Patti

Compiling Unification-Based Finite-State Automata
into Machine Instructions for a Superscalar
Pipelined RISC Processor

Sascha Brawer

March 1998

This thesis has been submitted to the Department of Computational Linguistics of the University of the Saarland, Saarbrücken/Germany, in partial fulfillment of the requirements for the degree of a "Diplom-Linguist (Computerlinguistik)" (Diploma in Computational Linguistics).

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

I hereby assure that I have written this thesis entirely on my own and that I have not used any other means than those indicated.

Bern, 4. März 1998

Primary Advisor: Prof. Dr. Hans Uszkoreit, University of the Saarland, Germany
Secondary Advisor: Prof. Dr. Manfred Pinkal, University of the Saarland, Germany

Typeset in ITC Zapf Humanist using MacWrite and ClarisDraw
Printed in Switzerland

An electronic version is available on the World-Wide Web:
http://www.coli.uni-sb.de/~brawer/patti/

E-mail address of the author: brawer@coli.uni-sb.de

Abstract

This thesis is about a method for speeding up natural-language analysis using a novel compilation technique. As its input, the compiler takes a unification-based linguistic formalism (non-deterministic finite-state automata, where transitions are labeled by attribute-value matrices according to a finite type logic with a simple-inheritance type hierarchy). As its output, the compiler generates machine instructions for the PowerPC chip, a pipelined RISC processor with superscalar instruction dispatch.

Because of its fine-grained knowledge about the task, the compiler is able to perform optimizations that would be very difficult to achieve using traditional techniques. Examples include heuristics for Static Branch Prediction, data cache control and scheduling the machine instructions to benefit from superscalarity, so that certain unifications are executed in parallel.

The system is evaluated by measuring the time it takes to extract noun groups in texts of some thousand words in length. On Apple PowerMacintosh machines, this task could be accomplished in fractions of a millisecond, theoretically corresponding to a speed of up to 21 million tokens per second. Hence, the generated code is so efficient that unification and pattern matching become neglectible factors in the overall performance of a natural-language system.

Due to the achieved speed, the presented techniques could form the foundation technology of new, real-time NLP applications.

Keywords

Natural Language Engineering · Finite-State Automata · Unification-Based Grammar Formalisms · Compilation · PowerPC

Table of Contents

1 Introduction

This first chapter discusses the motivation for the Patti project and outlines its history, before the larger context of its operation is explained. The chapter concludes with acknowledgements and presents an overview to the thesis.

1.1 Motivation

As widely known, the problem of dealing with human language is not an easy one. It is not surprising that quite elaborated algorithms have to be utilized, and naturally, these tend to be slow — in many cases so slow that many applications of that technology are not realized due to the restricted computing power. But the complexity of the task is only partly to be blamed for the long processing times: another factor is that not too many of the recent advances in computer technology had an impact on the way how natural language processing systems are built.

The work described by the present thesis aims to exploit the progresses of modern computer architecture for a highly efficient system to perform unification-based pattern matching. A compiler has been developed that takes a declarative linguistic formalism as its input, and generates high-performance Assembly code for the PowerPC chip. Due to its fine-grained knowledge about the task, the compiler is able to exploit very low-level methods to ensure extremely efficient processing. Examples include Static Branch Prediction, data cache control instructions, or mechanisms to benefit from the CPU's superscalarity so that certain unifications are executed in parallel.

The achieved speed is evaluated by measuring the time to extract simple noun groups from a number of different texts. Depending on the hardware (several Apple Power

Macintosh models), the matching speed ranges between 1.8 million up to 21 million tokens per second. It thus shows that the large effort invested into the compilation techniques is rewarded by an extremely high processing speed. It will be shown that these numbers are more of illustrative value, but in any case, the processing speed is so high that unification and pattern matching become entirely neglectible factors in the overall performance of a natural-language system.

To achieve this efficiency, certain drawbacks were necessary with regards to the expressiveness of the linguistic formalism: HPSG-style grammars probably could not be processed this fast. Nevertheless, the formalism offers a unification-based framework with a simple-inheritance type hierarchy and allows to use disjunction and negation in certain cases. However, when the question was to decide between computational efficiency and linguistic expressiveness, the former was preferred more often than not. The described system is hence clearly focused on Natural Language *Engineering,* not on Computational *Linguistics*.

1.2 History of the Patti Project

The basic ideas about compiling a very similar formalism were developed by the author during his time at the department for Computational Linguistics at the University of the Saarland, Germany (1993 – 1997). An performance-oriented unifier was implemented which worked very efficiently on bitvector-oriented data structures compiled out of a declarative specification of the feature logic. Some thoughts were pursued about viewing the grammar as a programming language that can be compiled into machine code; however, this latter kind of compilation was not realized back then.

From March to September 1997, the author had the opportunity to work at the Advanced Technology Group of Apple Computer, Inc. Among the tasks was to speed up the Pattern Matching module of a general-purpose Natural Language Analysis system. The author ported his previous compiler to Java, while extending it in numerous directions: the generation of instructions instead of data structures was added; the Unification Oracle (cf. section 5.3) was designed; and the context of the unification operation was changed from Chart Parsing to Finite-State Automata. The run-time algorithm for the traversal of finite-state transition networks was developed at Apple as well, and issues in instruction scheduling were discussed with Apple employees. According to the corresponding Intellectual Property Agreement, **certain parts of the present thesis might possibly describe intellectual property of Apple Computer, Inc.** The issue is currently being resolved; the author is trying to get ownership of all the algorithms described in this thesis in order to put them under a GNU General Public Licence.

From October to December 1997, several minor changes to the code generation took place. The instruction scheduling was improved substantially, and the cache control instructions were inserted.

1.3 Application of Patti

The Patti system is currently employed in a general-purpose “Linguistic Analysis Library,” developed for the (in the meantime dissolved) Advanced Technology Group of Apple Computer, Inc. When a text is passed to the library, it is sent through a Part-of-Speech Tagger. Currently, the English Constraint Grammar (ENGCG) developed by Lingsoft is used (cf. [Voutilainen et al., 1992], [Voutilainen, 1995]). For efficient representation, the text-based tagger output is converted into a more compact format.

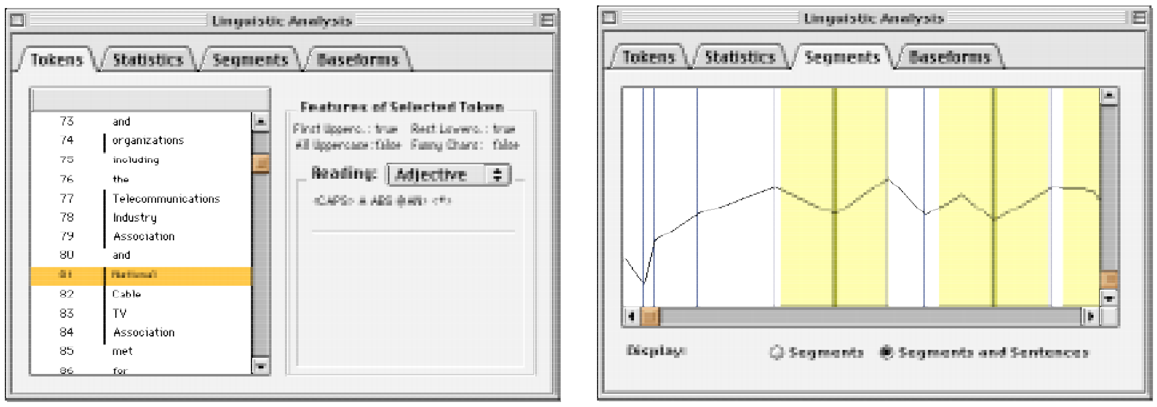
Patti is currently used to find basic linguistic units such as simple noun groups. Its task is thus comparable with Lingsoft’s NPtool (cf. [Voutilainen, 1993]) or parts of information extraction systems such as SMES (cf. [Neumann et al, 1997]) and FASTUS (cf. [Hobbs et al., 1997]).

The code generated by Patti is just one out of a number of modules in the Apple Linguistic Analysis Library which work on the tagger output. Additional modules provide the following functionality:

- the text is segmented into cohesive parts, using a modification of the algorithm described by [Hearst, 1994];
- structural elements such as titles or itemized lists are discovered using simple finite-state techniques not related to Patti;
- based on the shallow analysis, anaphora are resolved (cf. [Kennedy/Boguraev, 1996a], [Kennedy/Boguraev, 1996b], [Boguraev/Kennedy, 1997a]);
- technical terminology is identified (cf. [Boguraev/Kennedy, 1997b]);
- a quantitative salience measure is applied to find the most topical noun phrases in each discourse segment;
- finite-state automata (again not related to Patti) identify a “context” for the most topical noun phrases.

