K e-business service catalog data;

80K configuration file of a Chrome plugin;

150K Gospel of John;

1.6M Italian statistic data-bank ISTAT on food consumption (originally a csv file);

10M Google file on monograms on Google books (originally a csv file);

The last thing that needs to be specified before presenting the results is that the tests have been performed on three different versions of the implementation of the algorithm, which give different results. The version depends on the usage or not of two external libraries:

- Judy arrays have already been presented in section 5.2; in one version of the implementation (subsection 6.2.1) we used them to implement the token list, since they provide low cost (in terms of time, compared to scanning the entire list) insertions in arbitrary points of the list; in the other versions of the implementation (subsections 6.2.2 and 6.2.3) the usage of the token list has been reimplemented in a way similar to a stack, *i.e.* we keep a pointer to the last used element of the list so that, since we only have to perform local changes (*i.e.* to the substring in a near of the last read token), we do not have to execute a research for a particular position in the global list;

- the Hoard allocator [23] is a dynamic library that can be preloaded and acts as a wrapper for the malloc and free functions that we use

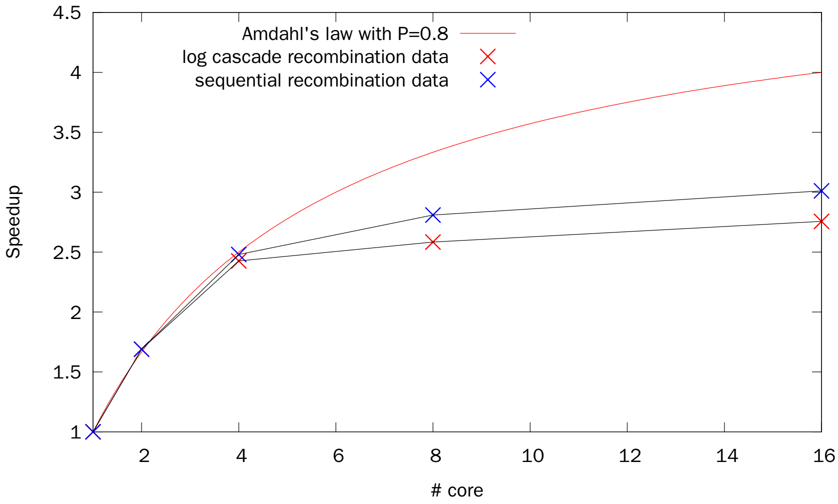


Figure 6.8: C: Speedup with respect to the number of cores without Hoard allocator and with Judy arrays.

extensively in our code optimized for multiple thread programs; its impact on performance is discussed in subsection 6.2.3.

We do not present the combination using both Hoard allocator and Judy arrays because it gave bad performance results both in terms of absolute times, speedup and scalability; the reasons why this happens are due to memory problems which are the same as the ones explained in sections 6.2.2 and 6.2.3.

6.2.1 Results using no Hoard allocator but Judy arrays

The first result of the C implementation can be seen in figure 6.8 on page 69, which shows the speedups with respect to the number of cores employed obtained with the two different recombination techniques. The string which gives these results is the longest one, *i.e.* the one with ten million characters. In accordance with the results obtained with the Python prototype, we still see that sequential recombination performs better than the logarithmic cascade. Moreover, we can observe that the speedup saturate at lower values, with a maximum of ~ 3 obtained by the sequential recombination technique using 16 cores. This is a rather low value, but it can be noticed that with 2 cores the speedup is ~ 1.68 and with 4 cores it is ~ 2.47 which are, from our point of view, acceptable values, since the maximum theoretical speedups, considering a level of parallelization of 100%, are 2 and 4, and the actual level of parallelization is lower since the last iteration is inherently sequential. An estimation of the speedup with Amdahl's law is shown in figure 6.8: considering the portion of parallelized code to be 80% of the total, we obtain that the maximum speedups for the different thread configurations are ~ 1.66, 2.5, 3.33 and 4. However, it has to be remembered that Amdahl's law gives an overestimation that does not take into account the practical limitations that an actual implementation implies and from which our implementation surely suffers.

As with the prototype, there is a saturation effect that comes into play after 4 cores that strongly limits the growth of the speedup. As of now, we think that this is the consequence of memory management problems, in particular coming from the extensive usage of memory allocation and deallocation. As we will see in the following subsections, the architecture we employed to perform the tests is made in a way that when more than one thread is running on a processor (*i.e.* 8 and 16 threads configurations), the bandwidth available to access the RAM memory and L3 cache is shared between threads running on the same processor; this obviously causes a bottleneck when making extensive usage of memory.

The next result is shown in figures 6.9 on page 71 and 6.10 on page 71. There are two interesting things to notice: the first one is that the shorter strings (2700 and 300000 characters long) do not display any particular speedup. This is because the execution times are very small and therefore the initialization time of the threads burdens a lot on the whole parsing time, thus nullifying the speedup. After those strings (starting from the 80K

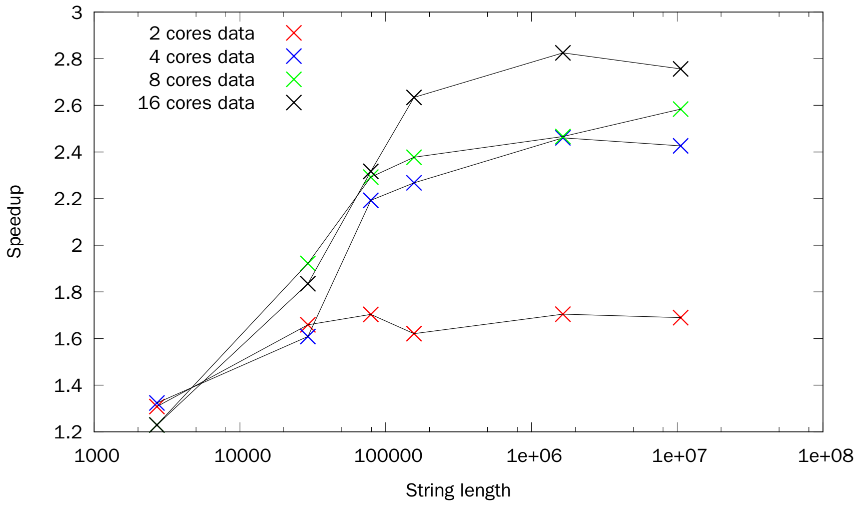


Figure 6.9: C: Speedup with respect to the length of the string with logarithmic cascade of recombinations without Hoard allocator and with Judy arrays.

