THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

GF Runtime System

Krasimir Angelov

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2009

GF Runtime System
Krasimir Angelov
ISSN 1652-876X

Technical Report no. 63L
Department of Computer Science and Engineering
Language Technology Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, 2009

Abstract

Natural languages have been subject of studies for centuries and are hot topic even today. The demand for computer systems able to communicate directly in natural language places new challenges. Computational resources like grammars and lexicons and efficient processing tools are needed.

Grammars are described as computer programs in declarative domain specific languages. Just like any other programming language they require mature compilers and efficient runtime systems.

The topic of this thesis is the runtime system for the Grammatical Framework (GF) language. The first part of the thesis describes the semantics of the Portable Grammar Format (PGF). This is a low-level format which for GF plays the same role as the JVM bytecode for Java. The representation is designed to be as simple as possible to make it easy to write interpreters for it. The second part is for the incremetal parsing algorithm in PGF. The parser performance was always a bottle neck in GF until now. The new parser is of orders of magnitude faster than any of the previous implementations. The last contribution of the thesis is the development of the Bulgarian resource grammar. This is the first Balkan language in the resource library and the first open-source grammar for Bulgarian. The grammar development was also an important benchmark for the development of the parsing algorithm.

Contents

Chapter 1

Introduction

The first attempts to understand the rules that govern the human language could be traced back in the ancient history. Around the 6th c. BC Yaska and latter Panini in 4th c. BC developed the first grammars for Sanskrit. The work was continued by Katyayana and Patanjali in 2nd c. BC and Pingala in ca. 200 BC. Both Yaska and Panini refer to Shaakataayana, an old grammarian who have worked before, which suggests that there were even earlier interests in the work.

In Europe, the earliest grammar emerged in Greece, 3rd c. BC with "Art of Grammar" by Dionysius Thrax. Later Roman authors followed the same model and created grammars for Latin.

In modern times the interest and the need to have systematic descriptions of modern languages is still alive. Furthermore the development of computer technologies placed the grammarians work in a different context. While the book based grammars are still widely used by scholars or just language learners, the desire to have machine processable formal grammars emerged. This new demand also placed new challenge. The book based grammars are necessary implicit, some details are often skipped and left to the reader's intuition. While this is the right approach for book it is simply not possible if the grammar is to be used by a machine. In the last decades a number of linguistic formalisms were developed for the formal description of natural languages. Among them the most widely known are Head-driven Phrase Structure Grammar (HPSG), Combinatory Categorial Grammar (CCG) and Lexical Functional Grammar (LFG).

This thesis is devoted to the Grammatical Framework (GF, Ranta 2004a) formalism. GF could be seen and described from many perespectives. First this is a grammar formalism in the same spirit as HPSG, CCG and LFG but it is grounded on a different mathematical theory - Martin Löf's Type Theory (Martin-Löf 1984).

Martin Löf's theory is also the basis of a number of computer-based proof systems such as Agda, ALF, Coq, Epigram, LEGO, NuPRL and Twelf and in this aspect GF is also a proof system. The meeting point is the notion of concrete and abstract syntax in GF. The abstract syntax is a logical framework similar to the frameworks that are used in the proof systems and the concrete syntax is a mapping between the logical representation and its surface form in some language. From software point of view GF is also programming language specialized in the development of natural language grammars. It is implemented as system which consists of a compiler and a runtime system for the GF programming language.

The runtime system of GF is the primary topic of this thesis. For a long time the compiler and the interpreter for GF were tightly coupled together. Someone who wanted to build an application had to carry along with the application a large part of the GF compiler as well. While this has a maintenance cost there is also another more important problem. The GF system is entirely written in Haskell - a good choice for a compiler since the functional languages are well suited for compilers development. However the same choice makes it difficult to integrate GF grammars in existing applications which are written in different languages.

Starting with GF version 3.0 there is a clear separation between compiler and a runtime system. The runtime is a compact Haskell library which is only about 10% of the whole system. The representation of the grammars in the runtime is also significantly simplified to make it easy and efficient to implement new interpreters from scratch in other programming languages. Proof of concept interpreters were implemented in JavaScript, Prolog, C and Java. There is also a portable file format which allows grammars compiled by the GF compiler to be used by the interpreter independently on the implementation language. Chapter 2 describes the semantics of the Portable Grammar Format (PGF) and is based on Angelov et al. (2008).

The grammars serve two major purposes: natural language generation and natural language parsing. GF is a generation oriented formalism in the sense that the grammarian describes how the natural language is produced from some abstract representation. Despite this both parsing and generation are possible thanks to the compositionality of the grammar rules. However the parsing problem is much more computationaly intensive. This is not limitation of the framework but a general observation for almost all grammar formalisms. One well known exception is the class of LR(n) grammars where both the parsing and the generation could be made in linear time. Unfortunately the LR(n) class is far too weak to describe natural languages.

The descriptive power of GF was investigated in Ljunglöf (2004) and was

proved that in absence of dependent types and high-order types, GF is equivalent to Parallel Multiple Context-Free Grammar (PMCFG). PMCFG as a formalism was known for a long time (Seki et al. 1991) and was well studied theoretically. Unfortunately it never become popular as a practical tool for implementing natural language processing systems. Still the closely related but weaker MCFG[1] formalism was used in computational biology for description of pseudoknot structures (Kato et al. 2006). Different parsing algorithms for MCFG were proposed in Nakanishi et al. (1997), Ljunglöf (2004) and Burden and Ljunglöf (2005) but none of them supports the full power of PMCFG.

Many different parsing algorithms were used in GF. The earliest one was based on context-free approximation. First the context-free backbone of the grammar is extracted and after that it is used to parse the input sentence. Since the new grammar is overgenerating it will produce many redundant parse trees. It could even accept sentences which are not acceptable by the original grammar. For that reason, a second post processing phase is needed which checks the trees for consistency with the real grammar. Since the approximating grammar is usually very ambiguous, the parsing is slow and the number of syntax trees is huge. This made the earliest parser useless for large grammars. Later the algorithms from Ljunglöf (2004) were implemented. Since they use formalism (MCFG) which is closer to GF they are significantly faster. Still grammars of the size of the grammars in the GF Resource Grammar Library (Ranta 2008) were too big challenge. The primary reason was that most of them could not even be compiled to a format suitable for parsing in a reasonable amount of memory. Even if it was possible, the incredible number of rules in the grammar will make the usage of the parser impractical. The problem was solved with the new parser introduced in GF 3.0. It uses PMCFG directly, so no post processing is needed and the set of trees returned is exactly the set that is consistent with the grammar. The compilation to PMCFG also turned out to be more compact than to MCFG. The parsing algorithm is the subject of chapter 3 and is based on the paper (Angelov 2009).

The primary motivation for the development of the new parsing algorithm was to make it more efficient. In fact it was born as a result of analyzing the existing algorithms that were in use in GF. The outcome is very satisfactory. In addition to being more efficient it has two other important characteristics: this is the first published algorithm which supports PMCFG directly and this is the first PMCFG/MCFG algorithm which is incremental.

[1] MCFG is the same as PMCFG except that some further restrictions are inforced on the grammar rules. See chapter 2 for information

Theoretically the algorithm has at most polynomial time and space complexity but in the practical experiments it shows almost linear complexity. The experiment in chapter 3 (figure 3.3) shows empirical measurements with twelve languages from the Resource Grammar Library. In all cases the time grows linearly with the number of tokens in the sentence. The constant factor varies with the language. In the experiment German has the highest constant factor while English and Swedish have almost the same complexity and are very close to the x-axis. Even for German the statistics show that a sentence of 30 tokens could be recognized for about a second. The linear behaviour is preserved for the other languages also (figure 3.4).

For simple grammars it is even possible to compute the time complexity analytically. An example is the exponential language $\{a^{2^m} \mid m \geq 0\}$. In GF it is defined by the grammar:

$$\begin{aligned} &\textbf{cat } S \\ &\textbf{fun } s : S \to S \\ &\quad z : S \end{aligned}$$

$$\begin{aligned} &\textbf{lincat } S = Str \\ &\quad \textbf{lin } s\ x\ = x\ ++\ x \\ &\quad z = \text{"}a\text{"} \end{aligned}$$

Here S is a syntactic category and s and z are two functions. The function z does not have any arguments and just returns a string with one symbol a. The other function s takes as argument a string and returns another string which is the original string concatenated by itself. In this way the expression $s\ (s\ z)$ will produce the string $aaaa$. The parser takes a string and tries to find tree (expression) which produces the same string. The string is processed strictly from left to right. After each token, only the set of trees that are consistent with the already consumed input are kept. For example, after the first token a the current candidate is z. After the second token, the current input is aa and the candidate is $s\ z$. When the third token is consumed, the candidate is $s\ (s\ z)$ and the parser knows that there should be one more symbol a for the sentence to be complete. In general after the k-th token the candidate is a tree of depth $\log_2 k$. Since in the worst case the parser traverses the whole tree after each token, the time needed to accept a new token is

proportional to $\log_2 k$. The total time needed to parse the whole string is then:

$$\sum_{k=1}^{n} \log_2 k = \log_2 n! \leq n \log_2 n$$

This approximation is very close to the best possible parsing time for this language. The most efficient algorithm is to compute the length of the string and then to check whether its length is an exponent of two. The computation of the length takes n steps and after that the computation of the binary logarithm needs $\log_2 n$ more steps. The total complexity will be $n + \log_2 n$. The difference should be attributed to the generality of the PMCFG algorithm compared to the best algorithm which works only for the exponential language. This theoretical result was also verified by empirical measurement with a string of one milion tokens.

In the computational linguistic a debated but common assumption is that all existing natural languages could be described with the class of languages called mildly context-sensitive (Aravind 1985). This class is characterized by three conditions:

- A parser with polynomial complexity must exist.
- The language must have constant growth. If we sort all strings in the language by increasing length then the differences between the lengths of any two consequent strings must be bounded by some constant.
- The language should admit limited cross-serial dependencies, allowing grammatical agreement to be enforced between two arbitrarily long subphrases;

The exponential language just described above clearly violates the second condition because the length of the strings grows exponentially. This kinds of languages look very ”unnatural“ because the natural languages does not seem to exhibit this behaviour. Since it is desirable for the computer system to be as efficient as possible, it is prefferable to use grammar formalisms that have just enough expressiveness and so not to loose efficiency to support features that are not going to be used in practice. From this point of view the usage of PMCFG is not justified and MCFG is to be prefered because it is mildly context-sensitive formalism.

However more recent studies showed that in some languages exceptions could be found which are more than mildly context-sensitive. For example in Mandarin Chinese the numerals name system (Radzinski 1991) and the yes/no questions (Stabler 2004) make the syntax not mildly context-sensitive. In a similar way

the multiple case marking in Old Georgian (Michaelis and Kracht 1996) makes it not mildly context-sensitive as well. Both the numerals name system in Mandarin Chinese and the multiple case marking in Old Georgian has fragments which could be exemplified by the language:

$$\{ab^k ab^{k-1} \ldots ab^2 ab \mid k > 0\}$$

It has been shown that this language is not mildly context-sensitive but it is trivial to define it as GF grammar [2]:

$$\begin{aligned}
&\textbf{cat } N, S \\
&\textbf{fun } z : N \\
&\quad s : N \to N \\
&\quad c : N \to S \\
\\
&\textbf{lincat } N = \{s1, s2 : Str\} \\
&\quad S = Str \\
&\quad \textbf{lin } z = \{s1 = \text{"}a\text{"} ++ \text{"}b\text{"};\ s2 = \text{"}b\text{"} ++ \text{"}b\text{"}\} \\
&\quad s\ x = \{s1 = \text{"}a\text{"} ++ x.s2 ++ x.s1;\ s2 = \text{"}b\text{"} ++ x.s2\} \\
&\quad c\ x = x.s1
\end{aligned}$$

Note how the expression $x.s2$ is used twice in the definition of function s. This nonlinearity is precisely what makes the language not mildly context-sensitive and is the same trick that was used to define the exponential language. The yes/no questions in Mandarin Chinese could be expressed by providing the answer choices in the question and this requires the verb phrase to be repeated:

Zhangsan	like	play	basketball	not	like	play	basketball
Zhangsan	ai	da	lanqiu	bu	ai	da	lanqiu

Does Zhangsan like to play basketball?

This is another example of nonlinearity in the language. Note that if the phrase is not exactly repeated then the expression has completely different meaning:

[2] it could be proved that this language is recognizable in linear time by the GF parser

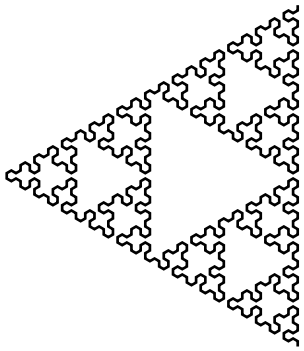
Zhangsan	like	play	basketball	not	like	play	volleyball
Zhangsan	ai	da	lanqiu	bu	ai	da	paiqiu

Zhangsan likes to play basketball, not to play volleyball

Another kind of ”artificial“ but ”natural“ languages that could be described by PMCFG is the Lindenmayer's system (Lindenmayer 1968). This is a class of artificial languages which are used to describe structures that appear naturally in the physical world. Initially Lindenmayer proposed the system for description of the growth of the red algae. Later the system was used for other kinds of phenomenons like the growth of crystals, the internal structure of musics or just for fractal structures. An example grammar that produces the Sierpinski's triangle is shown on figure 1.1. It produces PostScript code which when interpreted renders the triangle. Using the GF parser, graftals could not only be generated but also analized.

Another very practical advantage of using PMCFG instead of MCFG is that this leads to more compact grammars. For example the English resource grammar is only 409 KB when it is compiled to PMCFG and 944 KB when it is compiled to MCFG. In the same time the Bulgarian grammar is 3 MB in PMCFG and far too big to fit in 2 GB memory on 32 bits computer. The reason is that the algorithm which approximates PMCFG to MCFG (Ljunglöf 2004) leads to exponential grow of the grammar size. In fact it was impossible before, for most of the languages, to compile the grammars to format suitable for parsing (i.e. MCFG). This is not a problem if the grammar is used only as a library but having a parser which works even with big grammars is still an advantage. The grammars were made even more compact with the introduction of the common subexpression elimination for PMCFG (section 2.4.3).

Another prominent feature of the parsing algorithm is that it is incremental. The incrementality means that the parser takes the tokens one by one from left to right, and after each token only the set of trees that are consistent with the current prefix are kept in the chart. Since the algorithm uses top-down prediction this also makes it possible to predict the set of next possible tokens, taking into account the existing prefix. This is the basis of the new authoring help-system in GF 3.0 (Bringert et al. 2009, Ranta and Angelov 2009). The usage of authoring help-systems is popular in the controlled language community. A controlled language is a language which mimics some existing natural language but has simpler syntax and semantics. The advantage of the controlled language is that it could be



```
abstract Graftal = {
cat N; S;
fun z : N ;
    s : N -> N ;
    c : N -> S ;
}

concrete Sierpinski of Graftal = {
lincat N = {a : Str; b : Str} ;
lincat S = {s : Str} ;

lin z = {a = A; b = B} ;
lin s x = {a = x.b ++ R ++ x.a ++ R ++ x.b; b =x.a ++ L ++ x.b ++ L ++ x.a} ;
lin c x = {s = "newpath 300 550 moveto" ++ x.a ++ "stroke showpage"} ;

oper A : Str = "0 2 rlineto" ;
oper B : Str = "0 2 rlineto" ;
oper L : Str = "+60 rotate" ;
oper R : Str = "-60 rotate" ;

}
```

Figure 1.1: Sherpinski triangle

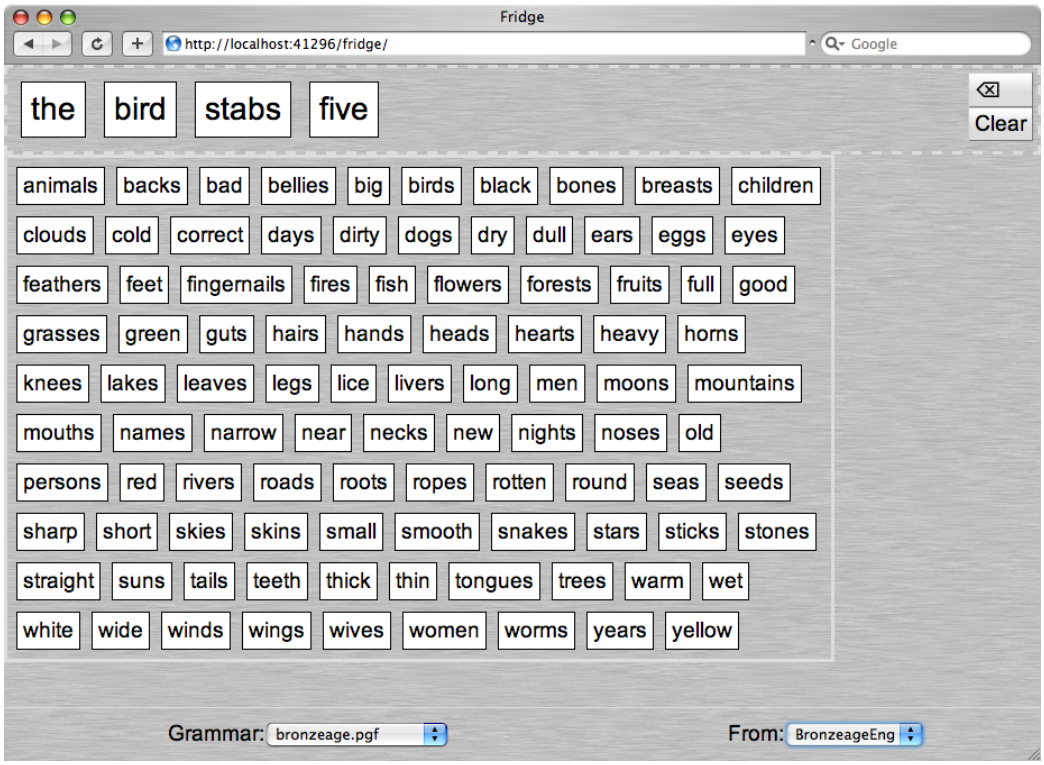


Figure 1.2: Fridge Poetry Screenshot

understood even from people without special training. In the same time, since the language has well defined formal syntax and semantics, it is also machine processable. In this aspect the controlled language is like a formal language but its natural syntax makes it easier to comprehend. While it is easy to understand content in a controlled language the authoring of a new content is not so trivial. Since the syntax resembles natural language it is tempting to use syntactic constructs which exist in the host language but not in the controlled sublanguage. Here is where the authoring help-systems comes in. They have to guide the user by suggesting which expressions are possible and which are not. The help-system in GF uses the word completion for this purpose (figure 1.2).

Chapter 4 departs from the topic of the GF runtime system and talks about the development of the Bulgarian resource grammar (Angelov 2008). The grammar writting is a great experience which the author wish to recomend to everyone who enters the computational linguistics field. For those who come from the computer science side, this is the best way to learn about the linguistic concepts. Those who

come from the linguistics side will have to learn a bit of programming to be able to write computational grammars. It is especially rewarding to try to formalize the rules of your own native language. This might give you new perespective to the language that you speak every day.