The outcome of the analysis is then presented to the user using a set of viewers designed to support rapid on-line skimming of document content. These viewers, described by [Boguraev et al., 1998], display in a number of different ways the most salient topics together with their context. A previous application of very similar basic technology was the construction of AppleGuide databases from index terms which were automatically extracted out of the Macintosh user documentation (cf. [Boguraev/Kennedy, 1997b]).

Fig. 1–1 depicts another viewer which graphically displays most of the internal data structures of the Apple Linguistic Analysis Library.



```
73    and
74  | organizations
75    including
76    the
77  | Telecommunications
78  | Industry
79  | Association
80    and
81  | National
82  | Cable
83  | TV
84  | Association
85    met
86    for
```

First Upperc.: true   Rest Lowerc.: true
All Uppercase: false   Funny Chars: false

Reading: Adjective

<CAPS> [illegible]

Display: ○ Segments ● Segments and Sentences

Fig. 1–1: *A specialized viewer, developed for debugging, displays internal data structures of the Apple Linguistic Analysis Library. Shown are views to the range of noun groups, detected with Patti, and to cohesiveness scores, used with the discourse segmentation algorithm. Similar visualizations exist for most other data structures.*

1.4 Acknowledgements

As required by the legal requirements for an academic thesis, the described work was clearly designed and implemented uniquely by the author. Nonetheless, it was certainly influenced by numerous discussions of the basic ideas with a large number of people.

First of all, the author would like to thank Branimir K. Boguraev who was responsible for natural-language-based knowledge mining in the Intelligent Systems Program at the former Advanced Technology Group of Apple Computer, Inc. Prof. Hans Uszkoreit of the University of the Saarland provided support already at an early stage of the project. Daniel M. Russel (now at Xerox PARC, Palo Alto) made it possible that the work could be completed by generously providing computing facilities.

Some general comments about the linguistic formalism were made by Karel Oliva (University of the Saarland), Susanne Riehemann (CSLI, Palo Alto) and Chris Kennedy (Northwestern University, Chicago). PowerPC instruction scheduling issues were discussed with Ivan Posva of Apple's "Yellow Box" department. A number of helpful comments came from Markus Becker, André Berthold and Christian Braun (University of the Saarland). Daniel Bobbert (University of the Saarland) and Michael Bosshard (Nyce AG, Bern) ran the benchmarks of chapter 6 on their respective machines.

The name for the Patti project was proposed by Sara Schödler, first because of its allegiance to Pattern Matching, and second because of Patti Smith, to whose music the author was listening while coding — in obscure ways, this might have been influential to the shape of the code.

1.5 Overview of the Thesis

The subsequent parts of the thesis are organized as follows:

- Chapter 2 sketches related work which is relevant as background for Patti. Modern applications of finite-state technology are discussed before turning to an extensive presentation of several methods to compile unification-based feature grammars. The chapter concludes with a description of bitvector-based unification algorithms.
- Chapter 3 describes the linguistic formalism which Patti takes as its input.
- Chapter 4 explains how the run-time execution works.
- Chapter 5, which constitutes the core of the thesis, describes in detail how the formalism of chapter 3 is transformed into the run-time executable of chapter 4.
- Chapter 6 presents the results of the empirical evaluation and compares them with other systems. The chapter concludes with answers to a number of common objections.
- Chapter 7 indicates possible directions for extending the Patti system in the future.

- Appendix A outlines some basic concepts of modern computer architecture such as RISC technology, Pipelining and Superscalar Instruction Dispatch. The techniques of Static Branch Prediction and Cache Control are introduced. Finally, a subset of the PowerPC instruction set is described, as far as needed to understand the listings in appendix C.
- Appendix B describes the full set of instructions in the intermediate processor-independent representation which the compiler generates. In some cases, its translation into actual PowerPC machine code is described.
- Appendix C shows an example input to Patti. The intermediate representation, as generated by the compiler, is itemized. Finally, the actual machine instructions, as produced by the PowerPC code generation module, are listed.
- Appendix D consists of literature references.

2 Related Work

The following chapter intends to sketch related work which was relevant as background for the present thesis. Patti is a compiler for non-deterministic finite-state automata where transitions are taken if their label (a typed feature structure) is unifiable with a feature structure associated with the current token. Therefore, it is based on previous work in two different fields: Finite-State Technology (as applied to linguistic tasks), and Unification-Based Grammar Formalisms.

First, it is elaborated why finite-state technology is currently very popular for Linguistic Engineering tasks. A short section refers to the formal properties of finite-state automata; many regard these as a reason to reject finite-state automata as linguistic formalism, in spite of their implementational advantages. For this reason, the most important approaches for *approximating* context-free grammars with finite-state automata are discussed.

The chapter then continues with a section describing a number of different methods to compile attribute-value matrices for efficient run-time processing. The main conceptual differences between these systems and Patti are discussed subsequently.

Finally, the chapter concludes with bitvector-based algorithms for type and feature unification, both relevant for the Patti system.

2.1 Syntactic Analysis with Finite-State Methods

For several years, a movement away from traditional phrase-structure approaches to the problem of syntactic analysis of natural language has been observed. Especially with the intensifying transfer of language technologies out of academic research into industrial projects, it became evident that the preferred methods of the 1980 years could not suffice for the higher requirements of real-world applications. As a consequence of this development, finite-state methods had a revival as a very robust technique, suitable for highly efficient algorithms.

Of course, the present thesis neither is able to convey the mathematical theory behind finite-state technologies, nor it is possible to sketch the multitude of its numerous applications in Linguistics, considering the sparse space restrictions. For the former, the reader might want to refer to [Lewis/Papadimitriou, 1981], while [Kornai, 1997] with its references could serve as introduction to the latter. Nonetheless, the subsequent section intends to outline the rationale for which finite-state methods were of great importance in Language Engineering, starting from the origins of the field up to the most recent developments.

Notably, the idea of syntactic analysis with finite-state technologies is not as novel as it might seem when considering the abundance of recent publications on this topic. The approach has indeed been proposed several times in the past; presumably the first system had been developed towards the end of the 1950 years. An outline of its architecture (which comes surprisingly close to that of current frameworks) is provided by [Joshi/Hopely, 1997].

2.1.1 Efficiency

With almost all industrial applications of natural-language technologies, a central concern is *efficiency,* both with regards to execution speed *(time efficiency)* and memory consumption *(space efficiency).*

In both these areas, the traditional parsing algorithms (e.g. Earley's Algorithm, Cocker-Kassami-Young parsing, etc.)[1] do not seem to be entirely capable to fulfill the requirements imposed by industrial-quality products. [Abney, 1997] presents a highly infor-

[1] A collection of the most common Parsing algorithms (though in German) is given by [Naumann/Langer, 1994].

mative collection of the execution speed of a number of different existing systems. For Chart Parsing, the preferred means of many implementations, he reports a processing speed of less than one word per second.

As experience shows, efficiency can be improved to a certain extent by dexterous implementation, fine-tuning the utilized algorithms[2] and modifications to the grammars.[3] In spite of these efforts, it seems unlikely that traditional parsing could be fast enough to analyze documents in (at least approximately) imperceptible time. In the case of a large number of interactive natural-language applications, for instance Grammar Checking or Speech User Interfaces, noticeable delays though void the usability and acceptance of the technology, how sophisticated it might be otherwise.

2.1.2 Robustness

Besides the eminent engineering problems with time and space efficiency, there thoroughly exist additional reasons for turning away from those methods for syntactic analysis which exhaustively try to detect the structure of the input sentences. The most important rationale is *robustness*. For example, [Abney, 1997], p. 340 argues as follows in favor of (cascaded) finite-state automata as opposed to traditional exhaustive parsing algorithms:

> *Unlike traditional parsers, there is no global optimization. This contributes not only to speed, but also to robustness. Namely, a common problem with traditional parsers is that correct low level phrases are often rejected because they do not fit into a global parse, due to the unavoidable incompleteness of the grammar. This type of fragility is avoided when low level phrases are judged on their own merits.*

Due to these reasons, traditional approaches to parsing lack the property of robustness, while *partial* parsing methods seem to be a very viable altemative for a large number of applications.

Recently, there were large achievements in the field of Message Understanding, or more exactly: populating a database by extracting relevant information from natural-language texts. Leading systems such as FASTUS (cf. [Hobbs et al., 1997]) and SMES

[2] [Russi, 1990], p. 82, concludes his experiments as follows: "by carefully tuning a parsing strategy, efficiency can be significantly increased. Memory consumption can be reduced by up to a factor of four and CPU time by up to a factor of three."