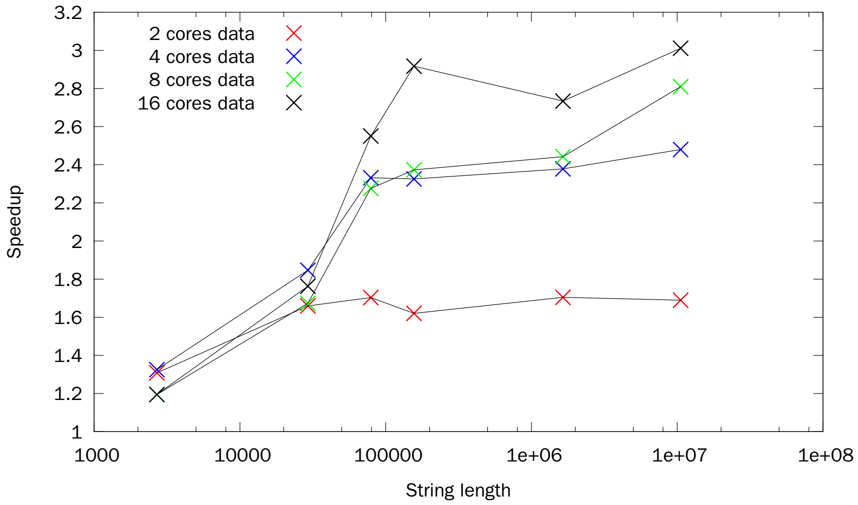


Figure 6.10: C: Speedup with respect to the length of the string with sequential recombination without Hoard allocator and with Judy arrays.

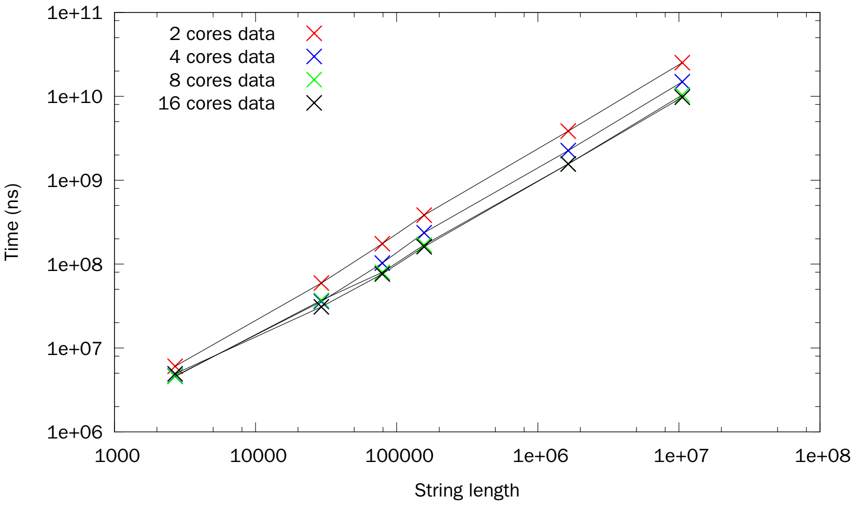


Figure 6.11: C: Execution time with respect to the length of the string with logarithmic cascade of recombinations without Hoard allocator and with Judy arrays.

characters long string) the speedups remain more or less constant, showing that the algorithm scales well with respect to the input length.

The last result is shown in figures 6.11 on page 72 and 6.12 on page 73. There are two things to notice here: the first is the comparison in terms of absolute times with the Python prototype. While the execution times of the prototype were in the order of seconds, here the timings on inputs of roughly the same length are several orders of magnitude smaller. Comparing the run-times of the two implementations on the same grammar (parenthesized arithmetic expressions) a factor of 60 was obtained. The second thing to notice is that, in contrast with what we have seen with the prototype, the results show an almost perfectly linear behavior. This validates our thesis stated in section 6.1, *i.e.* the algorithm complexity is linear and non linearities shown in the prototype come from the abstraction layers and not the algorithm itself.

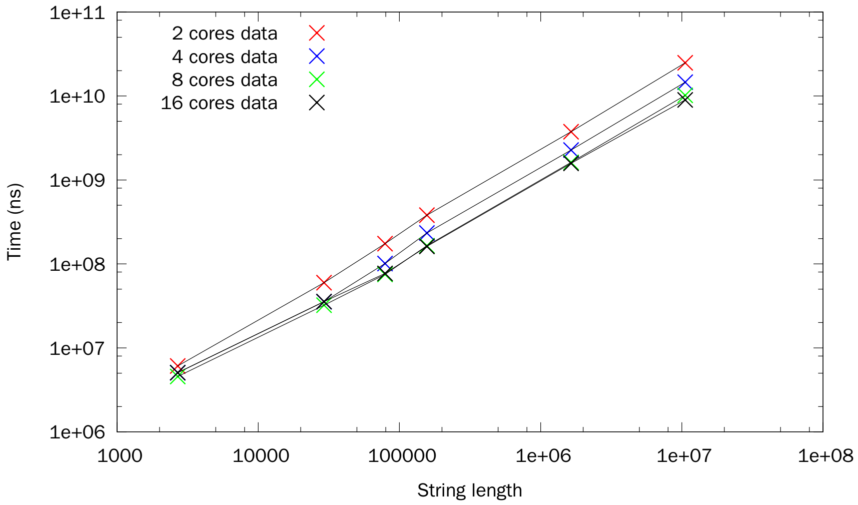


Figure 6.12: C: Execution time with respect to the length of the string with sequential recombination without Hoard allocator and with Judy arrays.

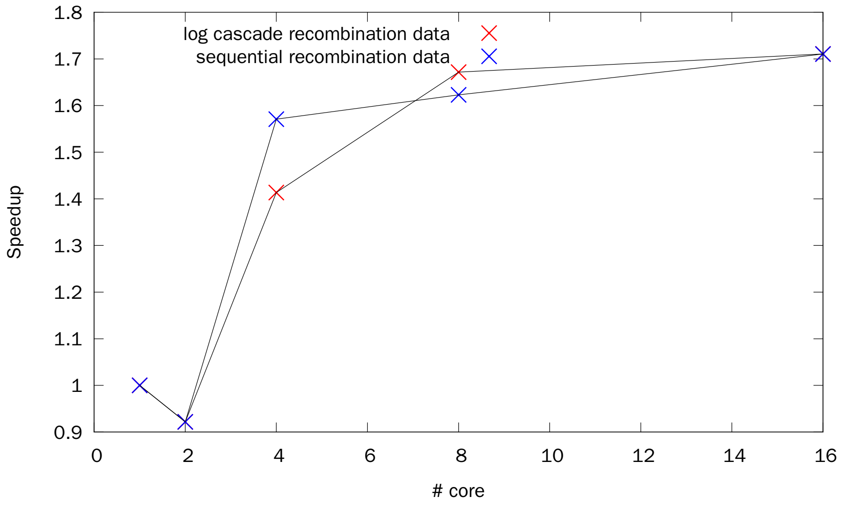


Figure 6.13: C: Speedup with respect to the number of cores without Hoard allocator and without Judy arrays.