The Bulgarian grammar is important contribution to GF in two ways: first it adds one more language to the resource grammar library and second it was important benchmark. In the early stages the grammar was always on the border between what could be compiled to PMCFG and what could not. This motivated the implementation of optimizations in the compiler which after that were beneficial for all other languages. Later on Finnish and the Romance languages were important benchmarks too but in the beginning the blow up that these languages caused was too high to start with.

This is also the first open-source grammar for Bulgarian. The open-source movement lead to major advancement in the software technology but unfortunately it is not as popular in the linguistic community. The advantages are clear. The field is difficult enough by itself and still a lot of work is being repeated because the original data is not available. In the case with Bulgarian, the author was not able to find even proprietary grammars. Other resources like dictionaries, thesaurus, etc exist but most of them are not freely available. The only free resources that could be obtained are from two well known open-source projects: the English-Bulgarian-English dictionary - SADict and the spell checker for Open Office - BGOffice. The morphological paradigms in the resource library are actually extracted from BGOffice and later extended with information about the verb's aspect and the animacy of the nouns. This only shows the advantages of the open-source development, the existing data even if it is not complete is still useful because it could be accessed and extended by everyone. The grammar itself is not complete either in the sense that it do not describe every possible syntactic construction in the language but only those that are defined in the abstract syntax of the library. Despite this later on it could be extended with new rules by someone else.

In summary the content of the thesis is:

- Chapter 2 is based on the journal paper:

  > Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. Journal of Logic, Language and Information, accepted, 2008.

  This is a joint work with Björn Bringert and Aarne Ranta. My main contri-

bution in the paper is section 2.4.

- Chapter 3 is based on the conference paper:

  > Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In European Chapter of the Association for Computational Linguistics, 2009.

- Chapter 4 is based on the conference paper:

  > Krasimir Angelov. Type-theoretical bulgarian grammar. In GoTAL '08: Proceedings of the 6th international conference on Advances in Natural Language Processing, pages 52–64, Berlin, Heidelberg, 2008. Springer-Verlag.

Sections 2.4.3 and 3.6 in the thesis contain evaluation information that is more up to date than the corresponding papers.

Chapter 2

A Portable Grammar Format

2.1 Introduction

PGF (Portable Grammar Format) is a grammar formalism designed to capture the computational core of type-theoretical grammars, such as those written in GF (Grammatical Framework, Ranta 2004a). PGF thus relates to GF in the same way as JVM (Java Virtual Machine) bytecode relates to Java. While GF (like Java) is a rich language, whose features help the programmer to express her ideas on a high level of abstraction, PGF (like JVM) is an austere language, which is easy to implement on a computer and easy to reason about. The bridge between these two level, in both cases, is a compiler. The compiler gives the grammar writer the best of the two worlds: when writing grammars, she can concentrate on linguistic ideas and find concise expressions for them; when using grammars, she can enjoy efficient run-time performance and a light-weight run-time system, as well as integration in applications.

PGF was originally designed as a back-end language for GF, providing the minimum of what is needed to perform generation and parsing. Its expressive power is between context-free and fully context-sensitive (Ljunglöf 2004). Thus it can potentially provide a back end to numerous other grammar formalisms as well, providing for free a large part of what is involved in implementing these formalisms. In addition, PGF can be compiled further into other formats, such as language models for speech recognition systems (Bringert 2007a).

The most prominent characteristic of PGF (as well as GF) is **multilinguality**: a PGF grammar has an **abstract syntax**, which is a type-theoretical definition of tree structures. The abstract syntax is equipped with a set of **concrete syntaxes**,

which are reversible mappings of tree structures to strings. This grammar architecture is known as the Curry architecture, with a reference to Curry (1961). It is at the same time familiar from programming language descriptions, dating back to McCarthy (1962). While the Curry architecture was used by Montague (1974) for describing English, GF might be the first grammar formalism that exploits the multilingual potential of the Curry architecture.

Multilingual PGF grammars readily support translation and multilingual generation. They are also useful when writing monolingual applications, since the non-grammar parts of an application typically communicate with the abstract syntax alone. This follows the standard architecture of compilers, which use an abstract syntax as the format in which a programming language is presented to the other components, such as the type checker and the code generator (Appel 1997). Thus an application using PGF is easy to port from one target language to another, by just writing a new concrete syntax. The GF **resource grammar library** (Ranta 2008) helps application programmers by providing a comprehensive implementation of syntactic and morphological rules for 15 languages.

JVM is a general-purpose Turing-complete language, but PGF is limited to expressing grammars. Grammars can be seen as declarative programs performing tasks such as parsing and generation. In a language processing system, these tasks usually have to be combined with other functionalities. For instance, a question answering system reads input and produces output by using grammars, but the program that computes the answers from the questions has to be written in another language. **Embedded grammars** is a technique that enables programs written in another programming language to call PGF functionalities and also to manipulate PGF syntax trees as data objects. There are two main ways to implement this idea: interpreters and compilers. A PGF interpreter is a program written in a general-purpose language (such as C++, Haskell, or Java, which we have already written interpreters for). The interpreter reads a PGF grammar from a file and gives access to parsing and generation with the grammar. A PGF compiler translates PGF losslessly to another language (such as C, JavaScript, and Prolog, for which compilers have already been written). When we say that PGF is portable, we mean that one can write PGF interpreters and compilers for different host languages, so that the same PGF grammars can be used as components in applications written in these host languages.

The purpose of this paper is to describe the PGF grammar format and briefly show how it is used in applications. The PGF description is detailed enough to enable the reader to write her own PGF interpreters and compilers; but it is informal in the sense that we don't fully specify the concrete format produced by the cur-

rent GF-to-PGF compiler and implemented by the current PGF interpreters. For that level of detail, on-line documentation is more appropriate and can be found on the GF web page[1].

Section 2.2 gives a concise but complete description of the syntax and semantics of PGF. Section 2.3 gives a summary of the expressive power of PGF, together with examples illustrating its main properties. It also discusses some extensions of PGF. Section 2.4 describes the parser generation and sketches the parsing algorithm of PGF. Section 2.5 discusses the principal ways of using PGF in language processing applications, the compilation from PGF to other formats, and the compilation of PGF grammars from GF, which is so far the main way to produce PGF grammars. Section 2.6 gives a survey of actual applications running PGF and provides some data evaluating its performance. Section 2.7 discusses related work, and Section 2.8 gives a conclusion.

2.2 The syntax and semantics of PGF

2.2.1 Multilingual grammar

The top-most program unit of PGF is a **multilingual grammar** (briefly, grammar). A grammar is a pair of an **abstract syntax** $\mathscr{A}$ and a set of **concrete syntaxes** $\mathscr{C}_1, \ldots, \mathscr{C}_n$, as follows:

$$\mathscr{G} = < \mathscr{A}, \{\mathscr{C}_1, \ldots, \mathscr{C}_n\} >$$

2.2.2 Abstract syntax

An abstract syntax has a finite a set of **categories** and a finite set of **functions**. Categories are defined simply as unique identifiers, whereas functions are unique identifiers equipped with **types**. A type has the form

$$(\mathsf{C}_1, \ldots, \mathsf{C}_n) \to \mathsf{C}$$

where n may be 0 and each of $\mathsf{C}_1, \ldots, \mathsf{C}_n, \mathsf{C}$ is a category. These types are actually **function types** with **argument types** $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and **value type** C.

Here is an example of an abstract syntax, where we use the keyword **cat** to give the categories and **fun** to give the functions. It defines the categories S (sentence), NP (noun phrase), and VP (verb phrase). Sentences are formed by the

[1] `http://gf.digitalgrammars.com/`

Pred function. John is given as an example of a noun phrase and Walk of a verb phrase.

cat S; NP; VP
fun Pred : (NP, VP) → S
fun John : () → NP
fun Walk : () → VP

An **abstract syntax tree** (briefly, tree) is formed by applying the functions obeying their typings, as in simply typed lambda calculus. Thus, for instance, Pred (John, Walk), is a tree of type S, usable for representing the string *John walks*.

2.2.3 Concrete syntax

A concrete syntax has judgements assigning a **linearization type** to each category in the abstract syntax and a **linearization function** to each function. Linearization types are **tuples** built from **strings** and **bounded integers**. Thus we have the following forms of linearization types T:

$$\mathsf{T} ::= \mathsf{Str} \mid \mathsf{Int}_n \mid \mathsf{T}_1 * \ldots * \mathsf{T}_n$$

Linearization functions are **terms** t of these types, built in the following ways:

$$t ::= [\,] \mid \text{“foo”} \mid t_1 ++ t_2 \mid i \mid <t_1, \ldots, t_n> \mid t_1 \,!\, t_2 \mid \$_i$$

The first three forms are canonical for strings: the empty string $[\,]$, a token (quoted, like “foo”), and the **concatenation** $++$. Concatenation is associative, and it preserves the separation between tokens. The empty string is nilpotent with respect to concatenation. The canonical PGF representation of the string *John loves Mary*, if conceived as consisting of three tokens, is thus “John” $++$ “loves” $++$ “Mary”.

The form i, where i is a numeric constant $1, 2, \ldots, n$, is canonical for integers bounded by n. The form $<t_1, \ldots, t_n>$ is canonical for tuples, comprising a term t_i for each component type T_i in a tuple type.

The last two forms in the term syntax are non-canonical. The form $t_1 \,!\, t_2$ is the **projection** of t_2 from t_1. In order for the projection to be well-formed, t_1 must be a tuple and t_2 an integer within the bounds of the size of the tuple. The last form $\$_i$ is an **argument variable**. It is bound to a term given in a context, which, as we shall see, always consists of the linearizations of the subtrees of a tree.

Strings:

$$[\,]:\mathsf{Str} \quad \text{“foo”}:\mathsf{Str} \quad \frac{s,t:\mathsf{Str}}{s \mathbin{+\!\!+} t:\mathsf{Str}}$$

Bounded integers:

$$i:\mathsf{Int}_i \quad \frac{i:\mathsf{Int}_m}{i:\mathsf{Int}_n}\, m<n$$

Tuples:

$$\frac{t_1 \;:\; T_1 \;\ldots\; t_n \;:\; T_n}{<t_1,\ldots,t_n> :\; T_1 * \cdots * T_n}$$

Projections:

$$\frac{t:\mathsf{T}^n \;\; u:\mathsf{Int}_n}{t\,!\,u:\mathsf{T}} \quad \frac{t:\mathsf{T}_1 * \ldots * \mathsf{T}_n}{t\,!\,i:\mathsf{T}_i}\, i=1,\ldots,n$$

Argument variables:

$$\frac{}{T_1,\ldots,T_n \vdash \$_i:\mathsf{T}_i}\, i=1,\ldots,n$$

Table 2.1: The type system of PGF concrete syntax.

Table 2.1 gives the static typing rules of the terms in PGF. It defines the relation $\Gamma \vdash t \;:\; T$ (“in context Γ, term t has type T”). The context is a sequence of types, and it is left implicit in all rules except the one for argument variables.

The context is in each linearization function created from the linearization types of the arguments of the function. Thus, if we have

fun $f:(\mathsf{C}_1,\ldots,\mathsf{C}_n)\to\mathsf{C}$
lincat $\mathsf{C}_1=\mathsf{T}_1;\ldots;\mathsf{C}_n=\mathsf{T}_n$
lincat $\mathsf{C}=\mathsf{T}$
lin $f=t$

then we must also have

$$T_1,\ldots,T_n \vdash t \;:\; T$$

The alert reader may notice that the typing rules for projections are partial: they cover only the special cases where either all types in the tuple are the same (denoted T^n) or the index is an integer constant. If the types are different and the index has an unknown value (which happens when it e.g. depends on an argument variable), the type of the projection cannot be known at compile time. As we will

see in Section 2.5.4, PGF grammars compiled from GF always fall under these special cases.

2.2.4 Examples of a concrete syntax

Let us build a concrete syntax for the abstract syntax of the Section 2.2. We define the linearization type of sentences to be just strings, whereas noun phrases and verb phrases are more complex, to account for **agreement**. Thus a noun phrase is a pair of a string and an agreement feature, where the feature is represented by an integer. A verb phrase is a tuple of strings—as many as there are possible agreement features. In this simple example, we just assume two features, corresponding to the singular and the plural.

$$\textbf{lincat } \mathsf{S} = \mathsf{Str}; \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2; \mathsf{VP} = \mathsf{Str} * \mathsf{Str};$$

The linearization of John is the string "John" with the feature 1, representing the singular. The linearization of Walk gives the two forms of the verb *walk*.

$$\textbf{lin } \mathsf{John} = <\text{"John"}, 1>; \mathsf{Walk} = <\text{"walks"}, \text{"walk"}>$$

The agreement itself is expressed as follows: in predication, the first field (1) of the noun phrase ($\$_1$)—is concatenated with a field of the verb phrase ($\$_2$). The field that is selected is given by the second field of the first argument ($\$_1 \,!\, 2$):

$$\textbf{lin } \mathsf{Pred} = \$_1 \,!\, 1 +\!\!+ \$_2 \,!\, (\$_1 \,!\, 2)$$

The power of a multilingual grammar comes from the fact that both linearization types and linearization functions are defined independently in each concrete syntax. Thus German, in which both a two-value number and a three-value person are needed in predication can be given the following concrete syntax:

$$\begin{array}{l}
\textbf{lincat } \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3 \\
\qquad\quad \mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) \\
\textbf{lin } \mathsf{Pred} = \$_1 \,!\, 1 +\!\!+ (\$_2 \,!\, (\$_1 \,!\, 2)) \,!\, (\$_1 \,!\, 3) \\
\quad \mathsf{John} = <\text{"John"}, 1, 3> \\
\quad \mathsf{Walk} = <<\text{"gehe"}, \text{"gehst"}, \text{"geht"}>, <\text{"gehen"}, \text{"geht"}, \text{"gehen"}>>
\end{array}$$

Strings:

$$[\,] \Downarrow [\,] \quad \text{“foo”} \Downarrow \text{“foo”} \quad \frac{s \Downarrow v \;\; t \Downarrow w}{s +\!\!+ t \Downarrow v +\!\!+ w}$$

Bounded integers:

$$i \Downarrow i$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \;\ldots\; t_n \Downarrow v_n}{<t_1,\ldots,t_n> \Downarrow <v_1,\ldots,v_n>}$$

Projections:

$$\frac{t \Downarrow <v_1,\ldots,v_n> \;\; u \Downarrow i}{t!u \Downarrow v_i}\; i = 1,\ldots,n$$

Argument variable:

$$v_1,\ldots,v_n \vdash \$_i \Downarrow v_i \; (i = 1,\ldots,n)$$

Table 2.2: The operational semantics of PGF.

2.2.5 Linearization

Linearization—the operation $\mapsto$ converting trees into concrete syntax objects—is performed by following the operational semantics given in Table 2.2. The table defines the relation $\gamma \vdash t \Downarrow v$ (“in context γ, term t evaluates to term v”). The context is a sequence of terms, and it is left implicit in all rules except the one for argument variables.

To linearize a tree, we linearize its immediate subtrees, and the sequence of the resulting terms constitutes the context of evaluation for the full tree:

$$\frac{a_1 \mapsto t_1 \;\ldots\; a_n \mapsto t_n \;\; t_1,\ldots,t_n \vdash t \Downarrow v}{f\,a_1 \ldots a_n \mapsto v}\mathbf{lin}\, f = t$$

Since linearization operates on the linearizations of immediate subtrees, it is a homomorphism, in other words, a compositional mapping. A crucial property of PGF making it possible to maintain the compositionality of linearization in realistic grammars is that its value need not be a string, but a richer data structure. If strings are what ultimately are needed, one can require that the start category has the linearization type `Str`; alternatively, one can define a realization operation that finds the first string in a tuple recursively.

2.3 Properties of PGF

2.3.1 Expressive power

Context-free grammars have a straightforward representation in PGF: consider a rule

$$C \longrightarrow t_1 \ldots t_n$$

where every t_i is either a nonterminal C_j or a terminal s. This rule can be translated to a pair of an abstract function and a linearization:

$$\begin{array}{l}\textbf{fun}\, f : (\mathsf{C}_1, \ldots, \mathsf{C}_m) \to \mathsf{C} \\ \textbf{lin}\, f = u_1 ++ \ldots ++ u_n\end{array}$$

where f is a unique label, $(\mathsf{C}_1, \ldots, \mathsf{C}_m)$ are the nonterminals in the rule, and each u_i is either $\$_j$ (if $t_i = \mathsf{C}_j$) or s (if $t_i = s$).

That PGF is stronger than context-free grammars, is easily shown by the language $a^n b^n c^n$, whose grammar shows how **discontinuous constituents** are encoded as tuples of strings. The grammar is the following:

$$\begin{array}{l}\textbf{cat}\ \mathsf{S}; \mathsf{A} \\ \textbf{fun}\ s : (\mathsf{A}) \to \mathsf{S}; e : () \to \mathsf{A}; a : (\mathsf{A}) \to \mathsf{A} \\ \textbf{lincat}\ \mathsf{S} = \mathsf{Str}; \mathsf{A} = \mathsf{Str} * \mathsf{Str} * \mathsf{Str} \\ \textbf{lin}\ \ s = \$_1 ! 1 ++ \$_1 ! 2 ++ \$_1 ! 3 \\ \qquad e = <[\,], [\,], [\,]> \\ \qquad a = <\text{“a”} ++ \$_1 ! 1, \text{“b”} ++ \$_1 ! 2, \text{“c”} ++ \$_1 ! 3>\end{array}$$

The general result about PGF is that it is equivalent to PMCFG (Parallel Multiple Context-Free Grammar, Seki et al. 1991). Hence any PGF grammar is parsable in polynomial time, with the exponent dependent on the grammar. This result was obtained for a more complex subset of GF by Ljunglöf (2004). Section 2.4 on parser generation will outline the result for PGF; it will also present a definition of PMCFG.

Being polynomial, PGF is not fully context-sensitive, but it is not mildly context-sensitive in the sense of (Joshi et al. 1991), because it does not have the **constant-growth property**. A counterexample is the following grammar, which defines the exponential language $\{a^{2^n} \mid n = 0, 1, 2, \ldots\}$:

$$\begin{array}{l}\textbf{cat}\ \mathsf{S} \\ \textbf{fun}\ a : () \to \mathsf{S}; s : (\mathsf{S}) \to \mathsf{S}\end{array}$$

lincat $\mathsf{S} = \mathsf{Str}$
lin $a =$ "a"; $s = \$_1 ++ \$_1$

2.3.2 Extensions of concrete syntax

A useful extension of concrete syntax is **free variation**, expressed by the operator $|$. Free variation is used in concrete syntax to indicate that different expressions make no semantic difference, that is, that they have the same abstract syntax. A term $t \mid u$ is well-formed in any type T, if both t and u have type T. In operational semantics, free variation requires the lifting of other operations to cover operands of the form $t|u$. For instance,

$$(t \mid u) \,!\, v = (t \,!\, v) \mid (u \,!\, v)$$

As shown by Ljunglöf (2004), the semantics can be given in such a way that the complexity of parsing PGF is not affected.