[3] For example, [Pulman, 1993] and [Brawer, 1995] present a number of efficiency-oriented coding techniques for unification-based phrase-structure grammars.

(cf. [Neumann et al., 1997]) are able to extract material information from texts of a significant level of complexity without the demand for a full syntactic analysis of the sentences. Certainly, a major factor for the current success of these *Information Extraction Systems* is their robustness: The underlying technology copes much better than traditional, parsing-based systems with unrestricted,[4] real-world texts.

2.1.3 Formal Language Issues

Even after decades of controversy about the issue which class of formalisms are adequate to describe and formalize the syntax of natural languages, the final answer has yet to be found.[5]

Since the beginnings of Mathematical Linguistics, it is common knowledge that the formal expressiveness of finite-state automata (which only are able to cover regular languages) is certainly not satisfactory for natural languages. Nevertheless, several authors have pointed out that the human language processing is subject to certain restrictions, as they would follow from a parser for regular languages.[6]

From an engineering perspective, it makes sense to process the output of a finite-state transducer with other modules based on different technologies. These subsequent units do not have to be related to the actual grammar formalism — for example, processes as complex as anaphora resolution work on the results of the Pattern Matcher described by the present thesis (cf. section 1.3). If these modules are able to filter out analyses of the finite-state technologies, they can affect the mathematical properties of the overall system.[7]

[4] Most systems however require certain, though small, adaptations to the respective domain.

[5] [Savitch et al., 1987] presents a selection of the most important publications on this subject.

[6] See the references for section 3.2.1 of [Gazdar/Pullum, 1985]. Another example constitutes [Abney, 1997], p. 343ff., with the hypothesis that the longest-match heuristics, commonly used with Pattern Matching algorithms, corresponds to a linguistic property of English. Abney claims to be able to model at least certain garden-path phenomena more accurately than it would be possible with a phrase-structure grammar.

[7] An example for such a component would be a module which utilizes context-free rules to filter out the output of a finite-state transducer. Since the intersection of a context-free language with a regular language is context-free (a proof is given by [Lewis/Papadimitriou, 1981], p. 121), the language recognized by this system would be context-free.

2.1.4 Finite-State Approximations of Context-Free Grammars

It already has been mentioned that the Patti system is based on finite-state techniques. In this context, it might be of interest that several different algorithms have been developed to construct finite-state approximations of context-free grammars (and of context-free-equivalent augmented phrase-structure formalisms), despite the fact that none of these methods have been incorporated yet into the system described by this thesis.

Without doubt, the outcome of such a "compilation" algorithm can not be accurately equivalent to its input formalism: It is a well established fact that the context-free languages are a proper superset of those languages finite-state automata can possibly recognize.[8] Nevertheless, it is very well possible to construct a finite-state automaton whose recognized language is an *approximation* of the original context-free language. Below, some relevant transformation algorithms will be presented. However, it is difficult to state a useful quantitative measure of approximation quality:[9] On the one hand, the language recognized by the approximation should not be substantially smaller in comparison to the input grammar. On the other hand, a heavily overgenerating automaton is of not too much interest either.[10]

One proposal (cf. [Black, 1989]) relies on arbitrary depth cutoffs in rule application. The application of context-free rules is followed up to a certain, pre-specified depth of recursion; any further rule invocation is omitted.

[Pereira/Wright, 1996] object to this depth-cutting approach; they argue that with the method described by [Black, 1989], the language of the resulting automaton neither is a subset nor a superset of the language of the input phrase-structure grammar. In contrast, they present an algorithm which produces an exact finite-state automaton for many grammars generating regular languages and does not reject input exceeding a fixed depth of embedding. Basically, their method operates by constructing a nondeterministic shift-reduce pushdown recognizer for the input grammar. This recog-

[8] The proof follows from the pumping theorem for regular languages. See any textbook on the theory of computation, e.g. [Lewis/Papadimitriou, 1981], p. 75.

[9] [Pereira/Wright, 1996], p. 20, discuss a number of possible criteria to quantify approximation quality.

[10] A trivial example for the latter case is an automaton for Σ^*, Σ being the alphabet of the input grammar. The language of this automaton is a superset of the language of any context-free grammar, but nevertheless, it is entirely useless as an approximation.

nizer is subsequently transformed into a FSA by eliminating the stack and turning reduce moves into ε-transitions. Since the moves of a typical pushdown recognizer depend heavily on the contents of its stack, ignoring the stack entirely would lead to massive overgeneration of the FSA in many cases. This is partly avoided by incorporating information about the stack contents into the states of the resulting automaton. To achieve this goal, the (in general infinite) set of possible stacks at a given state is partitioned according to an equivalence relation. In effect, this collapses recursion by ignoring repetitive parts of the stack. Thus, the resulting automaton will not reject any input that is acceptable by the given context-free grammar, whatever the depth of embedding is. However, the FSA will in some cases accept strings rejected by the CFG, and the information about the depth of embedding is lost.

[Rood, 1996] presents an algorithm along similar lines. As with [Pereira/Wright, 1996], her finite-state approximation does not reject any input recognized by the CFG, but in addition, it is exact to an arbitrary depth of recursion.

2.1.5 Unification-Based Transition Networks

Among the first approaches to natural-language analysis were adaptations of finite-state techniques to cope with the recursive nature of certain linguistic constructions. *Recursive Transition Networks* (RTNs) were developed as an extension to non-deterministic finite-state automata. As an alternative to the consumption of a symbol of the input alphabet, RTNs are able to invoke other transition networks or even themselves. However, the RTN formalism got obsolete, since its expressive power was shown to be exactly the same as that of context-free grammars, for which superior parsing algorithms exist.

Nevertheless, there exists a recent thesis based on RTNs: Thomas Russi's grammar formalism, *Unification-Based Transition Networks (UTNs),* is an extension of the RTN concept in two respects (cf. [Russi, 1990], p. 36):

- terminal and non-terminal symbols are no longer atomic symbols, but (untyped) feature structures;
- in addition to the linear precedence and immediate dominance relations encoded in the topology of the networks, additional constraints between terminals and constituents can be specified using unification equations.

The relation of the UTN formalism with Patti is the combination of finite-state methods with unification of feature structures, though Patti's automata are not recursive and do not allow yet to enforce agreement between different structures.

2.2 Compilation Techniques for Feature-Logic Grammars

It is a very old and common technique in Computer Science to transform a high-level formalism (which is easy for humans to understand) into a low-level one (which is efficient for computers to process). For example, the task of a Pascal compiler is to map concepts such as loops into suitable sequences of machine instructions, which then can be executed directly by the microprocessor.

The idea of compilation has been adapted to the problems of Natural Language Processing several times. Patti certainly goes farther with compilation than most other systems — to the author's knowledge, there exist no other NLP-related compilers which generate machine instructions for a real CPU (or equivalent Assembly-language code). Nevertheless, other compilation algorithms have influenced the design of Patti. The subsequent sections thus present three representative methods to compile feature-logic grammars. These compilers exhibit an increasing degree of deviance from standard Prolog unification. A concluding section then shows that in many typical natural-language systems, hidden interpretation processes reduce performance, despite the application of advanced compilation techniques.

2.2.1 ProFIT

An example for a straightforward compilation technique for feature-logic grammars is the system developed by Gregor Erbach (cf. [Erbach, 1994]). *Prolog with Features, Inheritance and Templates (ProFIT)* is a compiler which is able to translate typed feature structures into ordinary Prolog terms (and vice versa, for debugging). To unify two feature structures at run-time, standard Prolog unification is utilized.

The mapping works along the following lines:

- Since the feature structures are totally well-typed and the usual restrictions on appropriateness (such as the Feature Introduction Condition, cf. [Carpenter, 1992], p. 85ff.) hold, feature names can be omitted by assigning a fixed position in the resulting Prolog terms.
- Feature structures of consistent types are compiled to Prolog terms of the same functor and the same arity.

- The first argument of the Prolog term is a variable whose only purpose is to express whether two terms are coreferent or whether they just happen to have the same type and the same values for all features.
- The second argument serves to encode the actual type of the structure and to represent those features which are not encoded as arguments of the main term, because they were introduced by some sub-type of the type encoded as functor.
- Templates are expanded at compile-time by partial evaluation.
- Clauses containing disjunctive terms are compiled to several clauses, one for each consistent combination of disjuncts. However, disjunction of a finite set of atoms is encoded in a single Prolog term, using a technique originally developed by Colmerauer (cf. [Pulman, 1993] or the references in Erbach's article).