6.2.2 Results using no Hoard allocator and no Judy arrays

A first result of this configuration is shown in figure 6.13 on 73, where the speedups with respect to the number of cores employed of the different recombination techniques are presented. It can be noticed that speedups of this version are generally lower than what we have seen with the previous one with a maximum of ~ 1.71 at 16 cores. A more in-depth representation of the results is shown in figures 6.14 on page 76 and 6.15 on page 76. There, the speedups are not consistent with respect to the length of the analyzed string. For the shortest strings (up to 80K characters) the usual behavior can be noticed, the speedup tends to grow because the time of the initialization operations becomes smaller compared with the entire execution time. However, after these strings, the speedups tend to shrink.

This behavior can be explained making some considerations on the architecture of the employed machine. Each processor has 6MB L3 cache shared between its cores (each one having an additional 512KB L2 cache, considering that AMD caches are exclusive). This makes a total of 8MB of L2 + L3 cache available per processor and a grand total of 32MB considering all four CPUs. The major part of the data that is being used during the parsing process is due to the token list and the parsing stack. In fact, each element of the token list is composed of 16 bytes, while each element of the stack is composed of 24 bytes. Since the token list is long roughly as the input string, we can estimate the amount of memory it occupies depending on the input: at 150K it will take $\sim 2.3MB$, at 1.6M it will take $\sim 25MB$ and at 10M it will take $\sim 156MB$.

Moreover the number of concurrently growing stacks is the same as the number of threads employed (obviously each parsing thread has its own stack) and therefore, as the number of threads grows, the stacks will require more and more memory and more and more memory accesses. A last consideration is the one on the grammar used to express JSON: as shown at the beginning of this section, some of the most frequently used rules are right recursive. For our algorithm, this implies that each of these rules may generate a long sequence of recursions which require the stack to grow until the last element of the sequence is reduced, at which point all the sequence can be reduced. This obviously implies a great usage of memory both in terms of allocated quantity and of frequency of accesses.

Taking all these consideration into account, we can explain the behavior with respect to the longest strings (from 150K characters upward) shown in the figures as follows:

- in general, the 2 cores line is lower than the other configurations: this is intuitive and what is expected to happen;

- for the 150K and 1.6M characters long strings the 4 cores configuration is the fastest: this can be explained considering that each thread is mapped on a different processor and therefore has access to its full cache with full bandwidth; this does not happen with the 8 and 16 cores configurations, which have increasing competition to access memory;

- for the last string (10M characters) the amount of required memory is such that it cannot be completely stored in cache and will surely have to be located inside RAM: this means that the effect of the exclusive access to the cache of which the 4 cores configuration enjoys is nullified by the fact that it has to access the slower memory; in fact we can see that the order of the configurations is restored to what it is intuitively expected;

Even if the speedup results are worse than the first version of the implementation presented, the reason why we implemented and are now presented this version is that it improves performances in terms of absolute execution times. As an example, the time required to parse the 10M characters string with one core using the previous version is $\sim$ 24.8s, while with this version

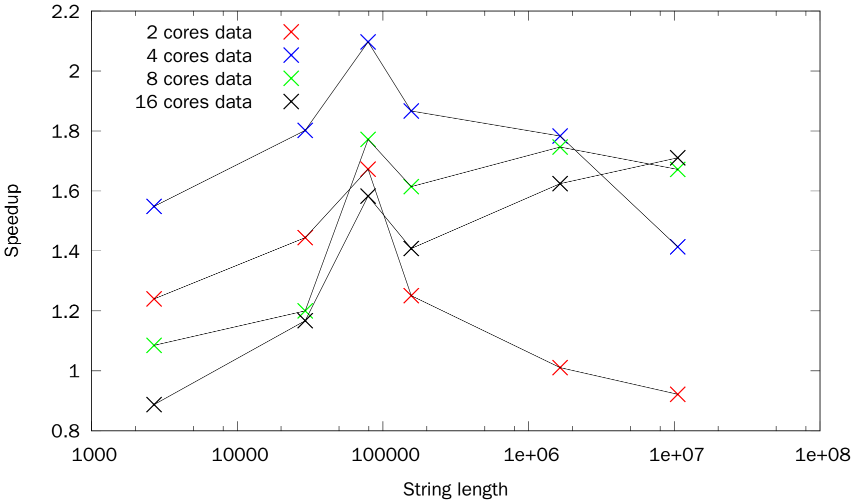


Figure 6.14: C: Speedup with respect to the length of the string with logarithmic cascade of recombinations without Hoard allocator and without Judy arrays.

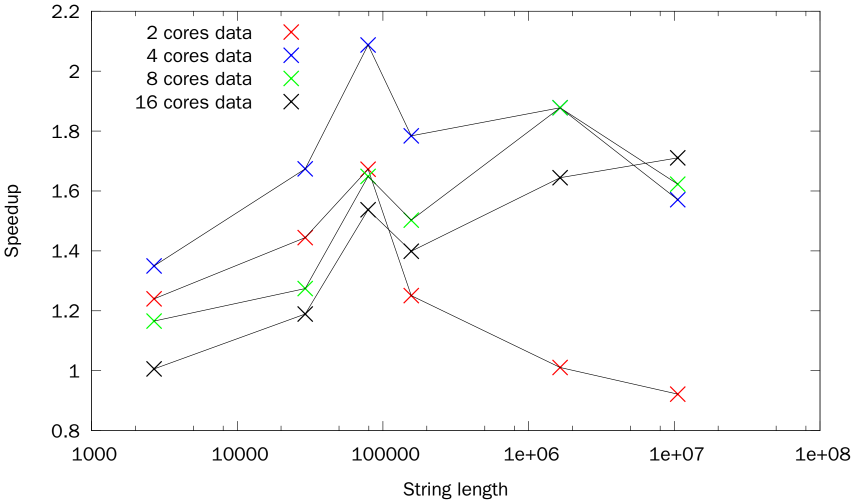


Figure 6.15: C: Speedup with respect to the length of the string with sequential recombination without Hoard allocator and without Judy arrays.