In practical applications of PGF, it is useful to have non-canonical forms that decrease the size of grammars by factoring out common parts. In particular, the use of **macros** factors out terms occurring in several linearization rules. The method of **common subexpression elimination** often results in a grammar whose code size is just one tenth of the original. This method is standardly applied as a back-end optimization in the GF grammar compiler (Section 2.5.4), which may otherwise result in code explosion.

2.3.3 Extensions of abstract syntax

One of the original motivations of GF was to implement type-theoretical semantics as presented in Ranta (1994). This kind of semantics requires that the abstract syntax notation has the strength of a **logical framework**, in the sense of Martin-Löf (1984) and Harper et al. (1993). What we have presented above is a special case of this, with three ingredients missing:

- **Dependent types**: a category may take trees as arguments.
- **Higher-order abstract syntax**: a function may take functions as arguments.
- **Semantic definitions**: trees may be given computation rules.

Ranta (2004a) gives the typing rules and operational semantics for these extensions of GF. The extensions have been transferred to PGF as well, but only the Haskell implementation currently supports them. The reason for this is mainly that these extensions are rarely used in GF applications (Burke and Johannisson 2005 and Jonson 2006 being exceptions). Language-theoretically, their effect is either dramatic or null, depending on how the language defined by a grammar is conceived. If parsing is expected to return only well-typed trees, then dependent types together with semantic definitions makes parsing undecidable, because type checking is undecidable. If parsing and dependent type checking are separated, as they are in practical implementations, dependent types make no difference to the parsing complexity.

Metavariables are an extension useful for several purposes. In particular, they are used for encoding suppressed arguments when parsing grammars with erasing linearization rules (Section 2.4.5). In dependent type checking, they are used for unifying type dependencies, and in interactive syntax editors (Khegai et al. 2003), they are used for marking unfinished parts of syntax trees.

2.4 Parsing

PGF concrete syntax is simple but still too complex to be used directly for efficient parsing and for that reason it is converted to PMCFG. It is possible to do the conversion incrementally during the parsing but this slows down the process, so instead we do the conversion in the compiler. This means that we have duplicated information in the PGF file, because the two formalisms are equivalent, but this is a trade off between efficiency and grammar size. At the same time it is not feasible to use only the PMCFG because it might be quite big in some cases while the client might be interested only in linearization for which he or she can use only the PGF syntax.

2.4.1 PMCFG definition

Definition 1 *A parallel multiple context-free grammar is a 8-tuple* $G = (N, T, F, P, S, d, r, a)$ *where:*

- N *is a finite set of categories and a positive integer* $d(A)$ *called dimension is given for each* $A \in N$.
- T *is a finite set of terminal symbols which is disjoint with* N.

- *F is a finite set of functions where the arity $a(f)$ and the dimensions $r(f)$ and $d_i(f)$ $(1 \leq i \leq a(f))$ are given for every $f \in F$. Let's for every positive integer d, $(T^*)^d$ denote the set of all d-tuples of strings over T. The function is a total mapping from $(T^*)^{d_1(f)} \times (T^*)^{d_2(f)} \times \cdots \times (T^*)^{d_{a(f)}(f)}$ to $(T^*)^{r(f)}$ and it is defined as:*

  $f := < \alpha_1, \alpha_2, \ldots, \alpha_{r(f)} >$

  Each α_i is a string of terminals and $\langle k;l \rangle$ pairs, where $1 \leq k \leq a(f)$ is called argument index and $1 \leq l \leq d_k(f)$ is called constituent index.

- *P is a finite set of productions of the form:*

  $A \rightarrow f[A_1, A_2, \ldots, A_{a(f)}]$

  where $A \in N$ is called result category, $A_1, A_2, \ldots, A_{a(f)} \in N$ are called argument categories and $f \in F$ is the function symbol. For the production to be well formed the conditions $d_i(f) = d(A_i)$ $(1 \leq i \leq a(f))$ and $r(f) = d(A)$ must hold.

- *S is the start category and $d(S) = 1$.*

We use the same definition of PMCFG as is used by Seki and Kato (2008) and Seki et al. (1993) with the minor difference that they use variable names like x_{kl} while we use $\langle k;l \rangle$ to refer to the function arguments. In the actual implementation we also allow every category to be used as a start category. When the category has multiple constituents then they all are tried from the parser.

The $a^n b^n c^n$ language from section 2.3.1 is represented in PMCFG as shown in Figure 2.1. The dimension of S and A are $d(S) = 1$ and $d(A) = 3$. Functions s and e are with 0-arity and function a is with 1-arity i.e. $a(s) = 0$, $a(a) = 1$ and $a(e) = 0$.

2.4.2 PMCFG generation

Both PGF and PMCFG deal with tuples but PGF is allowed to have bounded integers and nested tuples while PMCFG is restricted to have only flat tuples containing only strings. To do the conversion we need to get rid of the integers and to flatten the tuples.

Let's take again the categories from section 2.2.4 as examples:

lincat $\mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3$
$\mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str})$

$$S \to s[A]$$
$$A \to a[A]$$
$$A \to e[]$$

$$s := < \langle 1;1 \rangle \ \langle 1;2 \rangle \ \langle 1;3 \rangle >$$
$$a := < a \ \langle 1;1 \rangle, b \ \langle 1;2 \rangle, c \ \langle 1;3 \rangle >$$
$$e := < [], [], [] >$$

Figure 2.1: PMCFG for the $a^n b^n c^n$language.

The type for VP does not contain any integers so it is simply flattened to one tuple with six constituents. When the linearization type contains integers then the category is split into multiple PMCFG categories, one for each combination of integer values. The integers are removed and the remaining type is flattened to one tuple. In this case for NP we will have six categories and they will all have a single string as a linearization type. Category splitting is not an uncommon operation. For example, in part-of-speech tagging there are usually different tags for nouns in plural and nouns in singular. We do this also on a syntactic level and the NP category is split into:

NP_{11} NP_{12} NP_{13}
NP_{21} NP_{22} NP_{23}

The conversion from PGF rules to PMCFG productions starts with abstract interpretation (Table 2.3), which is very similar to the operational semantics in Table 2.2. The major difference is that the argument values are known only at run-time and actually the $\$_i$ terms should be replaced by $\langle i;l \rangle$ pairs in PMCFG for some l. We extend the PGF syntax with a $\langle i;\pi \rangle$ meta-value which is only used internally in the generation. Here, π is a sequence of indices and not only one index as it is the case in the final PMCFG. Like in the operational semantics, the rules define the relation $\gamma \vdash t \Downarrow v$ but this time the context γ is a sequence of assumption sets. Each assumption set a_k contains pairs (π, i) which say that if π is the sequence of indices $l_1 \ldots l_n$ then we assume that $\$_k!l_1 \ldots !l_n \Downarrow i$.

In addition, two sets have to be computed for each category C: $T_s(C)$ and $T_p(C)$. T_s is the set of all sequences of integers $i_1 \ldots i_n$ such that if x is an expression of type C then $x!i_1!i_2!\ldots!i_n$ is of type *Str*. T_p is the set of all (π, k) pairs

Strings:

$$[] \Downarrow [] \quad "foo" \Downarrow "foo" \quad \frac{s \Downarrow v \;\; t \Downarrow w}{s{+}{+}t \Downarrow v{+}{+}w}$$

Bounded integers:

$$i \Downarrow i \; (i = 1,2,\ldots)$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \;\ldots\; t_n \Downarrow v_n}{< t_1,\ldots,t_n > \Downarrow < v_1,\ldots,v_n >}$$

Projections:

$$\frac{t \Downarrow < v_1,\ldots,v_n > \;\; u \Downarrow i}{t!u \Downarrow v_i}\, i = 1,\ldots,n \quad \frac{t \Downarrow \langle k;\pi\rangle \;\; u \Downarrow i}{t!u \Downarrow \langle k;\pi\, i\rangle}\, i = 1,\ldots,n$$

Argument variable:

$$\$_k \Downarrow \langle k;\, \rangle$$

Parameter Substitution:

$$\frac{a_1 \ldots a_n \vdash t \Downarrow \langle k;\pi\rangle}{a_1 \ldots a_k \cup (\pi,i) \ldots a_n, \vdash t \Downarrow i}\; (\pi,m) \in T_p(A_k),\, i = 1,\ldots,m,\, \forall j.((\pi,j) \in a_k \Rightarrow i = j)$$

Table 2.3: Abstract interpretation of PGF.

where π is again a sequence $i_1 \ldots i_n$ but this time $x!i_1!i_2!\ldots!i_n$ is of type Int_k.

The abstract interpretation rules for strings, integers and tuples are exactly the same but the rule for argument variables is completely new. It says that since we do not know the actual value of the variable we just replace it with the meta-value $\langle k;\, \rangle$. Furthermore there is an additional rule for projection which deals with the case when we have argument variable on the left-hand side of a projection. Basically the rule for argument variables and the extra projection rule converts terms like $\$_k!l_1 \ldots !l_n$ to meta-values like $\langle k;\pi\rangle$ where π is the sequence $l_1 \ldots l_n$. Since the terms are well-typed at some point either $\pi \in T_s(A_k)$ or $(\pi,n) \in T_p(A_k)$ for some n will be the case. In the first case the meta value will be left unchanged in the evaluated term. The second case is more important because it triggers the parameter substitution rule. This rule is nondeterministic because it makes an arbitrary choice for i and records the choice in the context γ. The last side condition in the rule ensures that if we already had made some asumption for $\langle k;\pi\rangle$ we cannot choose any other value except the value that is already in the context. Since

the integers are bounded we have only a finite set of choices which ensures the termination.

Let's use the linearization rule from section 2.2.4 as an example again:

lin Pred $= \$_1 \,!\, 1 +\!\!+ (\$_2 \,!\, (\$_1 \,!\, 2)) \,!\, (\$_1 \,!\, 3)$

The first subterm $\$_1 \,!\, 1$ is simply reduced to $\langle 1;1 \rangle$ by the derivation:

$$\frac{\$_1 \;\Downarrow\; \langle 1; \rangle \quad 1 \;\Downarrow\; 1}{\$_1!1 \;\Downarrow\; \langle 1;1 \rangle}$$

The derivation of the second subterm is more complex because it contains argument variables on the right hand side of a projection, so they have to be removed using the parameter substitution rule:

$$\frac{\dfrac{a_1\ a_2 \vdash \$_1 \Downarrow \langle 1; \rangle \quad a_1\ a_2 \vdash 2 \Downarrow 2}{a_1\ a_2 \vdash \$_1!2 \Downarrow \langle 1;2 \rangle}}{(a_1 \cup \{(2,x)\})\ a_2 \;\vdash\; \$_1!2 \;\Downarrow\; x}$$

Since the parameter substitution is nondeterministic there are two possible derivations but they differ only in the final value, so we use the variable x to denote either value 1 or 2. In a similar way we can deduce that $(a_1 \cup \{(2,x),(3,y)\})\ a_2 \vdash \$_1!3 \;\Downarrow\; y$, where y is either 1, 2 or 3. Combining the two results the derivation for the second term gives:

$$\frac{\dfrac{\$_2 \Downarrow \langle 2; \rangle \quad \$_1!2 \Downarrow x}{\$_2!(\$_1!2) \Downarrow \langle 2;x \rangle} \quad \$_1!3 \;\Downarrow\; y}{(\$_2!(\$_1!2))!(\$_1!3) \;\Downarrow\; \langle 2;x\ y \rangle}$$

By applying the rule for the concatenation the final result we get:

$$\langle 1;1 \rangle +\!\!+ \langle 2;x\ y \rangle$$

The output from the abstract interpretation can be converted directly to a PMCFG tuple. In well-typed terms a tuple can appear only at the top-level, inside another tuple or on the left-hand side of a record projection. In the abstract interpretation all tuples inside record projections are removed so that the only choice for the evaluated term is to be a tree of nested tuples with leaves of type either Str or Int_n. The tree strictly follows the structure of the linearization type of the result category. The term can be flattened just like we did with the linearization types.

For each possible tree a new unique function is generated with definition containing all leaves of type Str as tuple constituents. For the example above this will lead to six functions:

$$
\begin{aligned}
Pred_1 &:= \langle 1;1\rangle\ \langle 2;1\ 1\rangle \\
Pred_2 &:= \langle 1;1\rangle\ \langle 2;2\ 1\rangle \\
Pred_3 &:= \langle 1;1\rangle\ \langle 2;1\ 2\rangle \\
Pred_4 &:= \langle 1;1\rangle\ \langle 2;2\ 2\rangle \\
Pred_5 &:= \langle 1;1\rangle\ \langle 2;1\ 3\rangle \\
Pred_6 &:= \langle 1;1\rangle\ \langle 2;2\ 3\rangle
\end{aligned}
$$

The bounded integers in the term are used to determine the right PMCFG result category and each assumption set a_i in the context γ is used to determine the corresponding argument category in the production. One production is generated for every function:

$$
\begin{aligned}
S &\rightarrow Pred_1[\mathrm{NP}_{11}, \mathrm{VP}] \\
S &\rightarrow Pred_2[\mathrm{NP}_{21}, \mathrm{VP}] \\
S &\rightarrow Pred_3[\mathrm{NP}_{12}, \mathrm{VP}] \\
S &\rightarrow Pred_4[\mathrm{NP}_{22}, \mathrm{VP}] \\
S &\rightarrow Pred_5[\mathrm{NP}_{13}, \mathrm{VP}] \\
S &\rightarrow Pred_6[\mathrm{NP}_{23}, \mathrm{VP}]
\end{aligned}
$$

2.4.3 Optimizations in PMCFG

The produced PMCFG could be very big without some optimizations. Three techniques are implemented so far and they are described in this section. All of them have been discovered in experiments with real grammars.

The first observation is that the conversion algorithm in the previous section always generates a pair of function definition and production rule with the same function. It happens to be pretty common that one function could be reused in two different productions because the original definitions are equivalent. In the real implementation first the definition is generated and after that it is compared with the already existing definitions. Only if it is distinct a new function is generated.

Another issue is that there are many constituents in the function bodies which are equal but are used in different places. For that reason we collect a list of

distinct constituents and then the function definitions are rewritten to contain only the indices of the corresponding constituents. The following is an example of two functions which have one and the same constituent:

$$F_1 := (\langle 0;0\rangle, \langle 0;1\rangle\langle 1;0\rangle)$$
$$F_2 := (\text{"a"}, \langle 0;1\rangle\langle 1;0\rangle)$$

The optimizer will extract the constituents in a separate table and then the functions will contain only references to them:

$$F_1 := (S_1, S_2)$$
$$F_2 := (S_3, S_2)$$

$$S_1 := \langle 0;0\rangle$$
$$S_2 := \langle 0;1\rangle\langle 1;0\rangle$$
$$S_3 := \text{"a"}$$

Here the constituent S_2 is referenced by both function F_1 and F_2.

The last observation is that there are groups of productions like:

$$A \to f[B,C_1,D_1] \quad A \to f[B,C_2,D_1] \quad A \to f[B,C_3,D_1] \quad A \to f[B,C_4,D_1]$$
$$A \to f[B,C_1,D_2] \quad A \to f[B,C_2,D_2] \quad A \to f[B,C_3,D_2] \quad A \to f[B,C_4,D_2]$$

Note that the only difference between the productions is the selection of categories C_i and D_j and that all possible combinations are realized. This happens when in some linearization function some parameters are used only when another parameter has a specific value. The abstract interpetation could not detect all these cases. It detects only the parameters that are not used at all and then the conversion rules are introduced. The list of productions can be compressed by introducing extra conversion rules:

$$A \to f[B,C,D]$$
$$C \to \_[C_1]$$
$$C \to \_[C_2]$$
$$C \to \_[C_3]$$
$$C \to \_[C_4]$$
$$D \to \_[D_1]$$
$$D \to \_[D_2]$$

Language	Productions		Functions		Constituents		File Size (Kb)	
	Count	Ratio	Count	Ratio	Count	Ratio	Size	Ratio
Bulgarian	3521	1.03	1308	2.59	76460	1.84	3119	1.91
Catalan	6047	1.23	1489	4.29	27218	3.82	1375	5.34
Danish	1603	1.05	932	1.51	8746	2.09	360	1.55
English	1131	1.03	811	1.34	8607	5.30	409	2.05
Finnish	135213	-	1812	-	42028	-	3117	103.53
French	11520	1.14	1896	6.05	40790	3.63	3719	4.37
German	8208	1.37	3737	2.81	21337	2.47	1546	2.76
Interlingua	2774	1.00	1896	1.29	3843	2.09	245	1.39
Italian	19770	1.17	2019	9.71	39915	3.86	2193	7.76
Norwegian	1670	1.05	968	1.51	8642	2.19	361	1.57
Romanian	75173	1.50	1853	39.19	24222	5.00	1699	21.09
Russian	6278	1.04	1016	4.10	19035	3.17	985	2.18
Spanish	6006	1.23	1448	4.38	27499	3.75	1369	5.36
Swedish	1492	1.03	907	1.44	8837	2.01	352	1.48

Table 2.4: Grammar sizes in number of productions, functions, constituents and file size, for the GF Resource Grammar Library. The ratio is the koefficient by which the corresponding value grows if the optimizations were not applied.

These optimizations have been implemented and tried with the resource grammar library (Ranta 2008), which is the largest collection of grammars written in GF. The produced grammar sizes are summarized in Table 2.4. The columns show the grammar size in number of PMCFG productions, functions, constituents and in file size. Each column has two sub columns. The first sub column shows the count or the size in the optimized grammar and the second sub column shows the ratio by which the grammar grows if the optimizations were not applied. As it shows the optimizations could reduce the file size up to 103 times for Finnish. Without optimizations the compiled Finnish grammar grows to 315 Mb and cannot even be loaded back in the interpreter on computer with 2Gb of memory.

2.4.4 Parsing with PMCFG

Efficient recognition and parsing algorithms for MCFG have been described in (Nakanishi et al. 1997), Ljunglöf (2004) and (Burden and Ljunglöf 2005). MCFG is a linear form of PMCFG where each constituent of an argument is used exactly once. Ljunglöf (2004) gives an algorithm for approximating a PMCFG with an

MCFG. With the approximation it is possible to use a parsing algorithm for MCFG to parse with a PMCFG, but after that a postprocessing step is needed to recover the original parse tree and to filter out spurious parse trees via unification. Instead we are using a parsing algorithm that works directly with PMCFG and also has the advantage that it is incremental. The incrementality is important because it can be used for word prediction (see section 2.5.1). The incremental algorithm itself is described in section 3.

2.4.5 Parse trees

The output from the parser is a syntax tree or a set of trees where the nodes are labeled with PMCFG functions. The trees have to be converted back to PGF abstract expression. In the absense of high-order terms (section 2.3.3) the transformation is trivial. The definition of each PMCFG function is annotated with the corresponding PGF linearization function so they just have to be replaced. The PMCFG grammar might be erasing i.e. some argument might not be used at all. In this case the slot for this argument is filled with meta variable.