ProFIT constitutes an example for a system where feature structures are compiled into another formalism which is more efficient to process. Obviously, the ProFIT system is very closely tied to Prolog as run-time engine. Therefore, the compilation algorithm could not be adapted directly for Patti.

2.2.2 ALE

The typical approach to processing attribute-value logic grammars is to encode the feature structures as Prolog terms, for example using a system such as ProFIT. There is no principal difference between those feature structures of the current sentence and those of the grammar; both are *data structures* which are processed by a unifier whose application is controlled by some parsing strategy.

[Carpenter/Penn, 1995] however notice that the feature structures in the grammar are *static:* they are never modified while input is being processed at run-time. Therefore, Bob Carpenter and Gerald Penn propose a compilation process to transform the grammars into a few efficient low-level instructions for the basic feature structure operations. Correspondingly, their system (called ALE for "Attribute-Logic Engine") compiles grammar descriptions into Prolog code, rather than into a Prolog representation of feature structures. At run-time, ALE then executes the code that was compiled for the grammar rules.

The compiler performs certain optimizations by precalculating parts of the unification result already at compile-time. Another efficiency gain is due to a representation of feature structures that is very closely adapted to the way in which the Warren Abstract Machine (the core engine of virtually all Prolog implementations) operates. For instance, disjunction of feature values is compiled into Prolog disjunctions — when a term is not unifiable, the Prolog engine performs backtracking and tries to unify with the next disjunct.

Although Carpenter and Penn claim that their compilation method is basically independent of Prolog, many of their optimizations are only effective in the context of the Warren Abstract Machine. Only the notion of compiling the static parts of a grammar into code was thus adapted for Patti; the other ideas were regarded as too closely linked with Prolog.

2.2.3 AMALIA

Similar to Bob Carpenter and Gerald Penn, Shuly Wintner views a natural-language grammar as *program code* which can be compiled into a lower-level formalism. However, Shuly Wintner does not compile the input formalism (a subset of ALE's specification language) to Prolog programs, but into instructions for his Abstract Machine for Linguistic Applications (AMALIA). Abstract Machines are a very common technique in Computer Science: the vintage P-Code Interpreters were Abstract Machines for Pascal, and the contemporary Java Virtual Machines are merely another instance of the same concept. AMALIA is heavily influenced by the Warren Abstract Machine (WAM) for Prolog.

The main benefit of Abstract Machines is their portability: Only the (ideally very simple) interpreter has to be modified, since the Abstract Machine Code is entirely platform-independent. However, as will be discussed more in detail in section 2.3.4, the disadvantage of using an interpreter for Abstract Machine Code is a — potentially very large — loss in execution speed.[11]

[Wintner/Francez, 1994] and [Wintner/Francez, 1995] provide a detailed technical description of AMALIA. The remainder of the current section gives a very terse overview thereof.

The basic formalism of AMALIA is a subset of ALE. In terms of [Carpenter, 1992], all feature structures have to be totally well-typed. This allows to omit feature names from the run-time representation by using an efficient positional encoding similar to

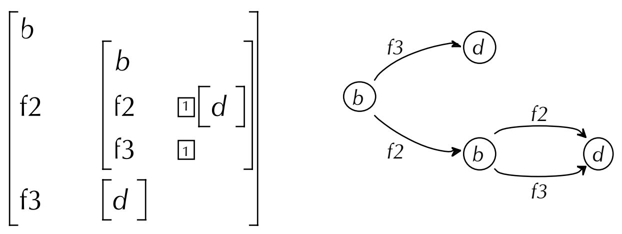
[11] For instance, compiled Java Bytecode is executed easily ten times faster than with an interpreting Virtual Machine. See http://www.pendragon-software.com/pendragon/cm3/ for the Java CaffeineMark benchmarks.

terms of first-order predicate logic. The set of types is ordered by the subsumption relation. It is required that the type inheritance hierarchy be bounded complete: Every set of consistent types must have a unique unifier (other than the contradictory type). In addition, it is also required that the appropriateness specifications of the features and the types be such that every feature is introduced by some least type, that appropriateness be monotone, and that the appropriateness specification does not contain loops.

The machine's engine is designed for unifying two different varieties of typed feature structures: a *query* and a *program*. Both are compiled to AMALIA instructions:

- Those feature structures that are part of the current sentence which is about to be parsed are called *query* (in allegiance to the Prolog WAM). Each query has to be compiled at run-time into instructions for AMALIA before its execution. Processing these instructions builds a graph representation of the query in a special section of the machine's memory, called *heap*.[12] Fig. 2–1 below illustrates the run-time representation of a feature structure in the heap.
- Those feature structures that are part of a grammar rule are called *program*. Because these structures do not change at run-time, they can be compiled in advance into instructions that try to unify the source AVM with a feature structure on the heap. If the unification succeeds, the heap is modified accordingly.

[12] It appears that it would have been simpler if the query was immediately processed into data structures on the heap, instead of compiling the query into machine instructions which in turn build data structures when executed by the Abstract Machine. The side-effect of processing a query, viz. setting registers, is an important notion with the WAM, but of no use for AMALIA (cf. [Wintner/Francez, 1995], section 3.7, p. 8). Unfortunately, Shuly Wintner never points out in his work why he chose the former, seemingly much more complicated method which might constitute a major overhead. The taken path is comparable to an approach where upon the run-time execution of a Pascal *readln* instruction, a compiler is invoked that compiles each character of the entered string into a piece of P-Code that stores the character in a specific memory cell when executed by the interpreter, instead of simply copying the string.



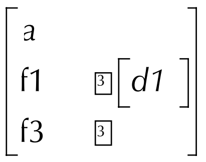
Address	Tag	Contents
1	STR	b
2	REF	4
3	REF	8
4	STR	b
5	REF	7
6	REF	7
7	STR	d
8	STR	d

Fig. 2–1: *In the AMALIA system, an attribute-value matrix (depicted to the left) is seen as a directed, labeled graph (middle) where nodes correspond to types and edges are labeled with attribute names. The heap used for run-time representation (right) consists of tagged cells. STR cells are used for the nodes, REF cells for the edges of the graph.*

The abstract machine code for a program AVM basically consists of the subsequent three different types of instructions:

- `get_structure` performs type unification of the (compiled) program AVM with a feature structure on the heap. In case of an uninstantiated variable, a new feature structure is built; otherwise, precompiled code is called, depending on the type of the heap AVM.
- `unify_variable` loads the address of the next heap cell into a register. This instruction is emitted when a node is accessed the first time.
- `unify_value` unifies the next heap cell with a heap cell whose address a register contains. This instruction is emitted for additional accesses to the same node.

Fig. 2–2 below depicts the code the compiler emits for a simple feature structure which is part of the program (i.e. a grammar rule).



```
get_structure    a/2, X1     % X1 = a(
unify_variable   X2          %        X2,
unify_value      X2          %            X2)
get_structure    d1/0, X2    % X2 = d1
```

Fig. 2–2: *For the attribute-value matrix (supposed to be part of a grammar rule) to the left, the compiler generates the Abstract Machine instructions depicted to the right. During execution of these instructions, a pointer to the current heap cell is incremented, hence the heap address does not need to be part of the machine instructions.*

Certain control instructions allow to process entire phrase structure rules, not only single attribute-value matrices. Because Patti utilizes finite-state techniques, the parsing-related parts of AMALIA though are not relevant in the context of the present thesis.

A recently introduced, unique feature of AMALIA is its suitability for both parsing and generation (cf. [Wintner et al., 1997]): depending on the task, the compiler is able to generate two different object code files for the same grammar. The output is then processed by the very same interpreter for the abstract machine code.

2.2.4 Criticism of the Above Systems

The above described systems have in common that they transform a high-level grammar formalism with typed feature structures into another representation, which is more low-level and hence easier and faster to process. In contrast to modern compilers for programming languages such as Pascal or C, however, none of these linguistic compilers generate output which is understood directly by the hardware of any computer. Instead, an intermediate interpreter has to ensure that the compilation result be processable by the actual computing machinery. Unfortunately, it is common practice to develop even those interpreters in an *interpreted* programming language. Not only does this hold for prototype research systems, but also for architectures intended for industrial processing of natural language.

This is certainly one reason why the processing efficiency of many "high-speed" natural-language systems is barely optimal. It is surprising how many hidden interpretation steps still are in place, despite the application of a number of highly sophisticated compilation algorithms. Fig. 2–3 might serve as an illustration thereof: even in a NLP system specifically designed for high-speed processing (cf. [Samuelsson, 1994]), there still remain several interpretation steps between the original formalism (in this case: phrase-structure grammars acquired with explanation-based learning methods) and the underlying machinery (e.g. an Intel Pentium or a Motorola 680x0 microprocessor).