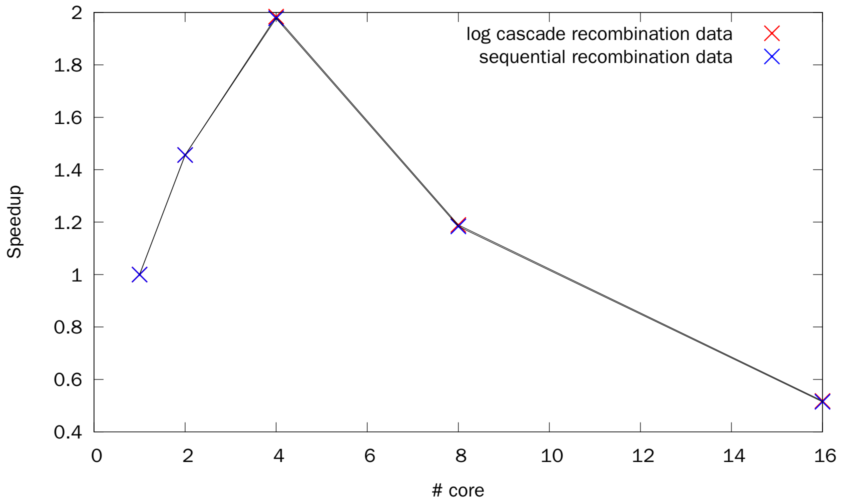


Figure 6.16: C: Speedup with respect to the number of cores with Hoard allocator and without Judy arrays.

it is ∼ 11.2s. We indeed believe that this is the direction along which future development efforts will move since there is no further need for a structure as complex as Judy arrays to manage the token list in our parser, and its usage definitely brings in an unnecessary overhead. The problem of which this version suffers are due to the fact that it still is being under development and these are the first results that we obtained.

6.2.3 Results using Hoard allocator but no Judy arrays

In order to ease the effect of the bottleneck introduced by the intensive use of memory, we tried to introduce the Hoard allocator. Hoard makes memory allocations and deallocations faster in a multithreaded context by preallocating a pool of memory that can be then used when necessary. This helps because it reduces the number of allocations that need to be performed; since those, as we said, tend to be serialized, Hoard generally improves performance of memory intensive multithreaded applications.

The results of the tests run with the Hoard allocator are shown in figure

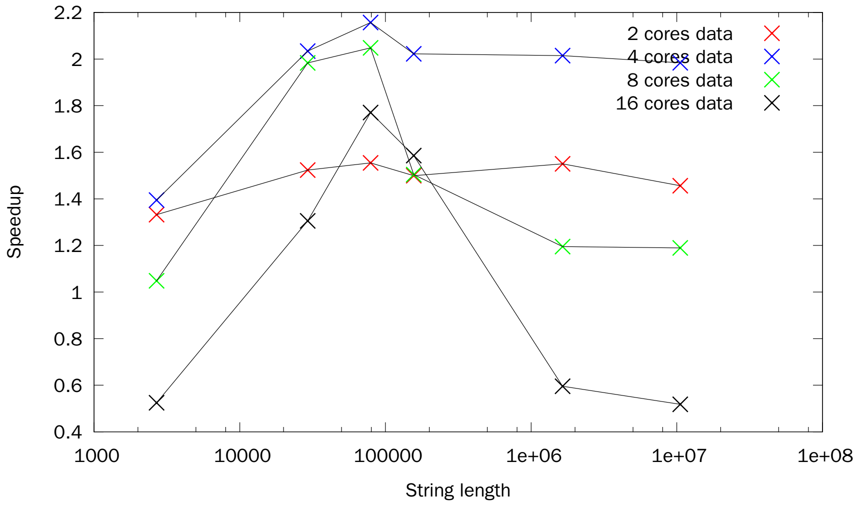


Figure 6.17: C: Speedup with respect to the length of the string with logarithmic cascade of recombinations with Hoard allocator and without Judy arrays.

6.16 on page 69 and in more details in figures 6.17 on page 78 and 6.18 on page 79. In general the same considerations expressed in subsection 6.2.2 hold and therefore the implementation still presents problems with respect with the speedup. However the effect of Hoard is indeed noticeable, especially in the 2 and 4 cores configurations, where the speedup is consistent with respect to the length of the string and in general the growth in speedup is acceptable for the considerations made in subsection 6.2.1. After that point, it looks like that Hoard is no longer able to hide the problems with memory management and therefore the speedup falls, especially on the longer strings.

6.3 Comparison with traditional parsers

In this section we present a comparison between the parallel operator precedence parser implementations and traditional parsing algorithm. For the C implementation, the choice of the traditional parser was rather obviously a parser generated with Bison, which is the state of the art of parser generator libraries which generates LALR(1) parsers. As we have seen, the

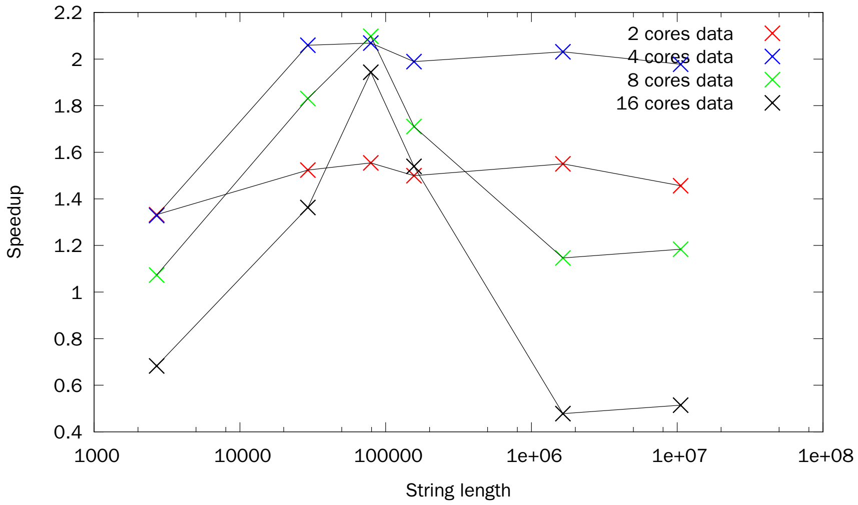


Figure 6.18: C: Speedup with respect to the length of the string with sequential recombination with Hoard allocator and without Judy arrays.

C implementation is much faster than the Python prototype and therefore in our opinion it was not meaningful to compare the prototype with Bison. Since we already used PLY to perform lexing in our prototype and as we have mentioned it is an implementation of both Lex and Yacc in Python, it seems rather natural to pick it as benchmark for Python LR parser.

The meaning of this test is that we wanted to investigate how our implementations performed compared to the state of the art implementation of sequential parsers. From a theoretical point of view the operator precedence parser is in the same complexity class as the other parsers and therefore the development of a parallel version is trivially justified. However, in practice, state of the art parsers have been developed for various years and are rather optimized. This, in addition to the fact that their complexity is linear with respect to the length of the input, makes the implementation of a competitive parallel parser all but trivial and the results will be discussed in the following subsections.