2.5 Using PGF

In this section, we will outline how PGF grammars can be used to construct natural language processing applications. We first list a number of operations that can be performed with a PGF grammar, along with some possible applications of these operations. We then give a brief overview of the APIs (Application Programmer's Interfaces) which are currently available for using PGF functionality in applications. Finally, we outline how PGF grammars can be produced from the more grammar-writer friendly format GF.

2.5.1 PGF operations

PGF grammars can be used for a wide range of tasks, either stand-alone, or as integral parts of a larger application. Figure 2.2 shows an overview of how PGF grammars can be used in natural language processing applications.

Parsing takes a string in some language and produces zero (in the case of an out-of-grammar input), one (in the case of an unambiguous input), or more (for ambiguous inputs) abstract syntax trees. PGF parsing is for example useful for handling natural language user input in applications. This lets the rest of the

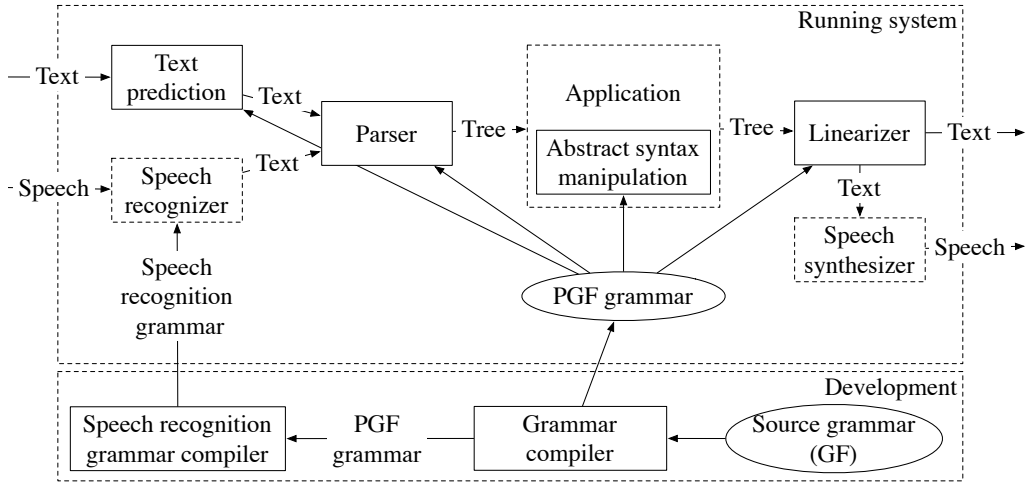


Figure 2.2: Overview of PGF applications.

application work on abstract syntax trees, which makes it easy to localize the application to new languages.

Text Prediction is related to parsing. The incremental PMCFG parsing algorithm can parse an incomplete input, and return the set of possible next tokens. If the last token in the given input is itself incomplete, the list of complete tokens can be given to the parser, and the result filtered to retain only those tokens that have the last (incomplete) token as a prefix.

Linearization, the production of text given an abstract syntax tree, can be used to produce natural language system outputs, either as text, or, by using speech synthesis software, speech output. In the latter case, the concrete syntax may contain annotations such as SSML tags which help the speech synthesizer.

Translation is the combination of parsing and linearization (Khegai 2006a).

Abstract syntax generation, is made easy by the PGF abstract syntax type system. A potentially infinite lazy list of abstract syntax trees can be generated randomly, or exhaustively through iterative deepening. Random generation can be used to generate a monolingual corpus (generate a list of random abstract syntax trees and linearize them to given language), a multilingual parallel corpus (generate trees and linearize to several languages), or a treebank (generate trees, linearize, and output text, tree pairs).

Abstract syntax manipulation is useful in any application that analyzes or produces abstract syntax trees. The PGF type system makes it possible to expose abstract syntax manipulation to the user in a safe way. When statically typed lan-

```
data S  = Pred NP VP
data NP = John
data VP = Walk
```

```
abstract class S {...}
class Pred extends S {NP np; VP vp; ...}
abstract class NP {...}
class John extends NP {...}
abstract class VP {...}
class Walk extends VP {...}
```

Figure 2.3: Haskell and Java data types for the abstract syntax in Section 2.2.2.

```
readPGF :: FilePath → IO PGF
linearize :: PGF → Language → Tree → String
parse :: PGF → Language → Category → String → [Tree]
generateRandom :: PGF → Category → IO [Tree]
generateAll :: PGF → Category → [Tree]
complete :: PGF → Language → Category → String → [String]
```

Figure 2.4: A part of the Haskell PGF Interpreter API.

guages such as Haskell or Java are used, it is possible to generate host language data types from an Figure 2.3 shows such data types generated from the example abstract syntax in Section 2.2.2.

2.5.2 PGF Interpreter API

A PGF Interpreter API allows an application programmer to make use of PGF grammars for the tasks listed above in a program written in some general purpose programming language. There are currently APIs for Haskell, Java, JavaScript, Prolog, C, C++, and web applications, with varying degrees of functionality. We examine the Haskell API in some detail. The other APIs, some of which are covered briefly below, have the same functionality, or subsets of it.

Haskell API Figure 2.4 shows the most important functions in the Haskell PGF Interpreter API. There are also functions for manipulating Tree value, for get-

```
public class PGF {
  public static PGF readPGF (String path)
  public String linearize (String lang, Tree tree)
  public List<Tree> parse (String lang, String text)
}
```

Figure 2.5: A part of the Java PGF Interpreter API.

ting metadata about grammars, and variants of generation functions which give more control over the generation process, for example by setting a maximum tree height.

Java PGF Interpreter API The Java API is very similar to the Haskell API, but with a more object-oriented design, see Figure 2.5.

JavaScript PGF Interpreter API PGF grammars can also be converted into a JavaScript format, for which there is an interpreter that implements linearization, parsing and abstract syntax tree manipulation (Meza Moreno 2008).

PGF Interpreter Web Service PGF parsing, linearization and translation is also available as a web service, in the form of a FastCGI (Brown 1996) program with a RESTful (Fielding 2000) interface that returns JSON (Crockford 2006) structures. For example, a request to translate the sentence *this fish is fresh* from the concrete syntax `FoodEng` to all available concrete syntaxes may return the following JSON structure:

```
[{"from":"FoodEng","to":"FoodEng","text":"this fish is fresh"},
 {"from":"FoodEng","to":"FoodIta","text":"questo pesce è fresco"}]
```

2.5.3 Compiling PGF to other formats

The declarative nature of PGF makes it possible to translate PGF grammars to other grammar formalisms. It is theoretically interesting to produce algorithms for translating between different grammar formalisms. However, it also has practical applications, as it lets us use PGF grammars with existing software systems

based on other grammar formalisms, for example speech recognizers (Bringert 2007a). When combined with PGF parsing, this makes it possible for an application to accept speech input, based solely on the information in the PGF grammar. Examples of such applications include dialogue systems (Ericsson et al. 2006; Bringert 2007b) and speech translation (Bringert 2008).

Most examples in the rest of this section will be based on the PGF grammar shown in Figure 2.7, which extends the earlier example grammar with the And and We functions. This grammar has been chosen to compactly include both agreement and left-recursion (at the expense of ambiguous parsing, this could be fixed with a slightly larger grammar).

Context-free approximation

A context-free grammar (CFG) that approximates a PGF grammar can be produced by first producing a PMCFG as described earlier. The PMCFG is then approximated with a CFG by converting each PMCFG category-field pair to a CFG category. In the general case, this is an approximation, since PMCFG is a stronger formalism than CFG. The example language with agreement is converted to the CFG shown in Figure 2.8.

The PMCFG grammar given in Section 2.4.1 for the context-sensitive language $a^n b^n c^n$ is approximated by the context-free grammar shown in Figure 2.14. In this case the context-free approximation is overgenerating, as it generates the language $a^* b^* c^*$. The simpler $a^n b^n$ language is context-free, but may be described by either a context-sensitive grammar in a similar way to the $a^n b^n c^n$ language above, or by a context-free grammar. The context-free approximation will preserve the language described by an already context-free grammar, but not necessarily the language of a context-sensitive grammar that defines a context-free language. It is not possible to devise an algorithm that converts all context-sensitive grammars that generate context-free languages to context-free grammars, since deciding whether a context-sensitive language is context-free is undecidable. We conjecture that this also holds for deciding whether a PMCFG generates a context-free language.

Context-free transformations

Our PGF compiler also implements a number of transformations on context-free grammars, such as cycle elimination, bottom-up and top-down filtering, left-recursion elimination, identical category elimination, and EBNF compaction. This makes

```
cat S; NP; VP
fun And      : S → S → S
    Pred     : NP → VP → S
    John, We : NP
    Walk     : VP
param Num  = Sg | Pl
param Pers = P1 | P2 | P3
lincat S   = Str
       NP  = {s : Str; n : Num; p : Pers}
       VP  = Num ⇒ Pers ⇒ Str
lin And x y = x ++ "und" ++ y
    Pred np vp = np.s ++ vp ! np.n ! np.p
    John = {s = "John"; n = Sg; p = P3}
    We   = {s = "wir"; n = Pl; p = P1}
    Walk =
      table { Sg ⇒ table {P1 ⇒ "gehe";
                          P2 ⇒ "gehst";
                          P3 ⇒ "geht"};
              Pl ⇒ table {P1 ⇒ "gehen";
                          P2 ⇒ "geht";
                          P3 ⇒ "gehen"} }
```

Figure 2.6: GF grammar.

```
cat S; NP; VP
fun And  : (S, S) → S
    Pred : (NP, VP) → S
    John : () → NP
    We   : () → NP
    Walk : () → VP
lincat S  = Str
       NP = Str * Int2 * Int3
       VP = (Str * Str * Str)
            * (Str * Str * Str)
lin And  = $1 ++ "und" ++ $2
    Pred = $1 ! 1 ++ ($2 ! ($1 ! 2)) ! ($1 ! 3)
    John = <"John", 1, 3>
    We   = <"wir", 2, 1>
    Walk =
      < <"gehe",
         "gehst",
         "geht">,
        <"gehen",
         "geht",
         "gehen"> >
```

Figure 2.7: PGF grammar.

```
S    → S "und" S | NP1 VP2 | NP2 VP1
NP1 → "John"
NP2 → "wir"
VP1 → "gehen"
VP2 → "geht"
```

Figure 2.8: CFG.

```
S    →  S "und" S{And ($1, $2)}
      | NP1 VP2{Pred ($1, $2)}
      | NP2 VP1{Pred ($1, $2)}
NP1 → "John"{John ()}
NP2 → "wir"{We ()}
VP1 → "gehen"{Walk ()}
VP2 → "geht"{Walk ()}
```

Figure 2.9: Fig. 2.8 with interpretation.

$$
\begin{array}{lll}
\mathsf{S} & \rightarrow & \mathsf{NP}_1\ \mathsf{S}_2 \mid \mathsf{NP}_2\ \mathsf{S}_3 \\
\mathsf{S}_2 & \rightarrow & \mathsf{VP}_2 \mid \mathsf{VP}_2\ \mathsf{S}_4 \\
\mathsf{S}_3 & \rightarrow & \mathsf{VP}_1 \mid \mathsf{VP}_1\ \mathsf{S}_4 \\
\mathsf{S}_4 & \rightarrow & \text{“und”}\ \mathsf{S} \mid \text{“und”}\ \mathsf{S}\ \mathsf{S}_4 \\
\mathsf{NP}_1 & \rightarrow & \text{“John”} \\
\mathsf{NP}_2 & \rightarrow & \text{“wir”} \\
\mathsf{VP}_1 & \rightarrow & \text{“gehen”} \\
\mathsf{VP}_2 & \rightarrow & \text{“geht”}
\end{array}
$$

Figure 2.10: Non-left-recursive CFG.

$$
\begin{array}{lll}
\mathsf{S} & \rightarrow & \mathsf{NP}_1\ \mathsf{S}_2\{\$_2\ (\$_1)\} \\
 & & \mid \mathsf{NP}_2\ \mathsf{S}_3\{\$_2\ (\$_1)\} \\
\mathsf{S}_2 & \rightarrow & \mathsf{VP}_2\{\lambda x.\mathsf{Pred}\ (x,\$_1)\} \\
 & & \mid \mathsf{VP}_2\ \mathsf{S}_4\{\lambda x.\$_2\ (\mathsf{Pred}\ (x,\$_1))\} \\
\mathsf{S}_3 & \rightarrow & \mathsf{VP}_1\{\lambda x.\mathsf{Pred}\ (x,\$_1)\} \\
 & & \mid \mathsf{VP}_1\ \mathsf{S}_4\{\lambda x.\$_2\ (\mathsf{Pred}\ (x,\$_1))\} \\
\mathsf{S}_4 & \rightarrow & \text{“und”}\ \mathsf{S}\{\lambda x.\mathsf{And}\ (x,\$_1)\} \\
 & & \mid \text{“und”}\ \mathsf{S}\ \mathsf{S}_4\{\lambda x.\$_2\ (\mathsf{And}\ (x,\$_1))\} \\
\mathsf{NP}_1 & \rightarrow & \text{“John”}\{\mathsf{John}\ ()\} \\
\mathsf{NP}_2 & \rightarrow & \text{“wir”}\{\mathsf{We}\ ()\} \\
\mathsf{VP}_1 & \rightarrow & \text{“gehen”}\{\mathsf{Walk}\ ()\} \\
\mathsf{VP}_2 & \rightarrow & \text{“geht”}\{\mathsf{Walk}\ ()\}
\end{array}
$$

Figure 2.11: Fig. 2.10 with interpretation.

$$
\begin{array}{lll}
\mathsf{S} & \rightarrow & \mathsf{NP}_1\ \mathsf{VP}_2\ \mathsf{S}_2 \mid \mathsf{NP}_2\ \mathsf{VP}_1\ \mathsf{S}_2 \\
\mathsf{S}_2 & \rightarrow & \text{“und”}\ \mathsf{S} \mid \varepsilon \\
\mathsf{NP}_1 & \rightarrow & \text{“John”} \\
\mathsf{NP}_2 & \rightarrow & \text{“wir”} \\
\mathsf{VP}_1 & \rightarrow & \text{“gehen”} \\
\mathsf{VP}_2 & \rightarrow & \text{“geht”}
\end{array}
$$

Figure 2.12: Regular grammar.

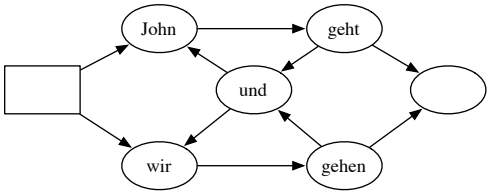


Figure 2.13: Finite-state automaton.

$$
\begin{array}{lll}
\mathsf{S} & \rightarrow & \mathsf{A}_0\ \mathsf{A}_1\ \mathsf{A}_2 \\
\mathsf{A} & \rightarrow & \mathsf{A}_0 \mid \mathsf{A}_1 \mid \mathsf{A}_2 \\
\mathsf{A}_0 & \rightarrow & \varepsilon \mid \text{“a”}\ \mathsf{A}_0 \\
\mathsf{A}_1 & \rightarrow & \varepsilon \mid \text{“b”}\ \mathsf{A}_1 \\
\mathsf{A}_2 & \rightarrow & \varepsilon \mid \text{“c”}\ \mathsf{A}_2
\end{array}
$$

Figure 2.14: Context-free approximation of the PMCFG grammar for the $a^n b^n c^n$ language.

it possible to produce grammars in a number of restricted context-free formats as required by speech recognition software.

Embedded tree building code

The rules for producing abstract syntax trees can be preserved through the grammar translations listed above. This makes it possible to include abstract syntax tree building code in the generated context-free grammars, for example in SISR (Burke and Van Tichelen 2006) format. Figure 2.9 shows such an annotated grammar. As is shown in Figure 2.11, the abstract syntax tree building annotations can be carried through the left-recursion removal transformation, by the use of λ-terms.

Regular / finite-state approximation

The PGF grammar compiler can also approximate the produced context-free grammars with regular grammars, which can in turn be compiled to finite-state automata. An example of this is shown in Figures 2.12 and 2.13. This lets us support speech recognizers which require regular or finite-state language models, or for typical finite-state natural language processing tasks such as marking noun phrases in a corpus.

2.5.4 Compiling GF to PGF

While PGF is suitable for efficient and simple implementation, PGF grammars are not meant to be produced by humans. Rather, it is intended as the target language of a compiler from a high-level grammar language. In this section, we will outline the differences between the human-friendly GF language, and the machine-friendly PGF language, and how grammars written in GF can be translated to PGF. The PGF syntax and semantics can be seen as a subset of the GF syntax and semantics, while the PGF type system is less strict than that of GF. Where the GF type system is concerned with correctness, the PGF type system only ensures safety.

Tables and records

In GF, there are two separate constructs that correspond to PGF tuples: tables and records. In GF, tables are tuples whose values all have the same type, and finite

algebraic data types or records are used to select values from tables. Tables are used to implemented parametric features such as inflection.

Records, on the other hand, can have values of different type, but the selectors used with records are labels which must be known statically. Records are used to implement inherent features and discontinuous constituents.

Both tables and records can be nested, but they always have a statically known size, which only depends on their type. This makes it possible to translate both records and tables to PGF tuples. For example, the GF grammar shown in Figure 2.6 is translated to the PGF grammar in Figure 2.7.

Structured parameter values and labels

As noted above, in GF, table projection is done with structured values known as parameters. These can be combinations of non-recursive algebraic data types and records of parameters. Since parameter records contain a known number of values, and the algebraic parameter values are non-recursive, each parameter type contains a finite number of values. This makes it possible to translate each parameter type to a bounded integer type, suitable for projection on PGF tuples.

Stricter type checking

As noted above, the GF concrete syntax type system is stricter than the PGF type system. In PGF, bounded integers are used to represent all parameter types and record labels, which means that many distinctions made by the GF type checker are not available in PGF. The differences between the GF and PGF type system can be compared to the differences between the Java type system and JVM bytecode verification. The type system for abstract syntax is identical in GF and PGF.

Pattern matching

Table projection in GF can be done by pattern matching with rather complex patterns. When compiling to PGF, all tables are expanded to have one branch with for each possible parameter value, to allow for translation to PGF tuples.

Modularity

GF grammars are organized into modules which can be compiled to core GF separately, like Java classes or C object files. PGF on the other hand has no module

system and is a single file, similar to a statically linked executable. This simplifies PGF implementations and makes it easy to distribute PGF grammars.

Auxiliary operations

In GF, auxiliary operations can be defined in order to implement for example morphological paradigms or common syntactic operations. These operations, like all GF concrete syntax can include complex features such as higher-order functions and pattern matching on strings. During the translation to PGF, all auxiliary operations are inlined, and all expressions are evaluated to produce valid PGF terms. This means, for example, that all strings which are used in pattern matching or in-token concatenation must be statically computable.