The reader might object that certain assumptions are slightly malicious, since the picture presents a worst-case scenario. Indeed, modern RISC and VLIW instruction sets[13] were designed to eliminate microcode interpreters. Moreover, nowadays there exist

[13] See Appendix A for a brief discussion of Reduced Instruction Set Computing (RISC). On superscalar RISC implementations, parallelism of several instructions is automatically achieved. On "Very Large Instruction Word" (VLIW) architectures, in contrast, the compiler (or Assembly programmer) has to specify explicitly which machine instructions can safely be executed in parallel.

Prolog-based systems which do not *interpret* WAM code, but directly *compile* Prolog into actual machine instructions for the respective microprocessor.

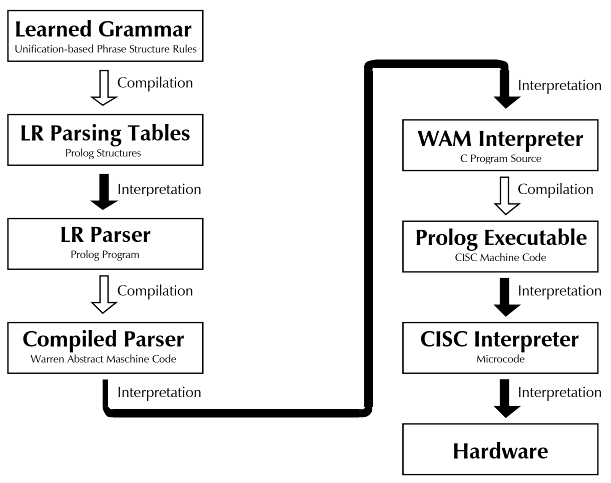


Fig. 2–3: *Interpretation and compilation steps in a typical system designed for fast syntactic analysis of natural language (cf. [Samuelsson, 1994]), running Prolog on a microcode-based CISC processor. Even after the application of advanced compilation techniques (white arrows), there remain several interpreters (black arrows) between the original formalism and the executing hardware.*

The answer to this objection is twofold. First, the depicted worst-case scenario is not unrealistic at all: Numerous Prolog (and Lisp) compilers work with Abstract Machines instead of actual machine code, and it is probably still the case that microcoded CPUs form the majority.

Second, and more important, even if those interpreters were replaced by compilers, the efficiency could still be improved to a large extent. The motivation for this audacious-sounding hypothesis is the following: "Compilation" does *not* necessarily mean "entire elimination of overhead." This is especially true if one compiler works on the output of another one. Certain optimizations are only possible by using (both explicitly or implicitly) *knowledge* about the given task. The more general this task is, the less optimizations will usually be applicable. But ordinarily, only the first, top-level compiler is in possess of the full knowledge about the problem. Not only do subsequent compilers operate on a lower level, but they also have to cope with more general problems. For example, even a *real* Prolog compiler (which emits "concrete" machine code) can not apply optimizations which might be possible with the knowledge that the program is a driver for finite-state cascades, Earley parsing or some other specialized NLP algorithm — its optimizations have to be based on general Prolog heuristics. Furthermore, not even the most specific compiler is able to perform all possible optimizations if its output formalism does not allow to express them. For instance, a high-level compiler might be able to perform an educated guess whether or not a certain conditional branch in the program flow is likely to be taken. There is however no means to express such knowledge with Prolog, Lisp or even C as output formalism — only if the high-level compiler directly emits Assembly code, the system can benefit of this optimization.

As will become clear towards the end of the present thesis, the main goal in designing Patti was to build a compiler which transforms a fairly high-level input formalism into a very low-level output formalism, thus being able to exploit sophisticated heuristics in order to speed up the system using techniques offered by modern Computer Architecture.

2.3 Unification with Bit-Vectors

Although the notions of *Type Unification* and *Feature Unification* are part of the foundation of modern Linguistics, it is not generally well-known that highly efficient methods, based on bit-vector operations, have been developed for implementing both tasks. The subsequent sections outline the basic ideas of these algorithms, while assuming familiarity with the linguistic concepts.

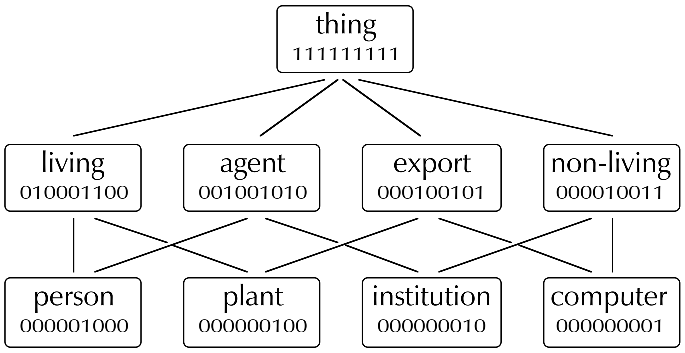
2.3.1 Type Unification with Bit-Vectors

Hassan Aït-Kaci and others have formulated an algorithm to determine efficiently the greatest lower bound in a lattice [Aït-Kaci, 1989]. A very concise description thereof is given by [Pulman, 1993].

Although Patti uses a different encoding for its types (hence its type unification algorithm is very dissimilar), one of the optimizations (the "Unification Oracle", cf. section 5.3) is based on the method developed by Aït-Kaci et al. The subsequent section gives an introduction to the basic idea. Figure 2–4 provides an illustration of the algorithm for an example lattice.

Aït-Kaci et al. encode the individual nodes in the lattice as bit-vectors. To each node, a corresponding vector position is assigned arbitrarily (cf. fig. 2–4, right part). Then, they calculate the reflexive transitive closure of the immediate dominance relation to determine the run-time encoding for each node. In the vector for a node *n,* those bits are set that correspond to nodes which are dominated by *n*.

The encoding of the Greatest Lower Bound of two nodes is found by performing an AND operation on their vectors. If there exists no Greatest Lower Bound, i.e. if there is no node dominated by both nodes in question, the result is an all-zero bit-vector.



Node	Bit
thing	0
living	1
agent	2
export	3
non-living	4
person	5
plant	6
institution	7
computer	8

Fig. 2–4: *A method for unification in multiple-inheritance hierarchies [Aït-Kaci et al., 1989]. The nodes are encoded as bit-vectors with as many bits as there are nodes in the lattice. For each node, those bits that correspond to the subsumed nodes are set. Unification is performed as bit-AND on the vectors. Unification failure (i.e. if there exists no common lower bound) is indicated by an all-zero result.*

2.3.2 Feature Unification with Bit-Vectors

A large part of the efficiency of the described system is due to the fast execution of the unification operation. Instead of performing actual feature unification, the unification is emulated by logical operations on a bit-vector representation. To the knowledge of the author, this standard technique has first been proposed by Nakazawa et al. (cf. [Nakazawa et al., 1988] and [Nakazawa/Neher, 1987]).

Nakazawa et al. describe a framework which incorporates disjunctive and negative feature values and which provides computationally efficient data representations and manipulations. The data representation uses vectors of feature descriptions and the logical operation of Boolean AND to unify them. To each possible value of a feature, one bit is assigned in the vector for the attribute-value matrix which contains the feature. Disjunction of values is represented by setting those bits that correspond to the individual disjuncts. Negation is represented by setting all bits that correspond to the possible values of a feature but that of the negated values.

In their framework, the linguist can enforce the *absence of a feature* from a syntactic category. In addition to a bit for each possible feature value, they need hence another bit for every feature to indicate if the feature itself is absent or present in the AVM.

Because their formalism is not typed, the run-time representations do not contain a type identifier, and the bit-vectors for all top-level AVMs have the same length. If a feature is complex, i.e. its value is an entire attribute-value matrix instead of merely (possibly disjoined and negated) atomic values, an additional bit-vector is required as run-time representation for the value. The paths in a feature structure can thus be of arbitrary depth.