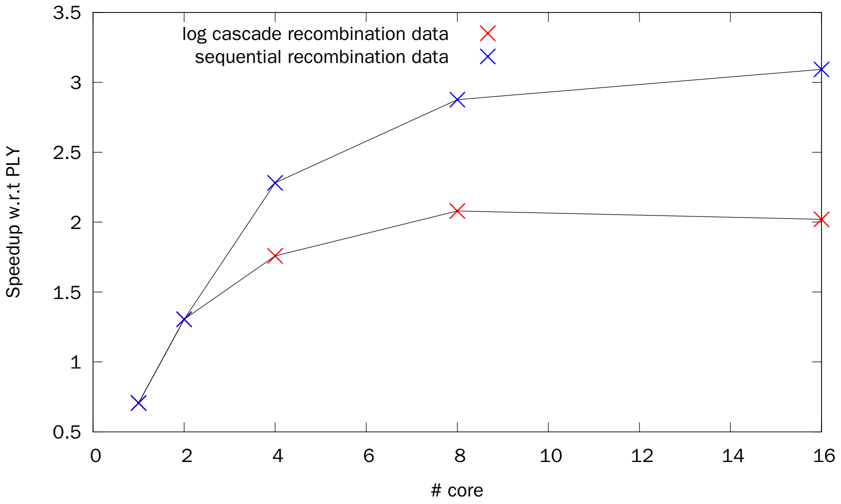


Figure 6.19: Python: Speedup compared to the PLY execution time with respect to the number of cores employed.

6.3.1 Python prototype compared with PLY

As with the other tests on the prototype, the used grammar is the usual parenthesized arithmetic expressions grammar and the employed test string is the longest one (250k characters long).

As it can be seen in figures 6.19 on page 80 and 6.20 on page 81, the prototype compared with the PLY generated parser shows rather good results. Despite the fact that on single core it is slower than PLY (250s vs 176s, speedup of ~ 0.7), already with two cores it performs better. It can be seen that the maximum speedup is obtained with 16 cores, point where our parser is ~ 3.09 times faster than the PLY generated one. As expected, our implementation with one core is slower than the PLY generated one. This, as previously said, is not a fault of the algorithm itself, but rather a poor implementation that has the only purpose of testing the algorithm and therefore does not comprehend any optimization at all.

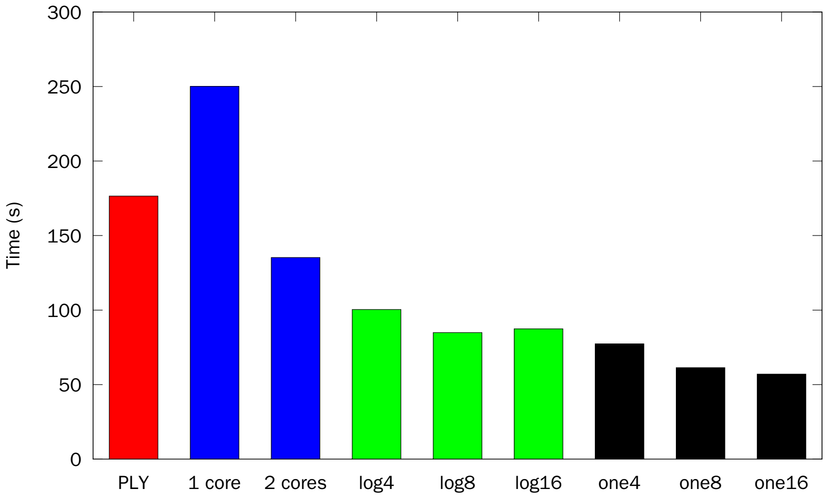


Figure 6.20: Python: Execution time of the various thread configurations compared to the PLY generated parser.

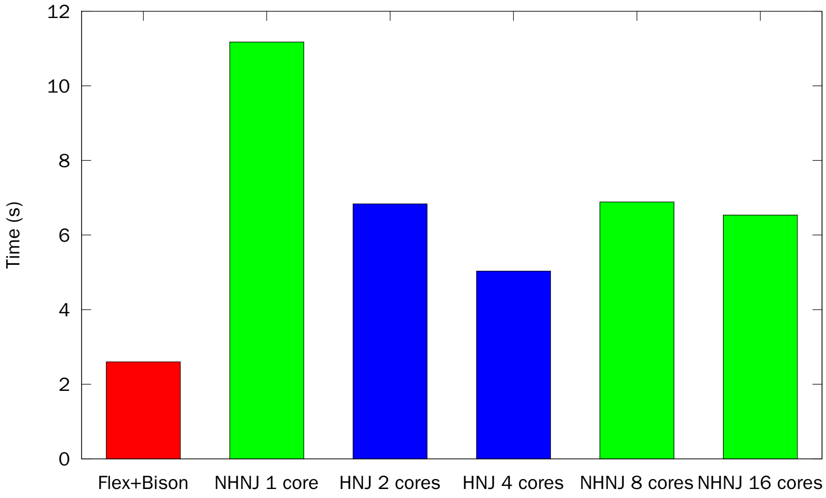


Figure 6.21: C: Execution time of the various thread configurations compared to the Flex and Bison generated parser.

6.3.2 C implementation compared with Flex and Bison

As it can be seen in figure 6.21 on page 81, the C implementation compared with the Flex and Bison generated parsers shows generally worse results. The Bison generated parser is a single threaded program that, as we said, implements an optimized LALR(1) parser. In the figure all the execution times are taken parsing the 10M characters long string; Bison on average took $\sim$ 2.6s to completely analyze the input. In comparison, the best performances for each threads configurations of our implementation are shown as the green and blue bars, where the green ones come from the version that does not use neither Judy nor Hoard, while the blue ones come from the version which employs only the Hoard allocator. On single thread, as it is now, our parser is $\sim$ 4.3 times slower than the Bison generated one, while our best performing configuration (*i.e.* using Hoard with 4 threads) we are $\sim$ 1.9 times slower. As it can be noticed, none of the configurations come from the version of the implementation which uses Judy arrays. This is another proof of the fact that our parser does not enjoy advantages from the usage of Judy arrays and justifies their removal. It is expected that with further development of the version of the implementation without Judy better results in terms of scalability and overall performance will be reached, since, as we mentioned, there is no theoretical reason for an operator precedence parser to perform worse than an LR one.

Chapter 7

Conclusion

In this thesis we studied a rather old paradigm of parsing, the operator precedence parser, and the theoretical properties of the grammars it could recognize, *i.e.* Floyd grammars. From this study the idea of a parallel parser was born and from it the main work of this thesis. We studied and developed a parallel parsing approach employing FGs, which first took the form of a Python prototype. After having verified the correctness of such an approach by mean of a suite of tests and after having obtained first encouraging results, we then developed a faster, more performing C implementation. This implementation displayed the same general behavior of the Python prototype, showing some degree of parallelism and scalability. Despite the fact that test results confirm that it is not yet at the level of current state of the art traditional parsers and, more importantly, its scaling capability is rather low, we think that it can greatly improve with further research and development.