Because this inlining and computation can produce very large and repetitive PGF terms, common sub-expression elimination is performed. This can recover some of the auxiliary operations, but in a simpler form, but it can also discover new opportunities for code sharing, as PGF macros allow open terms, are type agnostic, and shared between code that comes from unrelated GF modules.

2.6 Results and evaluation

2.6.1 Systems using PGF

A number of natural language applications have been implemented using PGF or its predecessor GFCM. Above, we have listed a number of individual tasks that can be performed with PGF grammars. However, realistic applications often make use PGF for more than one task. This helps avoid the duplicated work involved in manually implementing multiple components which cover the same language fragment.

GOTTIS (Bringert et al. 2005) is an experimental multimodal and multilingual dialogue system for public transportation queries. It uses the a generated Nuance GSL speech recognition grammar for speech input, embedded parsing and linearization for system input and output, and generated Java data types for analysing input abstract syntax trees and producing output abstract syntax trees. Both input and output in GOTTIS make use of *multimodal* grammars. The input grammar allows *integrated multimodality*, e.g. “I want to go here”, accompanied by a click on a map. This is implemented by using a two-field record (PGF tuple) to represent spatial expressions, where one field contains the spoken component,

and another contains the click component. System output makes use of *parallel multimodality*; one concrete syntax produces spoken route descriptions, and another produces drawing instructions which are used to display routes on a map. Other examples of using GF grammars for formal languages include the WebALT mathematics problem editor (Cohen et al. 2006) and the KeY software specification editor (Burke and Johannisson 2005),

PGF abstract syntax can be used as a specification of a dialogue manager (Ranta and Cooper 2004). Together with the existing speech recognition grammar generation with embedded semantic interpretation tags, and the PGF to JavaScript compiler, this can be used to generate complete multimodal and multilingual VoiceXML dialogue systems (Bringert 2007b).

DUDE (Lemon and Liu 2006) and its extension REALL-DUDE (Lemon et al. 2006) are environments where non-experts can develop dialogue systems based on Business Process Models describing the applications. From keywords, prompts and answer sets defined by the developer, the system generates a GF grammar. This grammar is used for parsing input, and for generating a language model in SLF or GSL format.

Several syntax-directed editors for controlled languages (Khegai et al. 2003; Johannisson et al. 2003; Meza Moreno and Bringert 2008), have been implemented using PGF and its predecessors. They make use of abstract syntax manipulation, parsing linearization.

PGF can be used to implement complete limited domain speech translation systems that use PGF to produce speech recognition grammars and to perform parsing and linearization (Bringert 2008).

Text prediction can be used to implement text-based user interfaces which can show the user what inputs are allowed by the grammar. Examples of applications where this might be useful are editors for controlled languages, language learning software or limited domain translation software such as tourist phrase books. A web-based controlled language translator using text prediction is currently being developed.

Jonson (2006, 2007) used random corpus generation to produce statistical language models (SLM) for speech recognition from a domain-specific grammar. When combined with a general SLM trained on existing general domain text, the precision on in-grammar inputs was largely unchanged, while out-of-grammar inputs where handled with much higher precision, compared to a pure grammar-based language model. When producing corpora for training SLMs, dependently typed abstract syntax can be used to reduce over-generation (Jonson 2006). Other, as yet unexplored, applications of PGF abstract syntax generation are the use of

generated multilingual parallel corpora for training statistical machine translation systems, and the use of generated treebanks for training statistical parsers.

The grammars from the GF resource grammar library (Ranta 2008) can be used not only to implement application-specific grammars, but also as general wide-coverage grammars. For example, the English resource grammar, with a large lexicon and some minor syntax extensions, covers a significant portion of the sentences in the FraCaS semantic test suite (Cooper et al. 1996).

2.7 Related work

Compilation of grammars to low-level formats is an old idea. The most well-known examples are parser generators in the YACC family (Johnson 1975). The output of YACC-like parser generation is a piece of host language source code, which can be seamlessly integrated in a program written in the host language. YACC-like systems for full context-free grammars suitable for natural language include the work by Tomita and Carbonell (1987), the NLYACC system (Ishii et al. 1994), and the GLR extension of the Happy parser tool for Haskell (Callaghan and Medlock 2004). The main differences between PGF and the YACC family are that PGF is stronger than context-free grammars, that PGF contains no host language code and is therefore portable to many host languages, that PGF supports linearization in addition to parsing, and that PGF grammars can be multilingual.

HPSG (Pollard and Sag 1994) and LFG (Bresnan 1982) are grammar formalisms used for large-scale grammar writing and processing. In their implementation, the use of optimizing compilers is essential, to support at the same time high-level grammar writing and efficient run-time execution. HPSG compilers, for instance, have used advanced compiler techniques such as data flow analysis (Minnen et al. 1995). The main focus in both grammars is parsing, but also generation is supported. Both in HPSG and LFG, systems of parallel grammars for different languages have been developed, but neither formalism is multilingual in the way GF/PGF is. The currently most popular implementations are LKB (Copestake 2002) for HPSG and XLE (Kaplan and Maxwell 2007) for LFG. To our knowledge, neither formalism supports generation of portable low-level code.

An emerging species of embedded grammar applications is language models for speech recognition. Regulus (Rayner et al. 2006) is a system that compiles high-level feature-based grammars into low-level context-free grammars in the Nuance format (Nua 2003). PGF can likewise be compiled into Nuance and a host of other speech grammar formats (Bringert 2007a).

Type-theoretical grammar formats based on Curry's distinction between tectogrammar and phenogrammar have gained popularity in the recent years: ACG (Abstract Categorial Grammars) (de Groote 2001), HOG (Higher-Order Grammars) (Pollard 2004), and Lambda Grammars (Muskens 2001) are the most well-known examples besides GF. The work in implementing these formalisms has not come so far as in GF. One obvious idea for implementing them is to use compilation to PGF. But it remains to be seen if PGF is general enough to support this. For instance, ACG is more general than PGF in the sense that linearization types can be function types, but less general in the sense that functions have to be linear (that is, use every argument exactly once). This means that the style of defining functions is very different, for instance, that a rule written with multiple occurrences of a variable in PGF is in ACG encoded as a higher-order function.

PMCFG, while known for almost two decades and having nice computational properties, has not been used for practical grammar writing. Even the use of PMCFG as a target format for grammar compilation seems to be new to the GF project.

2.8 Conclusion

PGF was first created as a low-level target format for compiling high-level grammars written in GF. The division between high-level source formats and low-level target formats has known advantages in programming language design, which have been confirmed in the case of GF and PGF. One distinguishing property is redundancy: the absence of redundancy from PGF makes is maximally easy to write PGF interpreters, to compile PGF to other formats, and to reason about PGF. In GF, on the other hand, computationally redundant features, such as intensional type distinctions, inlinable functions, and separately compilable modules, support the programmer's work by permitting useful error messages and an abstract programming style. GF as compiled to PGF has made it possible to build grammar-based systems that combine linguistic coverage (the GF resource grammar library) with efficient run-time behaviour (mostly linear-time generation, incremental PMCFG parsing) and integration with other system components (web pages via JavaScript, spoken language models via context-free approximations). For other grammar formalisms than GF, compilation to PGF could be used as an economical implementation technique, which would moreover make it possible to link together grammars written in different high-level formalisms.

Chapter 3

Incremental Parsing with PMCFG

3.1 Introduction

Parallel Multiple Context-Free Grammar (PMCFG) (Seki et al. 1991) is one of the grammar formalisms that have been proposed for the syntax of natural languages. It is an extension of context-free grammar (CFG) where the right hand side of the production rule is a tuple of strings instead of only one string. Using tuples the grammar can model discontinuous constituents which makes it more powerful than context-free grammar. In the same time PMCFG has the advantage to be parseable in polynomial time which makes it attractive from computational point of view.

A parsing algorithm is incremental if it reads the input one token at the time and calculates all possible consequences of the token, before the next token is read. There is substantial evidence showing that humans process language in an incremental fashion which makes the incremental algorithms attractive from cognitive point of view.

If the algorithm is also top-down then it is possible to predict the next word from the sequence of preceding words using the grammar. This can be used for example in text based dialog systems or text editors for controlled language where the user might not be aware of the grammar coverage. In this case the system can suggest the possible continuations.

A restricted form of PMCFG that is still stronger than CFG is Multiple Context-Free Grammar (MCFG). In Seki and Kato (2008) it has been shown that MCFG is equivalent to string-based Linear Context-Free Rewriting Systems and Finite-Copying Tree Transducers and it is stronger than Tree Adjoining Grammars (Joshi

and Schabes 1997). Efficient recognition and parsing algorithms for MCFG have been described in Nakanishi et al. (1997), Ljunglöf (2004) and Burden and Ljunglöf (2005). They can be used with PMCFG also but it has to be approximated with overgenerating MCFG and post processing is needed to filter out the spurious parsing trees.

We present a parsing algorithm that is incremental, top-down and supports PMCFG directly. The algorithm exploits a view of PMCFG as an infinite context-free grammar where new context-free categories and productions are generated during parsing. It is trivial to turn the algorithm into statistical by attaching probabilities to each rule.

In Ljunglöf (2004) it has been shown that the **Grammatical Framework** (GF) formalism (Ranta 2004b) is equivalent to PMCFG. The algorithm was implemented as part of the GF interpreter and was evaluated with the **resource grammar library** (Ranta 2008) which is the largest collection of grammars written in this formalism. The incrementality was used to build a help system which suggests the next possible words to the user.

Formal definition of PMCFG has already been given in section 2.4.1. In section 3.2 the procedure for "linearization" i.e. the derivation of string from syntax tree is defined. The definition is needed for better understanding of the formal proofs in the paper. The algorithm introduction starts with informal description of the idea in section 3.3 and after that the formal rules are given in section 3.4. The implementation details are outlined in section 3.5 and after that there are some comments on the evaluation in section 3.6. Section 3.7 gives a conclusion.

3.2 Derivation

The derivation of a string in PMCFG is a two-step process. First we have to build a syntax tree of a category S and after that to linearize this tree to string. The definition of a syntax tree is recursive:

Definition 2 *$(f\ t_1 \dots t_{a(f)})$ is a tree of category A if t_i is a tree of category B_i and there is a production:*

$A \to f[B_1 \dots B_{a(f)}]$

The abstract notation for "t is a tree of category A" is $t : A$. When $a(f) = 0$ then the tree does not have children and the node is called leaf.

The linearization is bottom-up. The functions in the leaves do not have arguments so the tuples in their definitions already contain constant strings. If the

function has arguments then they have to be linearized and the results combined. Formally this can be defined as a function $\mathscr{L}$ applied to the syntax tree:

$$\begin{aligned}&\mathscr{L}(f\ t_1\ t_2 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{r(f)})\\ &\quad \text{where } x_i = \mathscr{K}\,(\mathscr{L}(t_1), \mathscr{L}(t_2) \ldots \mathscr{L}(t_{a(f)}))\ \alpha_i\\ &\quad \text{and } f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)}) \in F\end{aligned}$$

The function uses a helper function $\mathscr{K}$ which takes the already linearized arguments and a sequence α_i of terminals and $\langle k;l\rangle$ pairs and returns a string. The string is produced by simple substitution of each $\langle k;l\rangle$ with the string for constituent l from argument k:

$$\mathscr{K}\ \sigma\ (\beta_1\langle k_1;l_1\rangle\beta_2\langle k_2;l_2\rangle \ldots \beta_n) = \beta_1\sigma_{k_1l_1}\beta_2\sigma_{k_2l_2}\ldots\beta_n$$

where $\beta_i \in T^*$. The recursion in $\mathscr{L}$ terminates when a leaf is reached.

In the example $a^nb^nc^n$ language the function z does not have arguments and it corresponds to the base case when $n = 0$. Every application of s over another tree $t : N$ increases n by one. For example the syntax tree $(s\ (s\ z))$ will produce the tuple (aa, bb, cc). Finally the application of c combines all elements in the tuple in a single string i.e. $c\ (s\ (s\ z))$ will produce the string *aabbcc*.

3.3 The Idea

Although PMCFG is not context-free it can be approximated with an overgenerating context-free grammar. The problem with this approach is that the parser produces many spurious parse trees that have to be filtered out. A direct parsing algorithm for PMCFG should avoid this and a careful look at the difference between PMCFG and CFG gives an idea. The context-free approximation of $a^nb^nc^n$ is the language $a^*b^*c^*$ with grammar:

$$\begin{aligned}S &\to ABC\\ A &\to \varepsilon \mid aA\\ B &\to \varepsilon \mid bB\\ C &\to \varepsilon \mid cC\end{aligned}$$

The string "*aabbcc*" is in the language and it can be derived with the following steps:

$$
\begin{aligned}
& S \\
\Rightarrow\ & ABC \\
\Rightarrow\ & aABC \\
\Rightarrow\ & aaABC \\
\Rightarrow\ & aaBC \\
\Rightarrow\ & aabBC \\
\Rightarrow\ & aabbBC \\
\Rightarrow\ & aabbC \\
\Rightarrow\ & aabbcC \\
\Rightarrow\ & aabbccC \\
\Rightarrow\ & aabbcc
\end{aligned}
$$

The grammar is only an approximation because there is no enforcement that we will use only equal number of reductions for A, B and C. This can be guaranteed if we replace B and C with new categories B' and C' after the derivation of A:

$$
\begin{aligned}
B' &\rightarrow bB'' & \qquad C' &\rightarrow cC'' \\
B'' &\rightarrow bB''' & \qquad C'' &\rightarrow cC''' \\
B''' &\rightarrow \varepsilon & \qquad C''' &\rightarrow \varepsilon
\end{aligned}
$$

In this case the only possible derivation from $aaB'C'$ is *aabbcc*.

The PMCFG parser presented in this paper works like context-free parser, except that during the parsing it generates fresh categories and rules which are specializations of the originals. The newly generated rules are always versions of already existing rules where some category is replaced with new more specialized category. The generation of specialized categories prevents the parser from recognizing phrases that are otherwise withing the scope of the context-free approximation of the original grammar.

3.4 Parsing

The algorithm is described as a deductive process in the style of Shieber et al. (1995). The process derives a set of items where each item is a statement about the grammatical status of some substring in the input.

The inference rules are in natural deduction style:

$$\frac{X_1 \ldots X_n}{Y} < \text{side conditions on } X_1, \ldots, X_n >$$

where the premises X_i are some items and Y is the derived item. We assume that $w_1 \ldots w_n$ is the input string.

3.4.1 Deduction Rules

The deduction system deals with three types of items: active, passive and production items.

Productions In Shieber's deduction systems the grammar is a constant and the existence of a given production is specified as a side condition. In our case the grammar is incrementally extended at runtime, so the set of productions is part of the deduction set. The productions from the original grammar are axioms and are included in the initial deduction set.

Active Items The active items represent the partial parsing result:
$[{}^k_j A \rightarrow f[\vec{B}]; l : \alpha \bullet \beta]\,, \quad j \leq k$
The interpretation is that there is a function f with a corresponding production:

$$A \rightarrow f[\vec{B}]$$
$$f := (\gamma_1, \ldots \gamma_{l-1}, \alpha\beta, \ldots \gamma_{r(f)})$$

such that the tree $(f\ t_1 \ldots t_{a(f)})$ will produce the substring $w_{j+1} \ldots w_k$ as a prefix in constituent l for any sequence of arguments $t_i : B_i$. The sequence α is the part that produced the substring:

$$\mathscr{K}(\mathscr{L}(t_1), \mathscr{L}(t_2) \ldots \mathscr{L}(t_{a(f)}))\ \alpha = w_{j+1} \ldots w_k$$

and β is the part that is not processed yet.

Passive Items The passive items are of the form:
$[{}^k_j A; l; N]\,, \quad j \leq k$

INITIAL PREDICT

$$\frac{S \to f[\vec{B}]}{[^0_0 S \to f[\vec{B}]; 1 : \bullet\alpha]} \quad S \text{ - start category}, \alpha = \text{rhs}(f,1)$$

PREDICT

$$\frac{B_d \to g[\vec{C}] \qquad [^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r\rangle\ \beta]}{[^k_k B_d \to g[\vec{C}]; r : \bullet\gamma]}\ \gamma = \text{rhs}(g,r)$$

SCAN

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha \bullet s\ \beta]}{[^{k+1}_j A \to f[\vec{B}]; l : \alpha\ s \bullet \beta]} \quad s = w_{k+1}$$

COMPLETE

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha\bullet]}{N \to f[\vec{B}] \qquad [^k_j A; l; N]} \quad N = (A, l, j, k)$$

COMBINE

$$\frac{[^u_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r\rangle\ \beta] \qquad [^k_u B_d; r; N]}{[^k_j A \to f[\vec{B}\{d := N\}]; l : \alpha\ \langle d; r\rangle \bullet \beta]}$$

Figure 3.1: Deduction Rules

and state that there exists at least one production:

$$A \to f[\vec{B}]$$
$$f := (\gamma_1, \gamma_2, \ldots \gamma_{r(f)})$$

and a tree $(f\ t_1 \ldots t_{a(f)}) : A$ such that the constituent with index l in the linearization of the tree is equal to $w_{j+1} \ldots w_k$. Contrary to the active items in the passive the whole constituent is matched:

$$\mathscr{K}(\mathscr{L}(t_1), \mathscr{L}(t_2) \ldots \mathscr{L}(t_{a(f)}))\ \gamma_l = w_{j+1} \ldots w_k$$

Each time when we complete an active item, a passive item is created and at the same time we create a new category N which accumulates all productions for A that produce the $w_{j+1} \ldots w_k$ substring from constituent l. All trees of category N must produce $w_{j+1} \ldots w_k$ in the constituent l.

There are six inference rules (see figure 3.1).

The INITIAL PREDICT rule derives one item spanning the $0-0$ range for each production with the start category S on the left hand side. The $\text{rhs}(f,l)$ function returns the constituent with index l of function f.

In the PREDICT rule, for each active item with dot before a $\langle d;r\rangle$ pair and for each production for B_d, a new active item is derived where the dot is in the beginning of constituent r in g.

When the dot is before some terminal s and s is equal to the current terminal w_k then the SCAN rule derives a new item where the dot is moved to the next position.

When the dot is at the end of an active item then it is converted to passive item in the COMPLETE rule. The category N in the passive item is a fresh category created for each unique (A,l,j,k) quadruple. A new production is derived for N which has the same function and arguments as in the active item.

The item in the premise of COMPLETE was at some point predicted in PREDICT from some other item. The COMBINE rule will later replace the occurence A in the original item (the premise of PREDICT) with the specialization N.

The COMBINE rule has two premises: one active item and one passive. The passive item starts from position u and the only inference rule that can derive items with different start positions is PREDICT. Also the passive item must have been predicted from active item where the dot is before $\langle d;r\rangle$, the category for argument number d must have been B_d and the item ends at u. The active item in the premise of COMBINE is such an item so it was one of the items used to predict the passive one. This means that we can move the dot after $\langle d;r\rangle$ and the d-th argument is replaced with its specialization N.