Fig. 2–5 gives an example feature structure and its run-time representation.

$$\begin{bmatrix} \text{V} & + \\ \text{N} & - \\ \text{COMP} & \text{that} \\ \text{VFORM} & \text{BSE} \end{bmatrix}$$

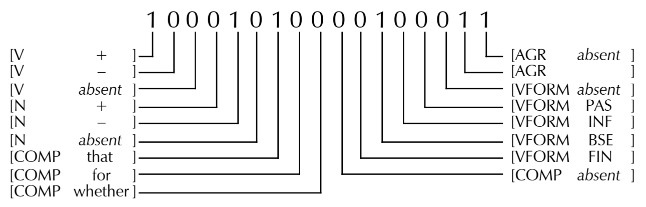


Fig. 2–5: *In the framework of Nakazawa et al., an attribute-value matrix (depicted to the left) is encoded as a bit-vector (depicted to the right). A vector position is assigned to each possible feature value. Additional bits mark for each feature if it is present or absent; this is needed because the linguist can explicitly enforce the absence of a feature. The illustration is taken from [Nakazawa et al., 1988], p. 468.*

3 Formalism

This chapter introduces the formalism which serves as input to the Patti compiler. Both a special version of non-deterministic finite-state automata and the utilized feature logic are defined formally. The chapter then continues with a section about how linguists currently have to specify their knowledge. Finally, the current direct-manipulative Graphical User Interface is described.

3.1 Formal Definition

This section defines the input formalism for the Patti compiler and describes deviations from the traditional definitions, as commonly found in the literature.

3.1.1 Non-deterministic Finite-State Automata with Unification

Basically, the input formalism is a set of non-deterministic finite-state automata enhanced with unification. Such an automaton is a quintuple $M = \langle K, \Sigma, \Delta, s, F \rangle$, where

- K is a finite set of states,
- Σ is the set of all typed feature structures that are well-formed according to the feature logic specified in the subsequent sections,
- Δ, the transition relation, is a finite subset of $K \times \Sigma \times K$,
- $s \in K$ is the initial state,
- $F \subseteq K$ is the set of final states.

Please notice the two deviations from the usual definition of this species of mathematical machines:[14]

- the input alphabet does not consist of atomic symbols, but of typed feature structures,
- each transition is labeled with a *single* feature structure, because $\Delta \subseteq K \times \Sigma \times K$. In contrast, the standard definition of non-deterministic finite-state automata allows for arbitrary strings over the input alphabet, including the empty word ε, since $\Delta \subseteq K \times \Sigma^* \times K$.

As usual, a *configuration* of M is defined as an element of $K \times \Sigma^*$. However, the relation $\vdash_M$ between configurations *("yields in one step")* is non-standard:[15] $\langle q, w\rangle \vdash_M \langle q', w'\rangle$ if and only if there are typed feature structures $u, u' \in \Sigma$ such that

- $w = uw'$,
- $\langle q, u', q'\rangle \in \Delta$, and
- u is unifiable with u', according to the usual definition of unification of feature structures (see for example [Carpenter, 1992], definition 3.11).

The other definitions related to non-deterministic finite-state automata remain unchanged compared to the common practice: $\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$, and a string $w \in \Sigma^*$ is accepted by M if and only if there is a state $q \in F$ such that $\langle s, w\rangle \vdash_M^* \langle q, \varepsilon\rangle$. Finally, $L(M)$, the language accepted by M, is the set of all strings over Σ^* accepted by M.

3.1.2 Type Hierarchy

[Carpenter, 1992] defines the mathematical foundations of most current feature-logic grammars. According to his formalization, a *type inheritance hierarchy* is a finite bounded complete partial order $\langle Type, \leq\rangle$, with *Type* being a finite set of types and $\leq$

[14] According to the standard definition (cf. [Lewis/Papadimitriou, 1981], p. 57, or any other textbook on the theory of computation), a non-deterministic finite-state automaton is a quintuple $\langle K, \Sigma, \Delta, s, F\rangle$, where K is a finite set of states, Σ is an alphabet, $s \in K$ is the initial state, $F \subseteq K$ is the set of final states, and Δ, the transition relation, is a finite subset of $K \times \Sigma^* \times K$.

[15] According to the standard definition, the relation $\vdash_M$ is defined as follows: $\langle q, w\rangle \vdash_M \langle q', w'\rangle$ if and only if there is a $u \in \Sigma^*$ such that $w = uw'$ and $\langle q, u, q'\rangle \in \Delta$.

being the reflexive transitive closure of a direct subsumption relation $<$ over *Type* $\times$ *Type*.[16] If $\sigma \leq \tau$, σ is called to *subsume,* or to *be more general than,* or to be *supertype* of, τ. Conversely, τ is called a *subtype* of σ. The usual restrictions on inheritance hierarchies hold, as formalized by [Carpenter, 1992], chapter 2. For instance, type hierarchies must not be cyclic.

In addition to those common constraints, the Patti type hierarchies are restricted to simple trees, where every type is directly subsumed by exactly one type. The only exception is the single most general type *Top,*[17] which is not directly subsumed by any type at all.

The result of *unifying* two types σ and τ ($\sigma \sqcup \tau$) is obtained by performing the greatest lower bound operation. In the case of a tree-structured inheritance hierarchy, the only way in which $\sigma \sqcup \tau$ can be defined is if $\sigma \leq \tau$ or $\tau \leq \sigma$, in which case the result is the more specific of the two types.[18]

In the Patti input formalism, two designated types, called *Atom* and *Matrix*, have specialized meanings which facilitate the compilation process. These two are the only types directly subsumed by *Top;* no user-defined type is allowed to be a direct subtype of *Top*.

3.1.3 Feature Structures

A *feature structure* over *Type* and a finite set of feature names *Feat* is a labeled, rooted, directed graph $F = \langle Q, q_0, \theta, \delta \rangle$ where Q is a finite set of nodes, $q_0 \in Q$ is a designated root node, $\theta \in Q \rightarrow Type$ is a total node typing function and $\delta \in (Feat \times Q) \rightarrow Q$ a partial feature value function.[19]

A *path* is a sequence of features; let $Path = Feat^*$ be the collection of paths. δ is ex-

[16] Carpenter calls the direct subsumption relation "ISA."

[17] Carpenter though calls this type *Bottom*. Many find this terminology counterintuitive, since they prefer to see the most general type on the top of the type lattice. This thesis adheres to the latter view and deviates from Carpenter, *Top* thus being the most general, *Bottom* ($\perp$) the inconsistent type.

[18] Cf. [Carpenter, 1992], p. 13.

[19] The symbols F and q have already been utilized in section 3.1.1 to define finite-state automata. Despite of this ambiguity, it was decided to adhere to the commonly used notation for both feature structures and automata. Many readers are assumed to be familiar with the standard symbols, which is why deviating from the usual terminology would introduce more confusion than it would resolve.

tended to paths so that $\delta(\pi, q)$ is the node that is reached by following the features in the path π from q.[20]

For efficiency reasons, the current version of Patti does not allow for structure sharing. More precisely, there do not exist two non-empty paths π_1, π_2 such that $\delta(\pi_1, q_0) = \delta(\pi_2, q_0)$ where $\pi_1 \neq \pi_2$. For this reason, there can not exist a path $\pi \neq \varepsilon$ such that $\delta(\pi, q) = q$ for any $q \in Q$, i.e. cyclic feature structures can not arise.

Although disjunction and negation are not provided as an expressive means in general, they are supported for subsorts of *Atom*. Extending both the syntax and the semantics of the feature logic is rather straightforward; the reader might want to refer to [Carpenter, 1992] for a thorough mathematical discussion thereof.

3.1.4 Appropriateness

The appropriateness conditions "specify the features that are appropriate for each type and provide restrictions on their vaues in a way that respects the inheritance hierarchy" (cf. [Carpenter, 1992], p. 85). To ensure maximal run-time efficiency, the appropriateness conditions of Patti are subject to more rigid restrictions than those in Carpenters work.

An *appropriateness specification* over the inheritance hierarchy $\langle Type, \leq \rangle$ and features *Feat* is a partial function $Approp \in (Feat \times Type) \rightarrow Type$ that meets the following conditions:

- for every feature $f \in Feat$, there is a most general type $Intro(f) \in Type$ such that $Approp(f, Intro(f))$ is defined;[21]
- if $Approp(f, \sigma)$ is defined and $\sigma \leq \tau$, then $Approp(f, \tau)$ is also defined and $Approp(f, \sigma) = Approp(f, \tau)$;[22]

[20] More formally: $\delta(\varepsilon, q) = q$ and $\delta(f\pi, q) = \delta(\pi, \delta(f, q))$. Cf. [Carpenter, 1992], p. 37.

[21] This condition is exactly the same as Carpenter's Feature Introduction Condition (cf. [Carpenter, 1992], p. 86).

[22] This condition is similar to Carpenter's Upward Closure/Right Monotonicity Condition (cf. [Carpenter, 1992], p. 86). Carpenter though merely requires $Approp(f, \sigma) \leq Approp(f, \tau)$; in other words, subtypes are allowed to impose further constraints on the type of an inherited feature. Patti, in contrast, does not allow subtypes to additionally restrain the type of their features.