Such a research could and should be focused on two main areas: optimizations on the implementation and optimizations on the algorithm itself. As for the former, we did not talk deeply in this thesis of how the algorithm has been implemented and therefore it would make no sense to introduce where we think the implementation should be optimized now, but, generally speaking, we think that bottlenecks now come from the memory management and this will be a major concern in the months to come. As for the latter, we already mentioned the optimizations of the algorithm that we identified as possible as of now in section 5.3.

Lastly, future work related to work of this thesis, which are not necessarily optimizations, can be located in the following areas:

- a PArallel PArser GENerator for Operator grammars (PAPAGENO), *i.e.* an algorithm that, given a grammar and its semantics rules, automatically generates an optimized parallel parser;

- parallel lexing; as of now we use an external library to perform lexing which obviously is sequential; a possible future work is the development of a library that performs parallel lexing speculatively; to our understanding as of now, this could be done with heuristics in order to balance the job of lexing over different threads and on their termination, verify whether the results around the cutoff points are consistent or not;

- incremental parsing, *i.e.* given a parse tree and a modification on the input string, an incremental parser modifies the parse tree accordingly without rebuilding it from scratch. Thanks to the locality principle of FGs, we think possible for a parallel operator precedence parser to be employed in order to be able to manage concurrent modifications to the same tree.

Appendix A

Infix grammar excursus

In this appendix we present the original idea to approach the problem of parallel parsing using FGs. Its basic idea is to define the grammars of the prefixes, suffixes and infixes of a given grammar, and use them to parse the different parts of an input string, resulting in several parse trees, which need to be linked together to reconstruct the parse tree of the original string. The definitions of the different grammars are given in the following sections and are taken or followed from [1]. As we said, after having performed a pass of parsing using these grammars, the original approach involved the recombination of the parse subtrees into a complete one. Since this approach has been abandoned during development, this last phase has not been properly defined, and therefore will not be presented. All three definitions use the *Fischer normal form* in order to express FGs, therefore we will introduce its definition:

A FG is in Fischer normal form if it is invertible, the axiom S does not occur in any right hand side of any rule, no empty rule exists except possibly $S \to \varepsilon$, the other rules having S as left hand side are renaming, and no other renaming rules exist.

A.1 Prefix grammar

Let $G = (V_N, \Sigma, P, S)$ be a FG in Fischer normal form. Its prefix grammar is $G' = (V'_N, \Sigma, P', S')$ where V'_N is the disjoint union of two sets denoted V^C and V^S, defined as follows:

$$V^C = \{A^C | A \in V_N \setminus \{S\}\}$$
$$V^S = \{A^S | A \in V_N \setminus \{S\}\}$$

To build the rule set, we introduce two transformations α^C and α^S

$$\alpha^C, \alpha^S : (V_N \cup \Sigma)^* \to (V'_N \cup \Sigma)^*$$

on strings α of the form

$$\alpha = x_0 A_1 x_1 ... A_n x_n, x_i \in \Sigma^*$$

If $x_n \neq \varepsilon$ then

$$\begin{cases} \alpha^C & = x_0 A_1^C x_1 ... A_n^C x_n \\ \alpha^S & = \text{undefined} \end{cases}$$

If $x_n = \varepsilon$ then

$$\begin{cases} \alpha^C & = x_0 A_1^C x_1 ... A_n^C \\ \alpha^S & = x_0 A_1^C x_1 A_2^C x_2 ... A_n^S \end{cases}$$

We also define the transformation $PREF(\alpha)$ as

$$\begin{aligned} PREF(\alpha) = & \left\{\beta_i^S | \beta_i \neq \varepsilon \text{ is prefix of } \alpha\right\} \cup \\ & \left\{\beta_i^C | \beta_i \neq \varepsilon \text{ is prefix of } \alpha \wedge \beta_i \in V^* \Sigma\right\} \end{aligned}$$

Then, the new rule set P' contains

$$S' \to A^S \text{ for every rule } S \to A \in P$$
$$A^C \to \alpha^C \text{ for every rule } A \to \alpha \in P$$
$$A^S \to PREF(\alpha) \text{ for every rule } A \to \alpha \in P$$

A.2 Suffix grammar

The construction of the suffix grammar proposed here is symmetric to the construction of the prefix grammar. Let $G = (V_N, \Sigma, P, S)$ be a FG in Fischer normal form. Its suffix grammar is $G' = (V'_N, \Sigma, P', S')$ where V'_N is the disjoint union of two sets denoted V^C and V^P, defined as follows:

$$V^C = \{A^C | A \in V_N \setminus \{S\}\} V^P = \{A^P | A \in V_N \setminus \{S\}\}$$

To build the rule set, we introduce two transformations α^C and α^P

$$\alpha^C, \alpha^P : (V_N \cup \Sigma)^* \rightarrow (V'_N \cup \Sigma)^*$$

on strings α of the form

$$\alpha = x_0 A_1 x_1 ... A_n x_n, x_i \in \Sigma^*$$

If $x_0 \neq \varepsilon$ then

$$\begin{cases} \alpha^C & = x_0 A_1^C x_1 ... A_n^C x_n \\ \alpha^P & = \text{undefined} \end{cases}$$

If $x_0 = \varepsilon$ then

$$\begin{cases} \alpha^C & = A_1^C x_1 ... A_n^C x_n \\ \alpha^P & = A_1^P x_1 A_2^C x_2 ... A_n^C x_n \end{cases}$$

We also define the transformation $SUFF(\alpha)$ as

$$\begin{aligned} SUFF(\alpha) = & \left\{ \beta_i^P | \beta_i \neq \varepsilon \text{ is suffix of } \alpha \right\} \cup \\ & \left\{ \beta_i^C | \beta_i \neq \varepsilon \text{ is suffix of } \alpha \wedge \beta_i \in \Sigma V^* \right\} \end{aligned}$$

Then, the new rule set P' contains

$$\begin{aligned} S' \rightarrow A^P & \text{ for every rule } S \rightarrow A \in P \\ A^C \rightarrow \alpha^C & \text{ for every rule } A \rightarrow \alpha \in P \\ A^P \rightarrow SUFF(\alpha) & \text{ for every rule } A \rightarrow \alpha \in P \end{aligned}$$