If the string β contains another reference to the d-th argument then the next time when it has to be predicted the rule PREDICT will generate active items, only for those productions that were successfully used to parse the previous constituents. If a context-free approximation was used this would have been equivalent to unification of the redundant subtrees. Instead this is done at runtime which also reduces the search space.

The parsing is successful if we had derived the $[{}_0^n S;1;S']$ item, where n is the length of the text, S is the start category and S' is the newly created category.

The parser is incremental because all active items span up to position k and the only way to move to the next position is the SCAN rule where a new symbol from the input is consumed.

3.4.2 Soundness

The parsing system is sound if every derivable item represents a valid grammatical statement under the interpretation given to every type of item.

The derivation in INITIAL PREDICT and PREDICT is sound because the item is derived from existing production and the string before the dot is empty so:

$$\mathscr{K}\ \sigma\ \varepsilon = \varepsilon$$

The rationale for SCAN is that if

$$\mathscr{K}\ \sigma\ \alpha = w_{j-1} \ldots w_k$$

and $s = w_{k+1}$ then

$$\mathscr{K}\ \sigma\ (\alpha\ s) = w_{j-1} \ldots w_{k+1}$$

If the item in the premise is valid then it is based on existing production and function and so will be the item in the consequent.

In the COMPLETE rule the dot is at the end of the string. This means that $w_{j+1} \ldots w_k$ will be not just a prefix in constituent l of the linearization but the full string. This is exactly what is required in the semantics of the passive item. The passive item is derived from a valid active item so there is at least one production for A. The category N is unique for each (A, l, j, k) quadruple so it uniquely identifies the passive item in which it is placed. There might be many productions that can produce the passive item but all of them should be able to generate $w_{j+1} \ldots w_k$ and they are exactly the productions that are added to N. From all this arguments it follows that COMPLETE is sound.

The COMBINE rule is sound because from the active item in the premise we know that:

$$\mathscr{K}\ \sigma\ \alpha = w_{j+1} \ldots w_u$$

for every context σ built from the trees:

$$t_1 : B_1;\ t_2 : B_2;\ \ldots t_{a(f)} : B_{a(f)}$$

From the passive item we know that every production for N produces the $w_{u+1} \ldots w_k$ in r. From that follows that

$$\mathscr{K}\ \sigma'\ (\alpha\langle d; r\rangle) = w_{j+1} \ldots w_k$$

where σ' is the same as σ except that B_d is replaced with N. Note that the last conclusion will not hold if we were using the original context because B_d is a more general category and can contain productions that does not derive $w_{u+1} \ldots w_k$.

3.4.3 Completeness

The parsing system is complete if it derives an item for every valid grammatical statement. In our case we have to prove that for every possible parse tree the corresponding items will be derived.

The proof for completeness requires the following lemma:

Lemma 1 *For every possible syntax tree*

$$(f\ t_1 \ldots t_{a(f)}) : A$$

with linearization

$$\mathscr{L}(f t_1 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{d(A)})$$

where $x_l = w_{j+1} \ldots w_k$*, the system will derive an item* $[{}^k_j A; l; A']$ *if the item* $[{}^k_j A \rightarrow f[\vec{B}]; l : \bullet \alpha_l]$ *was predicted before that. We assume that the function definition is:*

$$f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)})$$

The proof is by induction on the depth of the tree. If the tree has only one level then the function f does not have arguments and from the linearization definition and from the premise in the lemma it follows that $\alpha_l = w_{j+1} \ldots w_k$. From the active item in the lemma by applying iteratively the SCAN rule and finally the COMPLETE rule the system will derive the requested item.

If the tree has subtrees then we assume that the lemma is true for every subtree and we prove it for the whole tree. We know that

$$\mathscr{K}\ \sigma\ \alpha_l = w_{j+1} \ldots w_k$$

Since the function $\mathscr{K}$ does simple substitution it is possible for each $\langle d; s \rangle$ pair in α_l to find a new range in the input string $j' - k'$ such that the lemma to be applicable for the corresponding subtree $t_d : B_d$. The terminals in α_l will be processed by the SCAN rule. Rule PREDICT will generate the active items required for the subtrees and the COMBINE rule will consume the produced passive items. Finally the COMPLETE rule will derive the requested item for the whole tree.

From the lemma we can prove the completeness of the parsing system. For every possible tree $t : S$ such that $\mathscr{L}(t) = (w_1 \ldots w_n)$ we have to prove that the $[{}^n_0 S; 1; S']$ item will be derived. Since the top-level function of the tree must be from production for S the INITIAL PREDICT rule will generate the active item in the premise of the lemma. From this and from the assumptions for t it follows that the requested passive item will be derived.

3.4.4 Complexity

The algorithm is very similar to the Earley (1970) algorithm for context-free grammars. The similarity is even more apparent when the inference rules in this paper are compared to the inference rules for the Earley algorithm presented in Shieber et al. (1995) and Ljunglöf (2004). This suggests that the space and time complexity of the PMCFG parser should be similar to the complexity of the Earley parser which is $\mathscr{O}(n^2)$ for space and $\mathscr{O}(n^3)$ for time. However we generate new categories and productions at runtime and this have to be taken into account.

Let the $\mathscr{P}(j)$ function be the maximal number of productions generated from the beginning up to the state where the parser has just consumed terminal number j. $\mathscr{P}(j)$ is also the upper limit for the number of categories created because in the worst case there will be only one production for each new category.

The active items have two variables that directly depend on the input size - the start index j and the end index k. If an item starts at position j then there are $(n-j+1)$ possible values for k because $j \leq k \leq n$. The item also contains a production and there are $\mathscr{P}(j)$ possible choices for it. In total there are:

$$\sum_{j=0}^{n}(n-j+1)\mathscr{P}(j)$$

possible choices for one active item. The possibilities for all other variables are only a constant factor. The $\mathscr{P}(j)$ function is monotonic because the algorithm only adds new productions and never removes. From that follows the inequality:

$$\sum_{j=0}^{n}(n-j+1)\mathscr{P}(j) \leq \mathscr{P}(n)\sum_{i=0}^{n}(n-j+1)$$

which gives the approximation for the upper limit:

$$\mathscr{P}(n)\frac{n(n+1)}{2}$$

The same result applies to the passive items. The only difference is that the passive items have only a category instead of a full production. However the upper limit for the number of categories is the same. Finally the upper limit for the total number of active, passive and production items is:

$$\mathscr{P}(n)(n^2+n+1)$$

The expression for $\mathscr{P}(n)$ is grammar dependent but we can estimate that it is polynomial because the set of productions corresponds to the compact representation of all parse trees in the context-free approximation of the grammar. The exponent however is grammar dependent. From this we can expect that asymptotic space complexity will be $\mathscr{O}(n^e)$ where e is some parameter for the grammar. This is consistent with the results in Nakanishi et al. (1997) and Ljunglöf (2004) where the exponent also depends on the grammar.

The time complexity is proportional to the number of items and the time needed to derive one item. The time is dominated by the most complex rule which in this algorithm is COMBINE. All variables that depend on the input size are present both in the premises and in the consequent except u. There are n possible values for u so the time complexity is $\mathscr{O}(n^{e+1})$.

3.4.5 Tree Extraction

If the parsing is successful we need a way to extract the syntax trees. Everything that we need is already in the set of newly generated productions. If the goal item is $[{}^{n}_{0}S;0;S']$ then every tree t of category S' that can be constructed is a syntax tree for the input sentence (see definition 2 in section 3.2 again).

Note that the grammar can be erasing; i.e., there might be productions like this:

$$S \to f[B_1, B_2, B_3]$$
$$f := (\langle 1;1\rangle\langle 3;1\rangle)$$

There are three arguments but only two of them are used. When the string is parsed this will generate a new specialized production:

$$S' \to f[B'_1, B_2, B'_3]$$

Here S,B_1 and B_3 are specialized to S', B'_1 and B'_3 but the B_2 category is still the same. This is correct because actually any subtree for the second argument will produce the same result. Despite this it is sometimes useful to know which parts of the tree were used and which were not. In the GF interpreter such unused branches are replaced by meta variables. In this case the tree extractor should check whether the category also exists in the original set of categories N in the grammar.

Just like with the context-free grammars the parsing algorithm is polynomial but the chart can contain exponential or even infinite number of trees. Despite this the chart is a compact finite representation of the set of trees.

3.5 Implementation

Every implementation requires a careful design of the data structures in the parser. For efficient access the set of items is split into four subsets: $\mathbb{A}$, $\mathbb{S}_j$, $\mathbb{C}$ and $\mathbb{P}$. $\mathbb{A}$ is the agenda i.e. the set of active items that have to be analyzed. $\mathbb{S}_j$ contains items for which the dot is before an argument reference and which span up to position j. $\mathbb{C}$ is the set of possible continuations i.e. a set of items for which the dot is just after a terminal. $\mathbb{P}$ is the set of productions. In addition the set $\mathbb{F}$ is used internally for the generatation of fresh categories. The sets $\mathbb{C}$, $\mathbb{S}_j$ and $\mathbb{F}$ are used as association maps. They contain associations like $k \mapsto v$ where k is the key and v is the value. All maps except $\mathbb{F}$ can contain more than one value for one and the same key.

The pseudocode of the implementation is given in figure 3.2. There are two procedures *Init* and *Compute*.

Init computes the initial values of $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{A}$. The initial agenda $\mathbb{A}$ is the set of all items that can be predicted from the start category S (INITIAL PREDICT rule).

Compute consumes items from the current agenda and applies the SCAN, PREDICT, COMBINE or COMPLETE rule. The case statement matches the current item against the patterns of the rules and selects the proper rule. The PREDICT and COMBINE rules have two premises so they are used in two places. In both cases one of the premises is related to the current item and a loop is needed to find item matching the other premis.

The passive items are not independent entities but are just the combination of key and value in the set $\mathbb{F}$. Only the start position of every item is kept because the end position for the interesting passive items is always the current position and the active items are either in the agenda if they end at the current position or they are in the $\mathbb{S}_j$ set if they end at position j. The active items also keep only the dot position in the constituent because the constituent definition can be retrieved from the grammar. For this reason the runtime representation of the items is $[j; A \to f[\vec{B}]; l; p]$ where j is the start position of the item and p is the dot position inside the constituent.

The *Compute* function returns the updated $\mathbb{S}$ and $\mathbb{P}$ sets and the set of possible continuations $\mathbb{C}$. The set of continuations is a map indexed by a terminal and the

values are active items. The parser computes the set of continuations at each step and if the current terminal is one of the keys the set of values for it is taken as an agenda for the next step.

3.6 Evaluation

The algorithm was evaluated with all languages from the GF Resource Grammar Library (Ranta 2008). The grammars are not primarily intended for parsing but as a resource from which smaller domain dependent grammars are derived for every application. Despite this, the resource grammar library is a good benchmark for the parser because these are the biggest GF grammars.

The compiler converts a grammar written in the high-level GF language to a low-level PMCFG grammar which the parser can use directly. The sizes of the grammars in terms of number of productions and number of unique discontinuous constituents are given on table 2.4. The number of constituents roughly corresponds to the number of productions in the context-free approximation of the grammar. The experiments showed that although the theoretical complexity is polynomial, the real-time performance for practically interesting grammars tends to be linear. Figure 3.3 shows the time in miliseconds needed to parse a sentence of given length. The slowest grammar is for Finnish, after that are German and Italian, followed by the Romance languages French, Spanish and Catalan plus Bulgarian and Interlingua. The most efficient is the English grammar followed by the Scandinavian languages Danish, Norwegian and Swedish. Since the parsing time is linear, figure 3.3 shows only the constant factor for each language i.e. the average time needed to accept one token.

3.7 Conclusion

The algorithm has proven useful in the GF system. It accomplished the initial goal to provide suggestions in text based dialog systems and in editors for controlled languages. Additionally the algorithm has properties that were not envisaged in the beginning. It works with PMCFG directly rather that by approximation with MCFG or some other weaker formalism.

Since the Linear Context-Free Rewriting Systems, Finite-Copying Tree Transducers and Tree Adjoining Grammars can be converted to PMCFG, the algorithm presented in this paper can be used with the converted grammar. The approach

procedure Init() {
  $k = 0$
  $\mathbb{S}_i = \emptyset$, *for every i*
  $\mathbb{P}$ = *the set of productions P in the grammar*

  $\mathbb{A} = \emptyset$
  forall $S \to f[\vec{B}] \in P$ **do** // INITIAL PREDICT
    $\mathbb{A} = \mathbb{A} + [0; S \to f[\vec{B}]; 1; 0]$

  return $(\mathbb{S}, \mathbb{P}, \mathbb{A})$
}

procedure Compute$(k, (\mathbb{S}, \mathbb{P}, \mathbb{A}))$ {
  $\mathbb{C} = \emptyset$
  $\mathbb{F} = \emptyset$
  while $\mathbb{A} \neq \emptyset$ **do** {
    let $x \in \mathbb{A}$, $x \equiv [j; A \to f[\vec{B}]; l; p]$
    $\mathbb{A} = \mathbb{A} - x$
    case *the dot in x* **is** {
      before $s \in T \Rightarrow \mathbb{C} = \mathbb{C} + (s \mapsto [j; A \to f[\vec{B}]; l; p+1])$ // SCAN

      before $\langle d; r \rangle \Rightarrow$ **if** $((B_d, r) \mapsto (x, d)) \notin \mathbb{S}_k$ **then** {
          $\mathbb{S}_k = \mathbb{S}_k + ((B_d, r) \mapsto (x, d))$
          forall $B_d \to g[\vec{C}] \in \mathbb{P}$ **do** // PREDICT
            $\mathbb{A} = \mathbb{A} + [k; B_d \to g[\vec{C}]; r; 0]$
        }
        forall $(k; B_d, r) \mapsto N \in \mathbb{F}$ **do** // COMBINE
          $\mathbb{A} = \mathbb{A} + [j; A \to f[\vec{B}\{d := N\}]; l; p+1]$

      at the end $\Rightarrow$ **if** $\exists N.((j, A, l) \mapsto N \in \mathbb{F})$ **then** {
          forall $(N, r) \mapsto (x', d') \in \mathbb{S}_k$ **do** // PREDICT
            $\mathbb{A} = \mathbb{A} + [k; N \to f[\vec{B}]; r; 0]$
        } else {
          generate fresh N // COMPLETE
          $\mathbb{F} = \mathbb{F} + ((j, A, l) \mapsto N)$
          forall $(A, l) \mapsto ([j'; A' \to f'[\vec{B'}]; l'; p'], d) \in \mathbb{S}_j$ **do** // COMBINE
            $\mathbb{A} = \mathbb{A} + [j'; A' \to f'[\vec{B'}\{d := N\}]; l'; p'+1]$
        }
        $\mathbb{P} = \mathbb{P} + (N \to f[\vec{B}])$
    }
  }
  return $(\mathbb{S}, \mathbb{P}, \mathbb{C})$
}

Figure 3.2: Pseudocode of the parser implementation

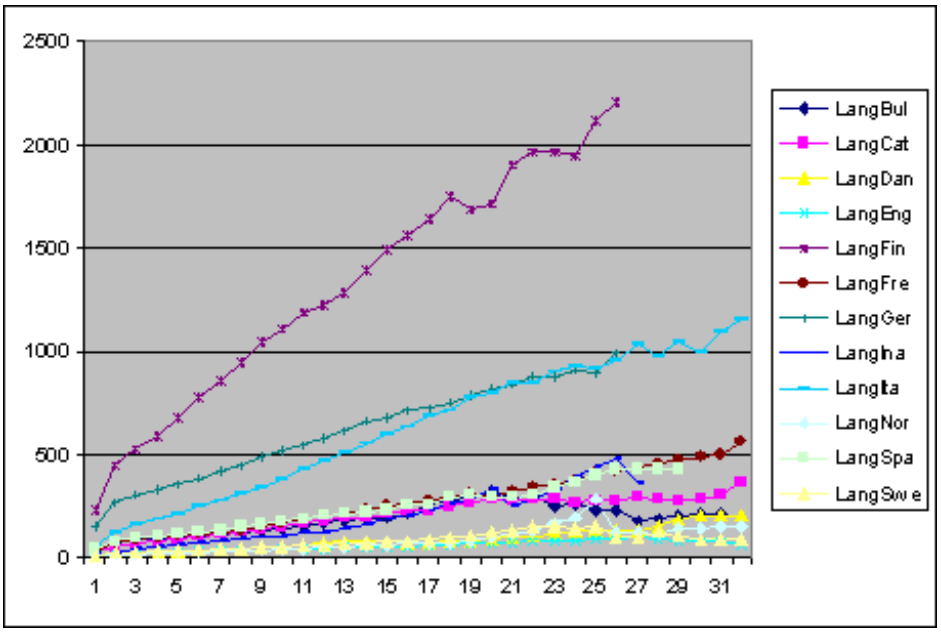


Figure 3.3: Parser performance in miliseconds per token

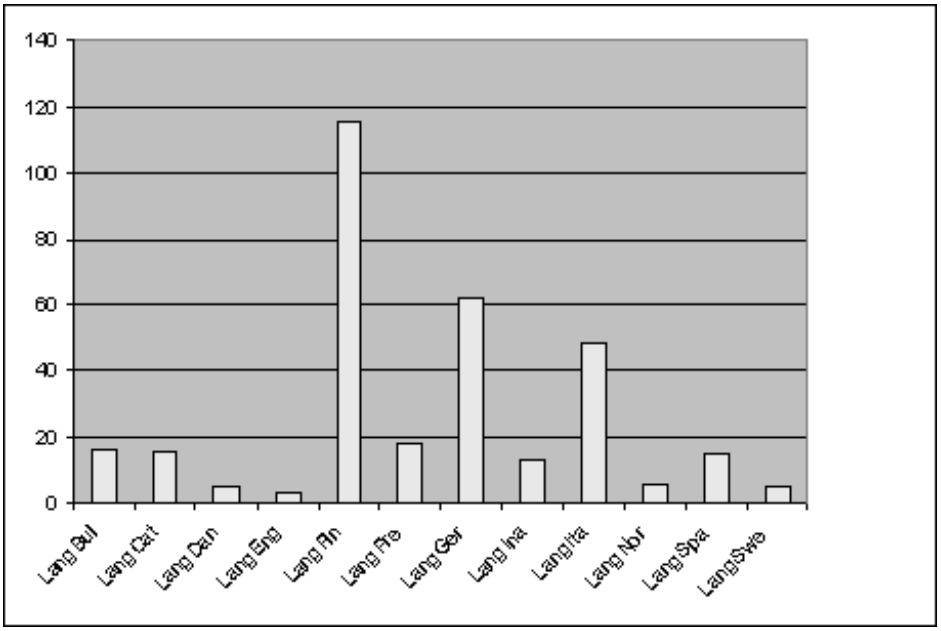


Figure 3.4: The average time in miliseconds to accept a new token in different languages

to represent context-dependent grammar as infinite context-free grammar might be applicable to other formalisms as well. This will make it very attractive in applications where some of the other formalisms are already in use.