- if $\tau = \mathit{Approp}(f, \sigma)$ is defined and $\mathit{Matrix} \leq \tau$, then there exists no type $\zeta \neq \tau$ such that $\tau \leq \zeta$;[23]
- there is no sequence of features $f_1, \ldots, f_n$ and types $\sigma_1, \ldots, \sigma_n$ such that $\mathit{Approp}(f_i, \sigma_i) = \sigma_{i+1}$ for $1 \leq i < n$ and $\mathit{Approp}(f_n, \sigma_n) = \sigma_1$;[24]
- if $\mathit{Approp}(f, \sigma)$ is defined, σ must be subsumed by *Matrix*.

Speaking with Carpenter's terminology, all Patti feature structures are finitely totally typable.[25] Together with the appropriateness conditions stated above, this ensures that the compiler is able to determine the exact amount of memory needed to hold all feature values, including those in deeply embedded feature structures. Even more important, the compiler can thus assign a fixed offset to each feature. With the encoding utilized by most other feature-logic compilers (cf. section 2.2), the time to retrieve a feature value grows linearly with the path length. By assigning fixed offsets, this retrieval can be performed in constant time, since it is not needed anymore to follow a chain of pointers.

Notably, compilers for many programming languages exhibit a similar behavior when processing record-like structures. An example is given in fig. 3–1 below.

[23] In other words, any matrix type which is appropriate for a feature value must not have subtypes.

[24] This prohibits loops in the appopriateness specifications. It corresponds to the Substructure Requirement (cf. [Carpenter, 1992], p. 97) and ensures that all feature structures are finitely totally typable.

[25] A feature structure is totally well-typed if and only if every feature for which it is defined is appropriate and takes an appropriate value and furthermore, every feature which is appropriate must be given a value. In the formalism utilized by Patti, cyclic feature structures can not exist. Every totally well-typed feature structure hence has a least totally well-typed extension (cf. [Carpenter, 1992], p. 96), and is thus finitely totally typable.

```
class b {          | class b {          | class b {
  int       f;     |   class a   f;     |   class a   *f;
};                 | };                 | };
                   |                    |
class a {          | class a {          | class a {
  class b   g;     |   class b   g;     |   class b   g;
};                 | };                 | };
```

Fig. 3–1: *A compiler for C++ avoids the need to follow a chain of pointers for retrieving the value of a variable, even in the case of members of an embedded class. In the leftmost example, the compiler assigns an offset to g.f relative to the beginning of the data for a structure of type a; g is not realized as a pointer. However, this is not possible in all cases. The C++ syntax has been designed to circumvent situations such as the one depicted in the middle, where infinite loops would occur while determining the run-time size of a structure. The declaration will only be accepted by C++ compilers with f explicitly declared to be realized using a pointer, as depicted to the right.*

3.1.5 Enforcing Run-Time Presence

Experience has shown that linguists tend to define type hierarchies which specify more linguistic knowledge than actually needed for the specific tasks. As will become clear in the chapter about the actual compilation algorithm, it thus makes sense to filter out those types and features which are not used in any of the automata. However, it might be the case that other modules, external to the code generated by Patti, need to access certain linguistic features. Therefore, a mechanism is needed to enforce the run-time presence of features and types, even if they would not be needed for pattern matching.

For this reason, both types and feature definitions bear a flag called *forcePresence*. Setting this flag means that external modules might need to access them, and the compiler preserves the type or feature from being filtered out.

3.1.6 Disjoinability

Depending on the utilized Part-of-Speech Tagger, certain features can have multiple disjoined values, while the value of others is always a single atom.

For instance, the Lingsoft English Constraint Grammar (cf. [Voutilainen et al, 1992]) always gives a unique value for the feature *"Adjective Degree":* It is either *Absolute,* *Comparative* or *Superlative*,[26] but it never happens that the same reading of a token gets assigned multiple tags from this set. If a token could be both a comparative and a superlative adjective, the tagger would emit two separate readings. In contrast, the feature *"Person/Number"* can have multiple values for a *single* reading of a token, e.g. a disjunction of *2nd Person Singular* and *3rd Person Plural*.

Since the Patti system represents feature values as bit-vectors, the representation can be more compact if the knowledge about disjoinability is taken into account. For example, it makes sense to identify prepositions by some atomic ID.[27] For the sake of argument, let there be 420 different prepositions. If Patti is informed that this ID is never disjoined for a single reading, merely 9 bits — $\lceil \log_2 420 \rceil$ — are needed to represent this ID at run-time. If it can be disjoined, however, a bit is needed for each possible disjunct, totalling to 420 bits.

For this reason, feature definitions carry a flag called *isDisjoinable*. Please notice that clearing this flag does not imply that the respective values have to be unambiguous for every *token*. It merely indicates that they must not be ambiguous for a *reading* of a token; in case of ambiguity, several readings would have to be emitted by the tagger.

3.2 Grammar Specification

There is currently no means for linguists to specify the Patti input formalism in a text file. Certain modifications are performed via a direct-manipulative Graphical User Interface (entirely written in 100% Pure Java), while others require minor changes to the Patti code, followed by recompilation.

This seemingly rather awkward approach was chosen due to the following reasons:

- All persons which were working with the described system had full access to the sources of Patti, which is why there was no need for a text-based specification language.
- The author is convinced that a direct-manipulative Graphical User Interface (GUI) is superior to any text-based specification language. Not only does the visualization immediately reveal many concepts which otherwise would demand hours of training, but it also prevents a number of

[26] The Lingsoft tagger actually emits abbreviated (hence less readable) names for these tags.

[27] This corresponds to the PCASE feature, proposed for some grammar formalisms.

common mistakes. For instance, a well-designed interface prevents the user from specifying cyclic type hierarchies.

- With a GUI, most parts of the compilation can occur while the user is still editing or viewing the grammar formalism. This allows for more elaborated compilation algorithms without slowing down the edit-compile-debug cycle. Patti has been designed to support multi-threaded compilation, using Java synchronization techniques.
- A GUI can facilitate distributed Grammar Engineering. Although not supported in the current version, Patti has been designed with a client/server environment in mind, where multiple clients work on the same data. Because of strict adherence to the object-oriented Model-View-Controller (MVC) paradigm, any modification would be visible simultaneously with all clients.

The subsequent figures depict parts of the Graphical User Interface. Patti's Java code runs without modification on other platforms than MacOS, such as Apple Rhapsody, Microsoft Windows or UNIX/X-Windows. Of course, the graphical display would slightly differ, depending on the respective implementation of the Java Abstract Windowing Toolkit.

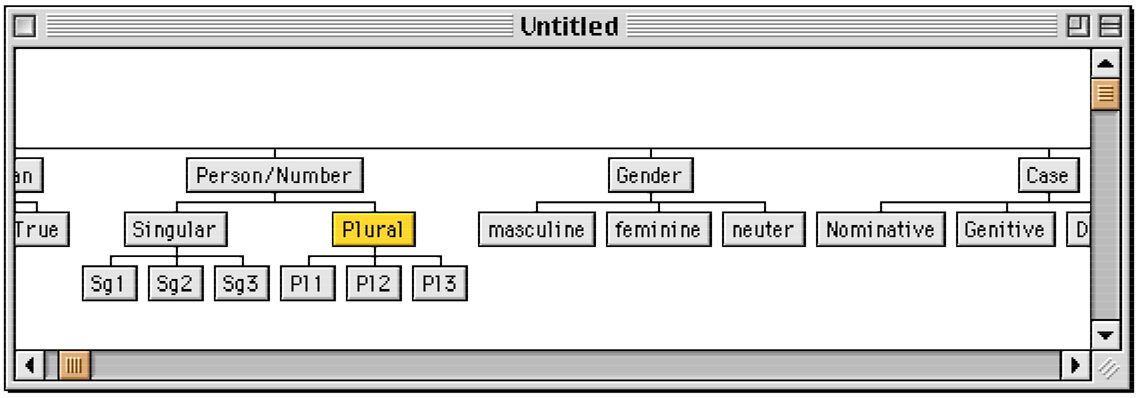
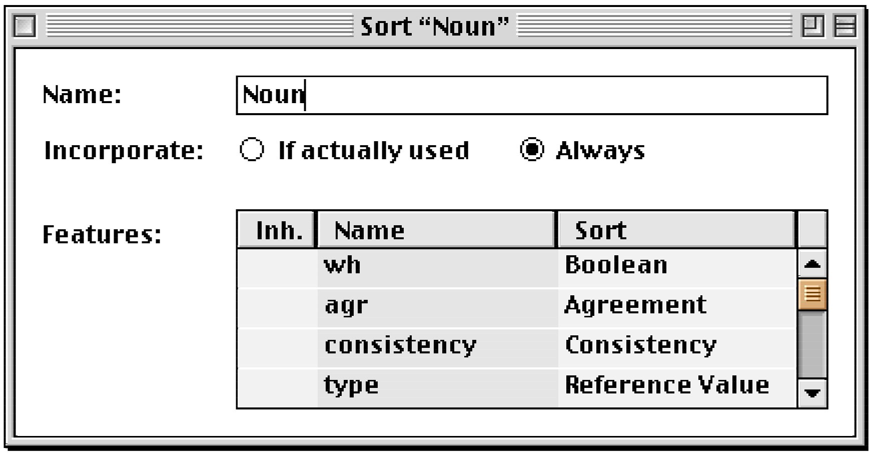