A.3 Infix grammar

The construction of the infix grammar is similar to the prefix and suffix ones, and it uses the definitions of $PREF(\alpha)$ and $SUFF(\alpha)$ presented in section A.1 and A.2. Let $G = (V_N, \Sigma, P, S)$ be a FG in Fischer normal form. Its infix grammar is $G' = (V_N', \Sigma, P', S')$, where V_N' is the disjoint union of four sets denoted V^C, V^I, V^P and V^S, defined as follows:

$$V^C = \{A^C | A \in V_N \setminus \{S\}\}$$
$$V^I = \{A^I | A \in V_N \setminus \{S\}\}$$
$$V^P = \{A^P | A \in V_N \setminus \{S\}\}$$
$$V^S = \{A^S | A \in V_N \setminus \{S\}\}$$

To build the rule set, we introduce four transformations α^C, α^I, α^P and α^S

$$\alpha^C, \alpha^I, \alpha^P, \alpha^S : (V_N \cup \Sigma)^* \to (V_N' \cup \Sigma)^*$$

on strings α of the form

$$\alpha = x_0 A_1 x_1 ... A_n x_n, x_i \in \Sigma^*$$

If $x_0 \neq \varepsilon \wedge x_n \neq \varepsilon$ then

$$\begin{cases} \alpha^C = x_0 A_1^C x_1 \dots A_n^C x_n \\ \alpha^P = \text{undefined} \\ \alpha^S = \text{undefined} \\ \alpha^I = \text{undefined} \end{cases}$$

If $x_0 \neq \varepsilon \wedge x_n = \varepsilon$ then

$$\begin{cases} \alpha^C = x_0 A_1^C x_1 \dots A_n^C \\ \alpha^P = \text{undefined} \\ \alpha^S = x_0 A_1^C x_1 \dots A_n^S \\ \alpha^I = \text{undefined} \end{cases}$$

If $x_0 = \varepsilon \wedge x_n \neq \varepsilon$ then

$$\begin{cases} \alpha^C = A_1^C x_1 \ldots A_n^C x_n \\ \alpha^P = A_1^P x_1 \ldots A_n^C x_n \\ \alpha^S = \text{undefined} \\ \alpha^I = \text{undefined} \end{cases}$$

If $x_0 = \varepsilon \wedge x_n = \varepsilon$ then

$$\begin{cases} \alpha^C = A_1^C x_1 \ldots A_n^C \\ \alpha^P = A_1^P x_1 \ldots A_n^C \\ \alpha^S = A_1^C x_1 \ldots A_n^S \\ \alpha^I = A_1^P x_1 \ldots A_n^S \end{cases}$$

We also defined the transformation $INF(\alpha)$ as

$$\begin{aligned} INF(\alpha) = &\left\{\beta_i^I | \beta_i \neq \varepsilon \text{ is infix of } \alpha\right\} \cup \\ &\left\{\beta_i^C | \beta_i \neq \varepsilon \text{ is infix of } \alpha \wedge \beta_i \in \Sigma V^* \Sigma\right\} \end{aligned}$$

Then, the new rule set P' contains

$$\begin{aligned} S' \rightarrow A^I &\text{ for every rule } S \rightarrow A \in P \\ A^C \rightarrow \alpha^C &\text{ for every rule } A \rightarrow \alpha \in P \\ A^P \rightarrow SUFF(\alpha) &\text{ for every rule } A \rightarrow \alpha \in P \\ A^S \rightarrow PREF(\alpha) &\text{ for every rule } A \rightarrow \alpha \in P \\ A^I \rightarrow INF(\alpha) &\text{ for every rule } A \rightarrow \alpha \in P \end{aligned}$$

Appendix B

Support algorithms

In this appendix we will present two algorithms that are used to build the support data structures used by the parsing process. In particular, we will see the reduction tree generation algorithm (section B.1) and the reduction set computation algorithm (section B.2).

B.1 Reduction tree generation

As we have seen in section 4.1.1, the reduction tree is a data structure that is used by the parser in order to recognize which right hand side is the correct one for a given handle. The data structure is initialized when the algorithm is started and, since it only depends on the grammar, its construction can be performed offline, *i.e.* before the analysis of an input string, and can be kept not only between different iterations but theoretically even between different strings, provided that the grammar remains unaltered. As it is implemented now, there is still no possibility to store the reduction tree and to reload it, therefore it is computed every time the parser starts.

The construction of the reduction tree can be seen in algorithm 5 on page 91. Since the rules recognition is done from right to left, the reduction tree has to support the reading of the rules in inverse order. The construction is based on the analysis of each right hand side in inverse order: starting from the root of the tree and the last token of the rhs, the algorithms tries to build a node for each token until it reaches the end of the rule, when it links the final node with the corresponding lhs and the semantic function used for the correct reduction. Since it is possible for multiple rules to have similar paths, nodes are reused for several rules and new nodes are created only when necessary.

Algorithm 5 Reduction tree generation algorithm

```
root := new_tree
for all grammar rules do
  node := root
  for all token in inverted rhs do
    temp := get_son_with_label(node, token)
    if temp $\neq$ NULL then
      node := temp
    else
      node := create_son_with_label(node, token)
    end if
  end for
  link_lhs_and_semantic_function(node)
end for
return root
```

We can observe that the complexity of the algorithm is $O(nm)$, where n is the number of production rules in the grammar, and m is the average number of tokens per rhs. We do not exclude the existence of more efficient versions of the algorithm, but, since the computation of the reduction tree is not a burden on the parsing process, we do not believe this to be a particular issue.

An example of reduction tree for the usual parenthesized arithmetic expressions grammar can be seen in figure 4.2 on page 37.

B.2 Reduction sets computation

As we have said in section 4.1.2, reduction sets are used to manage immediate rewrite rules after normal reductions. Let $G = (V_N, \Sigma, P, S)$ be a FG, we define the reduction set for each nonterminal $A \in V_N$ with respect to grammar G, *i.e.* $\mathit{reduction}_G(A)$, as follows:

$$\mathit{reduction}_G(A) = \{B \in V_N \mid B \overset{*}{\Rightarrow} A\}$$

The algorithm to build the reduction sets (which is shown in algorithm 6 on page 94) is composed of two different phases: a rewrite sets generator (lines 1-20) and the actual reduction sets generator (lines 21-28). The rewrite sets generator is used to build, for each nonterminal of the grammar, the set of nonterminals to which it can be rewritten to; in particular it builds these sets by analyzing both direct and indirect rewrite rules. This is done by first creating, for each $A \in V_N$, its corresponding direct rewrite set $\mathit{rewrite}_G(A)$ as

$$\mathit{rewrite}_G(A) = \{B \in V_N \mid \exists A \to B \in P\}$$

The direct rewrite sets are then iteratively used to create indirect rewrite sets as follows:

$$\begin{aligned}\mathit{rewrite}^k_G(A) = \mathit{rewrite}^{k-1}_G(A) \cup \\ \{B \in \mathit{rewrite}^{k-1}_G(C) \mid \forall C \in \mathit{rewrite}^{k-1}_G(A)\}\end{aligned}$$