Chapter 4

Type-Theoretical Bulgarian Grammar

4.1 Introduction

Grammatical Framework (GF) (Ranta 2004b) is a grammar formalism based on the type theory and there is a high-level language with compiler and interpreter for it. Within the framework a resource grammar library was developed that contains grammars for ten languages: one Slavic language: Russian (Khegai 2006b), three Scandinavian languages: Swedish, Danish and Norwegian, three Romance languages: Italian, Spanish and French, two Germanic languages: English and German and finally one Finno-Ugric language: Finnish. There is also a still unfinished grammar for Arabic (Dada and Ranta 2007). We added the eleventh language - Bulgarian.

Although this is not the first Slavic language in the library, this is the first South-Slavic language. Russian itself is East-Slavic and it differs significantly both in morphology and syntax. Furthermore, Bulgarian belongs to the Balkan linguistic area and has some characteristic properties such as a complete loss of case declension, lack of verb infinitive forms and the development of a definite article.

The GF grammars can share common code via inheritance and parametricity (Ranta 2007). This is already used to capture the similarities between the Scandinavian and the Romance languages. There are common Scandinavian and Romance modules from which the concrete grammars are inherited. Although this can not be easily done for Bulgarian and Russian, the Bulgarian grammar is

a good common ground candidate for Serbian and Macedonian since they are of the same family and of the same language area.

The grammar has two levels in GF: abstract and concrete. The abstract level is language independent and represents the original textual sentence as a typed lambda term (abstract syntax). The concrete level gives the mapping from a given lambda term to its concrete textual representation. It is possible to have multiple concrete syntaxes for each abstract syntax. This allows one language to be translated into another or to simultaneously generate equivalent texts in different languages.

Furthermore, the grammars in GF are divided into resource grammars and application grammars. The resource grammars have a wider coverage and are linguistically oriented but they can be highly ambiguous. The application grammars reuse the already existing resource grammars but specialize them in a specific context. In this way they are less ambiguous and usually they are extended with an application specific dictionary.

The grammar that we developed is a resource grammar. On top of it, the developer could create an application specific grammar with a very limited linguistic knowledge. It could be started with an English grammar which then could be translated to Bulgarian just by introduction of new lexicon since the abstract syntax representation is common for all languages.

The current abstraction proved to be enough for many dialog and text generation projects (KeY, TALK, WebALT). For that reason constructions that are not covered in the current abstract syntax are out of the scope of the current project. We found that the current design provides sufficient abstraction and we did not have to change it in order to fit the new language.

In the next sections we first explain the morphology and then the most important parts of the grammar: noun, verb and adjective phrases, numerals, declarative sentences, questions and imperative utterances. Comparative examples are given mainly in English but where it is more appropriate there are also examples from other languages.

4.2 Morphology

The grammar has a complete morphology for adjectives, nouns, verbs, numerals and pronouns. The full word classification is given in (Krustev 1984) where the words are divided in 187 paradigms. Each paradigm in GF is defined as a function that takes the base form and produces a table with all possible word forms.

GF has a small parallel dictionary of 350 words that is translated to all languages and is used mainly for development and testing purposes. Another small dictionary of about one hundred words contains structural words like pronouns, prepositions and quantifiers. These dictionaries are translated to Bulgarian as well. In addition there is a bigger dictionary of 57805 words that is only for Bulgarian and defines the base form, the part of speech category and its inflection paradigm. The dictionary is an import of all adjectives, nouns and verbs from the BGOffice project.[1]

In the abstract syntax the words are defined as constants of some of the lexical categories. For example in the Lexicon the words red, go and apple are declared as:

fun *red_A* : *A* ;
go_V : *V* ;
apple_N : *N* ;

The corresponding linearization rules in the concrete syntax for Bulgarian are[2]:

lin *red_A* = *mkA076* "červen" ;
apple_N = *mkN041* "jabâlka" ;
go_V = *actionV* (*mkV186* "otivam") (*mkV146* "otida") ;

Here the numbers 076, 041, 186 and 146 are the paradigm numbers given in Krustev (Krustev 1984). The verbs have two base forms which correspond to perfective and imperfective aspect. The usage of verb aspect is explained in section 4.4.

For comparison we also give the linearization rules for English:

lin *red_A* = *duplADeg* "red" ;
apple_N = *regN* "apple" ;
go_V = *mk5V* "go" "goes" "went" "gone" "going" ;

In English *apple* is a regular noun and it is defined with the *regN* function. In contrast *red* is defined with *duplADeg* because it doubles the ending consonant in the comparative form. The verb *go* itself is irregular and it is defined by enumerating all its forms.

The application grammars are supposed to use their own context specific dictionaries but when it is appropriate they can reuse the already existing one. The advantage of having a context specific dictionary is that it is much less ambiguous. For example the Bulgarian word *vreme* in general has the meaning of either time, weather or tense. Despite this an application in the weather forecast area will use

[1] `http://bgoffice.sourceforge.net`

[2] In the paper we use the scientific transliteration of cyrilic but the actual grammar is in cyrilic.

the second translation while an application for airport service is more likely to use the first one and neither of them will use the tense meaning.

4.3 Noun Phrases

The nouns in Bulgarian are divided into four genders: masculine animate, masculine inanimate, feminine and neuter. There are two numbers singular and plural. The main noun forms are illustrated on fig. 4.1.

In the sentence the noun modifiers (adjectives, numerals) are in gender and number agreement with the noun. The inflection for all modifiers except the cardinal numerals does not distinguish between masculine animate and masculine inanimate so they have effectively merged into a single gender. For the cardinals there is a distinct masculine animate form (mážkolična forma).

dvama máže two men
dva učebnika two textbooks

The animate gender includes all human masculine words like man, teacher and king while others like *kon* (horse) are inanimate despite that they are alive. Most informal grammars do not mention the existence of four genders and describe the masculine animate form as an exception. In the formalized rules it behaves exactly as a separate gender and this is not something uncommon in the other Slavic languages. For example in Czech there is a much clearer distinction between masculine animate and masculine inanimate in the declension system. In Bulgarian the animate gender is not fully developed.

In the rest of the section we will state masculine gender when there is no distinction between animate and inanimate and we will specify it explicitly otherwise.

Bulgarian and Macedonian are the only Slavic languages that developed definite article. The article is a clitic and attaches to the end of the first nominal in the noun phrase that is noun, adjective, pronoun or numeral. Its form depends on the ending of the nominal and on the case when the nominal is a masculine singular noun. The masculine nouns use full definite article (*pălen opredelitelen člen*) when they are in singular and they have the role of subject in the sentence. The definite noun forms are recorded in the inflection tables together with the singular and the plural forms.

There is also a vocative form used for a noun identifying the object being addressed: *máž**o***, *žen**o***. The vocative form is a remnant from the old vocative case in the Old Church Slavonic.

Figure 4.1: Noun Forms

Sg+Indef	Sg+Def	SgDefNom	Pl+Indef	Pl+Def	English	Gender
măž	*măž**a***	*măž**ăt***	*măž**e***	*măž**ete***	man	masc animate
učebnik	*učebnik**a***	*učebnik**ăt***	*učebnic**i***	*učebnic**ite***	textbook	masc inanimate
momče	*momče**to***	*momče**to***	*momče**ta***	*momče**tata***	boy	neut
žena	*žen**ata***	*žen**ata***	*žen**i***	*žen**ite***	woman	feminine

The masculine inanimate nouns have also a special plural form used for counting and after the determiner *njakolko* (few):

mnogo učebnici	many textbooks
njakolko učebnika	few textbooks
dva učebnika	two textbooks

Although some masculine animate nouns also have countable forms, their usage in the literary language is discouraged (Pashov 1999). One exception is the case of homonyms with animate and inanimate meanings. In this case they are two different abstract syntax constants in GF and are treated properly. The usage of the countable forms for masculine animate in our grammar is not supported.

In the concrete syntax the category *N* is represented by the record:

lincat $N = \{s : NForm \Rightarrow Str;\ g : DGender\}$;

where NForm is:

```
param NForm = NF Number Species
            | NFSgDefNom
            | NFPlCount
            | NFVocative
            ;
```

The *Number*, *Species* and *DGender* parameters are defined on figure 4.2. Here the *NF* constructor represents the common case while *NFSgDefNom*, *NFPlCount* and *NFVocative* represent the forms with full definite article, the countable form and the vocative. When the noun does not have some of the special forms then the corresponding normal singular or plural form is filled in.

The noun is the head of the common noun (*CN*) phrase. The phrase is formed by these three functions:

```
fun UseN   : N → CN ;
    UseN2  : N2 → CN ;
    ComplN2 : N2 → NP → CN ;
    AdjCN  : AP → CN → CN ;
```

Here *N2* is a category of the relational nouns and *AP* is the adjective phrase. The common noun itself is the head of the noun phrase (*NP*) formed by:

fun *DetCN* : *Det* → *CN* → *NP* ;
DetSg : *Quant* → *Ord* → *Det* ;
DetPl : *Quant* → *Num* → *Ord* → *Det* ;
every_Det, *someSg_Det*, *somePl_Det* : *Det* ;
much_Det, *many_Det* : *Det* ;
few_Det : *Det* ;

The first two determiners are synthetic and they combine quantifiers with ordinal and/or cardinal numerals. The *Quant* category specifies the definiteness. It has two members *DefArt* and *IndefArt* which in English generate the "a" and "the" articles. In Bulgarian there is no indefinite article and the noun phrase is just unchanged. When the *DefArt* is chosen then it adds the definite article to the first nominal in the noun phrase. The selection of the first nominal is ensured by a system of parameters in *DetSg*, *DetPl* and *AdjCN*.

The determiners also specify whether the noun has to be in singular, plural or countable plural form. Singular is used with *DetSg*, *DetPl* with cardinal one and with the lexical determiners *someSg_Det* (*njakoj*, *njakoja*, *njakoe*) and *much_Det* (mnogo). The determiners *somePl_Det* (niakoi), *many_Det* (mnogo) and *DetPl* without numeral (*NoNum*) require the noun to be in plural. Finally *few_Det* (nijakolko) and *DetPl* with any cardinal greater than one select a countable form for the noun.

In the *N* category the *g* field is of type *DGender* (fig. 4.2) and this allows the right numeral inflection in *DetPl*. At the same time the agreement in NP:

lincat *NP* = $\{s : Role \Rightarrow Str; a : Agr\}$;
contains only the simplified *Gender* which does not make a distinction between masculine animate and masculine inanimate. This reflects the fact that the adjectives and the verb participles have only one form for masculine.

The *Role* parameter plays the role of the case in the other languages. The role can be either subject (*RSubj*) or object (*RObj c*, where *c* is of type *Case*). In the other languages there is a nominative case which is used to mark the subject and is equivalent to our *RSubj* role. In Bulgarian it is more useful to distinguish between case and role because the case is also used as a synthesized parameter in the medial and phrasal verbs and also in the prepositions. In these situations the case is always accusative or dative and never nominative. In fact the Bulgarian language is mostly analytic and the case makes a distinction only for the pronouns and the definite forms in masculine.

Figure 4.2: Inflection Parameters

```
param  Number  = Sg | Pl ;
       Person  = P1 | P2 | P3 ;
       Gender  = Masc | Fem | Neut ;
       DGender = MascA | Masc | Fem | Neut ;
       Species = Indef | Def ;
       Case    = Acc | Dat ;

       AForm   = ASg Gender Species
               | ASgMascDefNom
               | APl Species
               ;

oper   Agr     = {gn : GenNum ; p : Person} ;
```

4.4 Verb Phrases

The verb category is the most complex. There are three simple verb tenses (present, aorist, imperfect) and three synthetic participles (perfect, plusquamperfect, present participle). These synthetic forms are used in six other compound tenses: future, past future, present perfect, past perfect, future perfect and past future perfect. There are also passive voice and imperative, conditional and inferential moods.

In addition almost all verbs come in pairs with equivalent lexical meanings but with different aspects. For example *otivam* and *otida* are two different verbs which are both translated as "go" in English but they express different aspects in Bulgarian. The former is with imperfective aspect and represents the event in action while the letter is with perfective aspect and represents the event as a whole together with its start and end. The grammar has only one abstract syntax constant for each lexical meaning and for that reason the verbs are coupled together in pairs. Which verb will be used in the linearization depends on the grammatical tense and aspect.

The verb category V in the concrete syntax is defined as:

```
param   VForm = VPres          Number Person
              | VAorist        Number Person
              | VImperfect     Number Person
              | VPerfect       AForm
              | VPluPerfect    AForm
              | VPresPart      AForm
              | VPassive       AForm
              | VImperative    Number
              | VGerund
              ;
        VType = VNormal
              | VMedial        Case
              | VPhrasal       Case
              ;
        Aspect = Imperf | Perf ;
lincat V = {s : Aspect ⇒ VForm ⇒ Str; vtype : VType} ;
```

The inherent *VForm* parameter is a mixture of tenses, voices and moods because it enumerates all possible morphological forms. The compound tenses and moods are not listed because they are formed on the level of the *VP* category.

The *VType* parameter marks two special kinds of verbs medial and phrasal:

Bulgarian	English	VType
Az rabotja	I work	VNormal
Az se smeja	I smile	VMedial Acc
Az si spomnjam	I remember	VMedial Dat
Men me trese	I shiver	VPhrasal Acc
Na men mi lipsvaš	I miss you	VPhrasal Dat

The medial verbs express middle voice but syntactically are marked with the passive voice particle *se/si*. The difference is that they either cannot be used without it (a) or they have different meanings if they are used without it (b-c):

a	* *Az boja*	
b	*Az se kazvam ...*	My name is ...
c	*Az kazvam ...*	I say ...

The passive/middle voice is marked with the clitic *se* for accusative or *si* for dative. Similarly, the phrasal verbs are always coupled with the clitic form of the personal pronoun in accusative or dative. The number and person agreement is marked on the clitic instead of on the verb. The subject in the sentence is inflected as an object in the specified case. The choice of accusative or dative case for both the medial and phrasal verbs depends only on the verb so it is specified in its lexical definition. In the lexicon the medial and phrasal verbs are created with

these functions:

$$\begin{array}{lcl} \mathit{medialV} & : & V \to \mathit{Case} \to V \\ \mathit{phrasalV} & : & V \to \mathit{Case} \to V \end{array}$$

First a normal verb form is created with some of the mkV function and after that it is modified with $\mathit{medialV}$ or $\mathit{phrasalV}$.

A verb forms a verb phrase VP with the following constructors:

$$\begin{array}{lll} \textbf{fun} & \mathit{UseV} & : V \to \mathit{VP} \ ; \\ & \mathit{ComplV2} & : \mathit{V2} \to \mathit{NP} \to \mathit{VP} \ ; \\ & \mathit{ComplV3} & : \mathit{V3} \to \mathit{NP} \to \mathit{NP} \to \mathit{VP} \ ; \end{array}$$

Here the $\mathit{V2}$ and $\mathit{V3}$ categories have a structure similar to V and contain the transitive and ditransitive verbs. The VP category itself is defined as:

$$\begin{array}{lll} \textbf{lincat} & \mathit{VP} = \{ \ s : & \text{Tense} \\ & & \Rightarrow \mathit{Anteriority} \\ & & \Rightarrow \mathit{Polarity} \\ & & \Rightarrow \mathit{Agr} \\ & & \Rightarrow \mathit{Bool} \\ & & \Rightarrow \mathit{Aspect} \\ & & \Rightarrow \mathit{Str} \ ; \\ & \} \ ; & \end{array}$$

There are also other fields but only the s field is shown because the others are not relevant for the explanation. The first two parameters *Tense* and *Anteriority* define the tense system in the resource library. The supported tenses are present, past, future and conditional[3]. In English the anteriority distinguishes between the simple and the perfect tenses. This system of tense and anteriority is oversimplified. There are 8 different combinations of *Tense* and *Anteriority* but this is not enough to cover all Bulgarian and even some English tenses. For example there are two past tenses - past imperfect and aorist, but only the second one is currently supported in the grammar. It is usually translated as past simple in English. In addition there is a past future tense which is translated as "was going to" tense but the "going to" tense is also not supported. This is overcome in the Romance grammar where the concrete syntax has more tenses than the abstract syntax. They are not accessible from the common API but still can be used from language specific modules. Currently only the minimal number of tenses is implemented for Bulgarian. The tense system in the resource library is just an intersection between the tenses of all supported languages.

The verb phrase in Bulgarian has a complex structure where clitics and auxiliary verbs are combined with the main verb to form a phrase (Avgustinova 1997).

[3]This is actually a mood but it is defined as a tense because it is parallel to the other tenses

The basic components are the pronoun and the reflexive clitics, the *li* clitic, the *da*, *ne* and *šte* articles and the auxilary verb *săm* (be). The particle *ne* marks phrases with negative polarity (the *Polarity* parameter is *Neg*) and it is always in the beginning of the phrase. The *li* clitic marks questions and is added after the first stressed word in the verb phrase. The future tense is formed with the *šte* particle and the auxiliary verb *săm* is used in the perfect tenses.

Most languages distinguish between progressive and finite actions. In English this is expressed with the so called -ing verb forms or gerund. In the grammar the verb phrases use nonprogressive tenses by default. This could be changed with the function:

$ProgrVP : VP \rightarrow VP$

It converts the phrase "work" for example to the continuous phrase "am working". For Bulgarian the same function deals with the lexical aspect of the verb. In the lexicon all verbs come in pairs: one with perfective aspect and one with imperfective. The perfective aspect is the default but when the *ProgrVP* function is applied it is replaced with the imperfective.

An important exception is the present tense when the perfective aspect cannot be used. In this case the imperfective aspect is used regardless of whether the *ProgrVP* function is applied or not. The imperfective aspect in present tense is ambiguous and can have the meaning of both progressive and finite action. The parser from Bulgarian will produce two different abstract trees from the verb *otivam*: *(UseV go_V)* and *(ProgrVP (UseV go_V))*. In the opposite direction from abstract tree to English phrase the linearizer will produce both "go" and "going".

Another kind of ambiguity exists between the reflexive verb phrases and the phrases with passive voice. In the grammar they are created with these functions:

$ReflV2 : V2 \rightarrow VP$

$PassV2 : V2 \rightarrow VP$

It was already mentioned that the reflexive verbs are marked with the clitic *se* or *si*, but the same construction might mean passive voice as well (Pashov 1999). The following sentences:

Pesenta se pee

Pesenta e pjata (passive participle)

both have the meaning of "the song is sung" (passive voice) but the first has also the meaning of "the song sings itself" (reflexive verb). Of course the second meaning is rejected from the common knowledge that the song cannot sing. On the other side, the second sentence is ambiguous between present and present perfect while the first one is not - *pesenta se e pjala*.

The grammar does not try to disambiguate between the tenses or between pas-

sive voice and reflexive verb because this would require external common sense knowledge which is beyond the scope of the current project. Instead, the phrases marked with *se/si* are parsed as reflexive and the phrases formed with the passive participle are parsed as passive but the anteriority parameter remains ambiguous in the abstract tree.

The Balkan languages are known to lack verb infinitives and Bulgarian is not an exception. Instead, the Bulgarian language has developed the *da* complex (Avgustinova 1997). The particle *da* is placed in the beginning of the verb phrases in the cases when we expect an infinitive in the other languages. The main verb itself can be in any tense and it is aggreement with the subject.

One place where infinitives are used in the other languages is the VV category. It contains verbs which take another verb in infinitive form as a complement. For example in Russian it is *Ja dolžen hodit'* (I must go) but *Ja hožu* (I go). In English usually the infinitive coincides with the present, singular, first person. In Bulgarian these phrases are translated with *da* complex:

Az trjabva da hodja
Az hodja

here the main verb (*hodja*) is in present simple while the VV verb (*trjabva*) can be in any tense, anteriority and polarity. Although *da* complexes with other conjugations of the main verbs are also possible the abstract syntax does not have constructions that require that.

4.5 Adjective Phrases

The adjectives have forms for masculine, feminine and neuter in singular and a separate form for plural which is common for all genders. There are also definite and indefinite forms. The A category definition is:

lincat $A = \{s : AForm \Rightarrow Str\}$;

There are also comparative and superlative forms which are formed analyticaly on the AP level.

krasiv	beautiful
po-krasiv ot nego	more beautiful than he
naj-krasiv	most beautiful

The examples are generated from the following abstract syntaxes:

PositA beautiful_A
ComparA beautiful_A (UsePron he_Pron)
OrdSuperl beautiful_A

For comparison, in German the comparative and superlative forms are synthethic (schön - schöner als er - schönst) but they still use the same abstract syntax.

The AP clauses are attached to the common nouns with the function:

fun $AdjCN : AP \to CN \to CN$;

and are in gender and number agreement with the noun.

4.6 Numerals

The basic building blocks of the numerals are the digits from one to nine. They are divided in three groups. The first group includes only the digit one. It has forms for masculine (*edin*), feminine (*edna*) and neuter (*edno*). There is also a form for plural (*edni*) which is used to refer to a group as a whole. The second group includes the digit two. It has forms for masculine animate (*dvama*) , masculine inanimate (*dva*) and a common form for feminine and neuter (*dve*). All other digits are in the third group. They have separated form for masculine animate and a common form for all other genders. There are also definite and indefinite forms.

The cardinal and ordinal numbers in Bulgarian are also marked morphologicaly (*edin-părvi*, *dve-vtori*).

The *Digit* category is defined as:

param $CardOrd = NCard\ DGenderSpecies$
$\quad \mid NOrd\ AForm$;
$\quad DForm = unit \mid teen \mid ten \mid hundred$;

lincat $Digit = \{s : DForm \Rightarrow CardOrd \Rightarrow Str\}$;

The *DForm* parameter enumerates all numeral forms which are defined syntheticaly.

edno	edinadeset	deset	sto
one	eleven	ten	hundred

A similar parameter exists in the other grammars but it usually has different values. For example in Arabic there are forms only for unit and ten. All other numerals are formed analytically. Since the parameters are only in the concrete syntax on an abstract level, the numerals have uniform representation in all languages.

4.7 Clauses and Declarative Sentences

The simplest clause is formed by *PredVP*:

lincat $Cl = \{s : Tense \Rightarrow Anteriority \Rightarrow Polarity \Rightarrow Order \Rightarrow Str\}$;
fun $PredVP : NP \rightarrow VP \rightarrow Cl$;

It just combines one noun phrase and one verb phrase. The *NP* category has an agreement as a synthesized parameter which is passed by *PredVP* to the verb as an inherent parameter. This guarantees the subject-verb agreement in the clause. The clause does not have fixed tense, anteriority and polarity. Instead they are given as an inherent parameters which are fixed on the sentence (S) level:

fun $UseCl : Tense \rightarrow Ant \rightarrow Pol \rightarrow Cl \rightarrow S$;

Here *Tense*, *Ant* and *Pol* are syntactic categories whose constructors does not have linearizations but just fix the corresponding *Tense*, *Anteriority* and *Polarity* parameters in *Cl*. The constructed sentences can be used to construct utterances or they could be embedded in other sentences.

4.8 Imperative Sentences

The GF resource grammars have a limited support for imperative sentences. Basically, you can turn any verb phrase to *Imp* and from it an imperative utterance can be formed with positive or negative polarity:

fun $ImpVP : VP \rightarrow Imp$;
$UttImpSg, UttImpPl, UttImpPol : Pol \rightarrow Imp \rightarrow Utt$;

In the formation of the *Imp* category, the verb phrase is turned into imperative mood and after that it is negated if negative polarity is specified.

4.9 Questions

There are various ways to form a question in GF. The two basic constructors are *QuestCl* and *QuestVP*. The first one creates yes/no questions and the second one creates wh-questions:

fun $QuestCl : Cl \rightarrow QCl$;
$QuestVP : IP \rightarrow VP \rightarrow QCl$;
$UseQCl : Tense \rightarrow Ant \rightarrow Pol \rightarrow QCl \rightarrow QS$;

Just like with the definitive sentences, the above constructors create clauses which do not have fixed tense and polarity. The *UseQCl* create interrogative sentences (QS) and fixes the tense and polarity.

In Bulgarian the questions are formed with the *li* clitic. It can appear either in the verb phrase or in the noun phrase and it indicates whether we are asking about the action or about the subject. In the GF grammar only the first option is used.

The reason is that questions like "Do you work?" usually ask about the action. The opposite question is "Is it you who work?". This is an idiomatic construction and has different translations in the different languages; anyway it still can be translated literally in Bulgarian and is still grammatical.

4.10 Future Work

There are many constructions that are not covered in the abstract syntax. Some are very language specific while others are more or less common. The right abstraction is not always obvious but it is also not necessary to have a single abstraction for all languages. Having a single language independent abstraction is definitely an advantage because then the application level grammar is just a parameterized module built on top of the abstract syntax. If there is not a single abstraction, then a language dependent abstract syntax can be used and the application level grammar will use different resource modules.

One immediate candidate for further extension is the tense system. A language specific module could be provided that implements all possible tenses, aspects and moods. However, the common abstract syntax could still work with the already existing language independent abstraction.

Other candidates are the pronouns. In Bulgarian the personal pronoun is quite often avoided because it is clear from the verb conjugation. For example:

(Ti) Govoriš li bălgarski? — Do you speak Bulgarian?

The pronoun *ti* is often skipped because it is clear from the verb conjugation that the subject is in third person singular. The generated sentence is still grammatically correct but the construction is used only for clarity or to stress the pronoun itself. In this respect it makes sense to have stressed and unstressed forms of the pronouns where the unstressed form will be just empty when the pronoun is in subject position. The same construction can be used also for the other pro-drop languages.

The possessive pronouns in Bulgarian have definite and indefinite forms: *moja-mojata* (my), *tvoja-tvojata* (your). The definite form is used to specify one particular object that belongs to the subject. In both cases in English my/your is used. This could be illustrated in Italian where both "la mia penna" and "una mia penna" (my pen) are possible. The difference is that in Bulgarian the definite article is a clitic and it attaches to the first word in the *NP* phrase which is the pronoun, so we have two different forms for the possessive pronouns.

4.11 Conclusion

The grammar does not cover all aspects of the language and it cannot be used to parse arbitrary text. This limits its usage in applications like information extraction and question answering where the input is often assumed to be free text. Despite this GF has been proven useful for dialog systems, text generation applications and for software localization where a well defined controlled language is used. The Bulgarian grammar can be successfully applied in the same areas. It is even more important for multilingual systems because with GF the translation can be done automatically or at least semi-automatically.

Bibliography

Krasimir Angelov. Type-theoretical bulgarian grammar. In *GoTAL '08: Proceedings of the 6th international conference on Advances in Natural Language Processing*, pages 52–64, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85286-5. doi: http://dx.doi.org/10.1007/978-3-540-85287-2_6.

Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *European Chapter of the Association for Computational Linguistics*, 2009.

Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, accepted, 2008.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, December 1997. ISBN 0521582741.

Joshi Aravind. Tree Adjoining Grammars: How much context-sensitivity is required to provide reasonable structural descriptions? *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, pages 206–250, 1985.

Tania Avgustinova. *Word Order and Clitics in Bulgarian*. PhD in Philosophy, der Universitat des Saarlandes, Saarbrucken, 1997.

J. Bresnan, editor. *The Mental Representation of Grammatical Relations*. MIT Press, 1982.

Björn Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 1–8, Prague, Czech Republic, 2007a. URL `http://www.aclweb.org/anthology/W/W07/W07-1801`.

Björn Bringert. Rapid Development of Dialogue Systems by Grammar Compilation. In Simon Keizer, Harry Bunt, and Tim Paek, editors, *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue*, pages 223–226, Antwerp, Belgium, September 2007b.

Björn Bringert. Speech Translation with Grammatical Framework. In *Coling 2008: Proceedings of the workshop on Speech Processing for Safety Critical Translation and Pervasive Applications*, pages 5–8, Manchester, UK, August 2008. Coling 2008 Organizing Committee. URL `http://www.cs.chalmers.se/~bringert/publ/gf-slt/gf-slt.pdf`.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, pages 53–60, Nancy, France, 2005. URL `http://www.cs.chalmers.se/~bringert/publ/mm-grammars-dialor/mm-grammars-dialor.pdf`.

Björn Bringert, Krasimir Angelov, and Aarne Ranta. Grammatical framework web service. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 9–12, Athens, Greece, April 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/E09-2003`.

Mark R. Brown. FastCGI: A High-Performance Gateway Interface. In Anton Eliëns, editor, *Programming the Web - a search for APIs, Fifth International World Wide Web Conference (WWW5), Paris, France*, May 1996. URL `http://www.cs.vu.nl/~eliens/WWW5/papers/FastCGI.html`.

Håkan Burden and Peter Ljunglöf. Parsing linear context-free rewriting systems. In *Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT)*, pages 11–17, October 2005. URL `citeseer.ist.psu.edu/burden05parsing.html`.

Håkan Burden and Peter Ljunglöf. Parsing Linear Context-Free Rewriting Systems. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 11–17, Vancouver, British Columbia, 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W05/W05-1502`.

D. A. Burke and K. Johannisson. Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In P. Blache and E. Stabler and J. Busquets and R. Moot, editor, *Logical Aspects of Computational*

Linguistics (LACL 2005), volume 3402 of *LNCS/LNAI*, pages 51–66. Springer, 2005.

David Burke and Luc Van Tichelen. Semantic Interpretation for Speech Recognition (SISR) Version 1.0. Working draft, W3C, November 2006. URL `http://www.w3.org/TR/2006/WD-semantic-interpretation-20061103`.

P. Callaghan and B. Medlock. Happy-GLR, 2004. URL `http://www.dur.ac.uk/p.c.callaghan/happy-glr/`.

Arjeh Cohen, Hans Cuypers, Karin Poels, Mark Spanbroek, and Rikko Verrijzer. WExEd - WebALT Exercise Editor for Multilingual Mathematical Exercises. In Mika Seppälä, Sebastian Xambo, and Olga Caprotti, editors, *WebALT 2006, First WebALT Conference and Exhibition, Eindhoven, The Netherlands*, pages 141–145, January 2006. URL `http://www.win.tue.nl/~amc/pub/wexed.pdf`.

Robin Cooper, Dick Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspars, Hans Kamp, David Milward, Manfred Pinkal, Massimo Poesio, Steve Pulman, Ted Briscoe, Holger Maier, and Karsten Konrad. *A Semantic Test Suite*. January 1996. URL `ftp://ftp.cogsci.ed.ac.uk/pub/FRACAS/del16.ps.gz`.

A. Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, 2002.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

Ali El Dada and Aarne Ranta. Implementing an Open Source Arabic Resource Grammar in GF. In *Perspectives on Arabic Linguistics XX*. John Benjamins Publishing Company, June 2007.

Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics, Toulouse,*

France, pages 252–259, Morristown, NJ, USA, July 2001. Association for Computational Linguistics. doi: http://dx.doi.org/10.3115/1073012.1073045. URL http://dx.doi.org/10.3115/1073012.1073045.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362007.362035.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. Technical Report 1.6, TALK Project, 2006. URL http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf.

Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

Masayuki Ishii, Kazuhisa Ohta, and Hiroaki Saito. An efficient parser generator for natural language. In *Proceedings of the 15th conference on Computational linguistics*, pages 417–420, Morristown, NJ, USA, 1994. Association for Computational Linguistics. doi: http://dx.doi.org/10.3115/991886.991959.

Kristofer Johannisson, Janna Khegai, Markus Forsberg, and Aarne Ranta. From Grammars to Gramlets. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *Proceedings of EACL'06*, 2006. URL http://citeseer.ist.psu.edu/jonson06generating.html.

Rebecca Jonson. Grammar-based context-specific statistical language modelling. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken*

Language Processing, pages 25–32, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W07/W07-1804.

A. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. In P. Sells, S. Shieber, and T. Wasow, editors, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, 1991.

Aravind Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

R. Kaplan and J. Maxwell. XLE Project Homepage, 2007. URL http://www2.parc.com/isl/groups/nltt/xle/.

Yuki Kato, Hiroyuki Seki, and Tadao Kasami. Stochastic multiple Context-Free Grammar for RNA pseudoknot modeling. In *Proceedings of the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms*, pages 57–64, Sidney, Australia, July 2006. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W06/W06-1508.

Janna Khegai. Grammatical Framework (GF) for MT in sublanguage domains. In *Proceedings of EAMT-2006, 11th Annual conference of the European Association for Machine Translation, Oslo, Norway*, pages 95–104, June 2006a. URL http://www.mt-archive.info/EAMT-2006-Khegai.pdf.

Janna Khegai. GF parallel resource grammars and Russian. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 475–482. Association for Computational Linguistics, 2006b. URL http://portal.acm.org/citation.cfm?id=1273135.

Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual Syntax Editing in GF. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2588 of *Lecture Notes in Computer Science*, pages 199–204. 2003. doi: http://dx.doi.org/10.1007/3-540-36456-0_48. URL http://dx.doi.org/10.1007/3-540-36456-0_48.

Borimir Krustev. *The Bulgarian Morphology in 187 type tables*. NI, Sofia, Bulgaria, 1984.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL `http://homepages.inf.ed.ac.uk/olemon/dude-final.pdf`.

Oliver Lemon, Xingkun Liu, Daniel Shapiro, and Carl Tollander. Hierarchical Reinforcement Learning of Dialogue Policies in a development environment for dialogue systems: REALL-DUDE. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 185–186, September 2006. URL `http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06\_lemon\_etal.pdf`.

Aristid Lindenmayer. Mathematical models for cellular interactions in development II. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, March 1968. doi: 10.1016/0022-5193(68)90080-5. URL `http://dx.doi.org/10.1016/0022-5193(68)90080-5`.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Department of Computer Science, Gothenburg University and Chalmers University of Technology, November 2004. URL `http://www.ling.gu.se/~peb/pubs/Ljunglof-2004a.pdf`.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Göteborg, Sweden, 2004. URL `http://www.ling.gu.se/~peb/pubs/p04-PhD-thesis.pdf`.

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

Moisés S. Meza Moreno. Implementation of a JavaScript Syntax Editor and Parser for Grammatical Framework. Master's thesis, Chalmers University of Technology, 2008.

Moisés S. Meza Moreno and Björn Bringert. Interactive Multilingual Web Applications with Grammatical Framework. In Bengt Nordström and Aarne Ranta,

editors, *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008, Gothenburg, Sweden*, volume 5221 of *LNAI*, pages 336–347, Heidelberg, August 2008. Springer.

Jens Michaelis and Marcus Kracht. Semilinearity as syntactic invariant. In *Logical Aspects of Computational Linguistics (LACL)*, pages 37–40, 1996.

G. Minnen, D. Gerdemann, and T. Gotz. Off-line optimization for earleystyle hpsg processing, 1995. URL `citeseer.ist.psu.edu/article/minnen95offline.html`.

R. Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.

R. Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

Ryuichi Nakanishi, Keita Takada, and Hiroyuki Seki. An Efficient Recognition Algorithm for Multiple Context-Free Languages. In *Fifth Meeting on Mathematics of Language*. The Association for Mathematics of Language, August 1997. URL `http://citeseer.ist.psu.edu/65591.html`.

Nuance Speech Recognition System 8.5: Grammar Developer's Guide. Nuance Communications, Inc., Menlo Park, CA, USA, December 2003.

Peter Pashov. *A Bulgarian Grammar*. Hermes, Plovdiv, Bulgaria, 1999.

C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

Carl Pollard. Higher-Order Categorial Grammar. In *Proceedings of Categorial Grammars 2004*, pages 340–361, June 2004. URL `http://www.ling.ohio-state.edu/~hana/hog/pollard2004-CG.pdf`.

Daniel Radzinski. Chinese number names, tree adjoining languages and mild context sensitivity. *Computational Linguistics*, 17:277–300, 1991.

A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

A. Ranta. GF Resource Grammar Library, 2008. URL `digitalgrammars.com/gf/lib/`. `digitalgrammars.com/gf/lib/`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004a. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796803004738. URL http://dx.doi.org/10.1017/S0956796803004738.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004b. URL http://portal.acm.org/citation.cfm?id=967507.

Aarne Ranta. Modular Grammar Engineering in GF. *Research on Language & Computation*, 5(2):133–158, June 2007. URL http://www.springerlink.com/content/r52766j31lg5u075/.

Aarne Ranta and Krasimir Angelov. Implementing controlled languages in GF. In *Pre-Proceedings of the Workshop on Controlled Natural Language (CNL 2009)*, volume Vol-448, Marettimo Island, Italy, June 2009. CEUR. URL http://ceur-ws.org/Vol-448/paper3.pdf.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: http://dx.doi.org/10.1023/B:JLLI.0000024736.34644.48. URL http://dx.doi.org/10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006. ISBN 1575865262.

Hiroyuki Seki and Yuki Kato. On the Generative Power of Multiple Context-Free Grammars and Macro Grammars. *IEICE-Transactions on Info and Systems*, E91-D(2):209–221, 2008. URL http://ietisy.oxfordjournals.org/cgi/reprint/E91-D/2/209.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October 1991. ISSN 0304-3975. URL http://portal.acm.org/citation.cfm?id=123648.

Hiroyuki Seki, Ryuichi Nakanishi, Yuichi Kaji, Sachiko Ando, and Tadao Kasami. Parallel Multiple Context-Free Grammars, Finite-State Translation Systems,

and Polynomial-Time Recognizable Subclasses of Lexical-Functional Grammars. In *31st Annual Meeting of the Association for Computational Linguistics*, pages 130–140. Ohio State University, Association for Computational Linguistics, June 1993. URL `http://acl.ldc.upenn.edu/P/P93/P93-1018.pdf`.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and Implementation of Deductive Parsing. *Journal of Logic Programming*, 24(1&2): 3–36, 1995. URL `http://citeseer.ist.psu.edu/11717.html`.

Edward Stabler. Varietes of crossing dependencies: Structure-dependence and mild context sensitivity. *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, 28:699–720, 2004.

Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *IJCAI*, pages 718–721, 1987.