Which intuitively means that on each iteration the rewrite set of nonterminal A is enriched with the entire rewrite set of each nonterminal C in the rewrite set of A. Thus, the resulting rewrite set of terminal A is:

$$\mathit{rewrite}_G(A) = \{B \in V_N \mid A \overset{*}{\Rightarrow} B\}$$

After having computed all the rewrite sets, the algorithm actually creates for each nonterminal symbol its corresponding reduction set, $\mathit{reduction}_G(A)$, as:

$$\mathit{reduction}_G(A) = \{A\} \cup \{B \in V_N \mid A \in \mathit{rewrite}_G(B)\}$$

For instance, considering the usual parenthesized arithmetic expressions grammar, the rewrite sets for the nonterminal symbols are

$$\begin{aligned}
\mathit{rewrite}_G(S) &= \{E, T, F\} \\
\mathit{rewrite}_G(E) &= \{T, F\} \\
\mathit{rewrite}_G(T) &= \{F\} \\
\mathit{rewrite}_G(F) &= \emptyset
\end{aligned}$$

from which the following reduction sets can be computed as described above:

$$\begin{aligned}
\mathit{reduction}_G(S) &= \{S\} \\
\mathit{reduction}_G(E) &= \{S, E\} \\
\mathit{reduction}_G(T) &= \{S, E, T\} \\
\mathit{reduction}_G(F) &= \{S, E, T, F\}
\end{aligned}$$

Therefore, whenever an F is recognized, the parser will stack the set $\{S, E, T, F\}$ because those are all the nonterminals that can immediately rewrite into F.

As for the reduction tree generator presented in section B.1, this algorithm was not written with performance in mind, because it is another part of the parser that can be executed offline and only once per grammar. As the previous one, also this data structure currently cannot be stored and reloaded but has to be computed on each run.

Algorithm 6 Reduction sets computation algorithm

```
rewrite := new_dictionary
repeat
  modified := false
  for all lhs do
    rules := get_direct_rewrite_rules(lhs)
    for all rule $\in$ rules do
      if rule $\notin$ rewrite[lhs] then
        rewrite[lhs] := rewrite[lhs] $\cup$ rule
        modified := true
      else
        for all temp $\in$ rewrite[rule] do
          if temp $\notin$ rewrite[lhs] then
            rewrite[lhs] := rewrite[lhs] $\cup$ temp
            modified := true
          end if
        end for
      end if
    end for
  end for
until modified is false
reduction := new_dictionary
for all nterm $\in V_N$ do
  reduction[nterm] := reduction[nterm] $\cup$ nterm
  for all token $\in$ rewrite[nterm] do
    reduction[token] := reduction[token] $\cup$ nterm
  end for
end for
return reduction
```

Glossary

context The couple of terminal symbols preceding and immediately following a substring.

formal grammar A tuple $< V_N, \Sigma, P, S >$, where V_N is the nonterminal alphabet, Σ is the terminal alphabet, P is the set of syntactic production rules and $S \in V_N$ is the axiom.

handle A series of tokens that identify a right hand side of a grammar production, with precedence relations in the form $\lessdot \doteq \cdots \doteq \gtrdot$.

immediate rewrite rule A grammar rule in the form $N \rightarrow M$ where $N, M \in V_N$.

infix An infix of a string $T = t_0, t_1 \ldots t_n$ is a string $I = t_m, t_{m+1} \ldots t_k$, where $k \leq m \leq n$.

parallel computing A paradigm of computation in which many different calculations are performed simultaneously.

parsing Analysis of a sequence of tokens s, one at a time, with respect to a given formal grammar G, determining the syntactical structure of the string, in order to define whether it belongs to the language $L(G)$ generated by G.

prefix A prefix of a string $T = t_0, t_1 \dots t_n$ is a string $P = t_0, t_1 \dots t_m$, where $m \leq n$.

regular language A formal language which can be expressed by a formal regular expression.

speedup $S_p = \frac{T_1}{T_p}$ where T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors.

string A finite sequence of symbols that are chosen from a set or alphabet.

suffix A suffix of a string $T = t_0, t_1 \dots t_n$ is a string $S = t_{n-m}, t_{n-m+1} \dots t_n$, where $m \leq n$.

token Either a terminal or a nonterminal symbol, which is produced from a string of characters during the lexing process.

Bibliography

[1] Stefano Crespi-Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*, pages 214–226. Springer, 2010.

[2] http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[3] http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] Stefano Crespi Reghizzi. *Formal Languages and Compilation.* Springer, 2009.

[6] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10:316–333, July 1963.

[7] http://googleresearch.blogspot.com/2010/12/6-million-to-faculty-in-q4-research.html.

[8] http://www.gnu.org/software/bison/manual/bison.html.

[9] M. Dennis Mickunas and Richard M. Schell. Parallel compilation in a multiprocessor environment (extended abstract). In *Proceedings of the 1978 annual conference*, ACM '78, pages 241–246, New York, NY, USA, 1978. ACM.

[10] J. Cohen, T. Hickey, and J. Katcoff. Upper bounds for speedup in parallel parsing. *J. ACM*, 29:408–428, 1982.

[11] J. Cohen and S. Kolodner. Estimating the speedup in parallel parsing. *IEEE Transactions on Software Engineering*, 11:114–124, 1985.

[12] D. Sarkar and N. Deo. Estimating the speedup in parallel parsing. *IEEE Transactions on Software Engineering*, 16:677–683, 1990.

[13] Lu Wei, Chiu Kenneth, and Pan Yinfei. A parallel approach to xml parsing. In *In The 7th IEEE/ACM International Conference on Grid Computing*, 2006.

[14] Pan Yinfei, Lu Wei, Zhang Ying, and Chiu Kenneth. A static load-balancing scheme for parallel xml parsing on multicore cpus. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID 2007.

[15] Yu Wu, Qi Zhang, Zhiqiang Yu, and Jianhui Li. A hybrid parallel processing for xml parsing and schema validation. In *Proceedings of Balisage: The Markup Conference 2008*, Balisage Series on Markup Technologies, 2008.

[16] Bhavik Shah, Praveen R. Rao, Bongki Moon, and Mohan Rajagopalan. A data parallel algorithm for xml dom parsing. In *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies*, XSym '09, pages 75–90, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer New York, 2008.

[18] http://wiki.python.org/moin/GlobalInterpreterLock.

[19] http://www.dabeaz.com/ply/.

[20] http://flex.sourceforge.net/.

[21] http://judy.sourceforge.net/.

[22] http://json.org/.

[23] http://www.hoard.org/.