Fig. 3–2: *Many aspects of the type hierarchy can be edited with a direct-manipulative Graphical User Interface, developed in 100% Pure Java using Sun's Abstract Windowing Toolkit (AWT). Both standard and contextual menus are supported for actions such as insertion of new types. Double-clicking a type (or selecting a type, followed by choosing the "Open" menu item or pressing the Enter key) opens an editor window for the type, as depicted in figures 3–3 and 3–4.*


Sort "Noun"

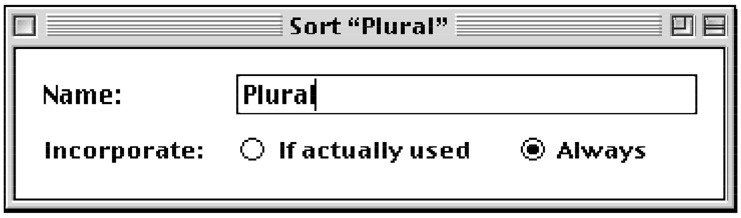
Name: Noun

Incorporate: ○ If actually used [✓ ◉ Always]

Features:

Inh.	Name	Sort
	wh	Boolean
	agr	Agreement
	consistency	Consistency
	type	Reference Value

Fig. 3–3: *The Patti GUI displays the appropriate features for a matrix type. The "Incorporate" radio buttons specify if the type should be present at run-time, even if never used in a transition label of any finite-state automaton. This allows for a smooth interaction with external modules, as described in section 3.1.4.*


Sort "Plural"

Name: Plural

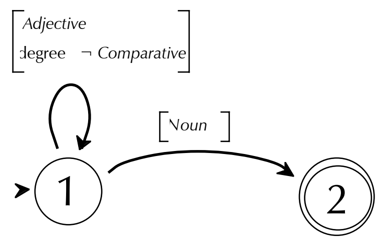
Incorporate: ○ If actually used [✓ ◉ Always]

Fig. 3–4: *A well-designed user interface can prevent common mistakes. For example, the underlying feature logic does not allow any features to be appropriate for atomic types such as "Plural". This is reflected in the interface by the lack of a panel for feature definitions. Hence, this type of errors can not occur.*

As noted above, only certain aspects of the input formalism are currently supported by the Graphical User Interface. Others need changes to the Java source code, followed by recompilation. The latter aspects include:

- modifying the appropriateness conditions for features of a matrix type;
- specifying finite-state automata;
- editing attribute-value matrices.

It is planned to enhance the GUI to cover these parts. Fig. 3–5 below illustrates how linguists have to specify a small finite-state automaton at the current time.



```
CAutomaton ng = new CAutomaton(d, "Noungroup");

CState     s1 = ng.start;
CState     s2 = new CState(ng);

CAVM       adj = new CAVM(d, "Adjective");
adj.addFeature("degree",
  new CNegation(new CAtomicValue(d.getSort("Comparative"))));

new CTransition(s1, s1, adj);
new CTransition(s1, s2, new CAVM(d, "Noun"));

s2.setAccepting(true);
```

Fig. 3–5: *Currently, finite-state automata and AVMs have to be specified directly in the Java code for Patti. The automaton depicted in the left part is specified with a couple of very simple, easy-to-understand Java function calls (right part).*

4 Runtime Execution

This chapter describes how the formalism of chapter 3 is processed at run-time. After an introduction to the utilized data structures, it is explained how the longest matches are detected in an efficient way, how an individual automaton is executed and how unifiability is checked by the generated code.

4.1 Data Structures

For each sentence, a data structure called *matchArray* is created to hold all information which is relevant to Patti. It is an array with an entry for each token of the current sentence, containing the following data:

- the *oracle word,* a bit-vector whose purpose is to enable rapid prediction whether a unification can possibly succeed. Details will be given later when discussing compilation in chapter 5.
- a pointer to a data structure with all data for a token, most of which have no importance for Patti. In the context of the present thesis, the only entry of interest is a pointer to the first item in the linked list of readings for the token. The data structure for a single reading consists of the following:[28]

[28] Not included is a large bit-vector to encode the original tags, as assigned by the Lingsoft English Constraint Grammar. This is needed for debugging, but is of no actual use for Patti. The code generated by Patti hence does not access this data. Substituting ENGCG with a proprietary Part-of-Speech Tagger would entirely void this bitvector.

  - a pointer to the next ambiguous reading, or NULL if there is none;
  - an integer to encode the type of the reading;
  - a bit-vector to hold the individual feature values. The length of this bit-vector depends on the above type.
- an array of pointers, one for each automaton in the grammar. If a match range ends at the current token, this entry points back to the beginning of the range; otherwise, its value is NULL.

These data structures are best explained graphically: fig. 4–1 depicts the data for a simple sentence, while fig. 4–2 illustrates the data structures for a token with two readings.

	0	1	2	3	4	5	6	7	8
	■	the	red	cat	sat	on	the	drop	■
Oracle	00000	00001	01000	10000	00010	00100	00001	10010	00000
Token	—	*Ptr*	*Ptr*	*Ptr*	*Ptr*	*Ptr*	*Ptr*	*Ptr*	—
m1	—	—	—	2	—	—	—	7	—

Fig. 4–1: *The basic run-time data structure for Patti is an array with an entry for each word of the sentence (columns 1–7). Before and after the actual data, there are entries which never will be part of a match (columns 0 and 8). The "m" cells represent the longest matches for each automaton: in the illustration, the automaton #1 has found a match ranging from col. 2 to col. 3 ("red cat"), and another one ranging from col. 7 to col. 7 ("drop").*

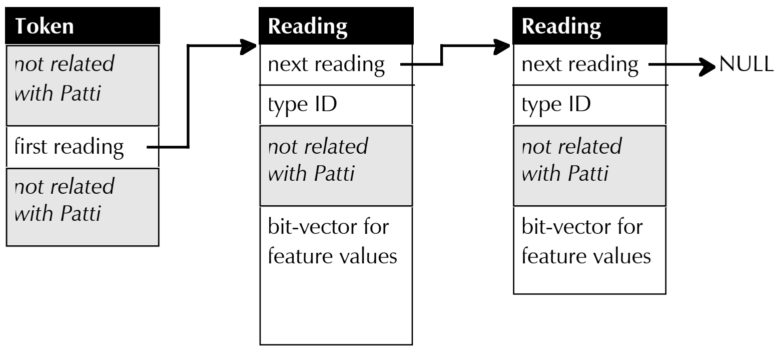


Fig. 4–2: *The ambigous readings of a token are represented as a linked list.*

4.2 Matching in a Sentence

In case there are several automata in the grammar, each automaton *X* processes the entire sentence on an individual basis. After being finished with a sentence, the appropriate *mX* cells will contain pointers indicating the maximal ranges of the matching tokens.

Algorithm: MatchingInSentence (for automaton X)

Input:

- *matchArray* — an array of match slots, as described in section 4.1. All *mX* cells are expected to have a NULL value.
- *sentenceLength* — number of tokens in the sentence.

Output: none.

Side-Effects: The *mX* cells in *matchArray* are set according to the longest ranges of matching tokens.

Method:

```
maxAcceptedEntry := 0;
for i := 1 to sentenceLength do
  acceptedEntry := ExecuteAutomaton(matchArray, i);
  if (acceptedEntry > maxAcceptedEntry)
        matchArray[acceptedEntry].mX := i;
        maxAcceptedEntry := acceptedEntry;
  endif;
enddo;
```

Fig. 4–3: *For each token, it is determined where the longest match ends when the automaton starts at that token. The match is only recorded if its end is the rightmost up to now. This excludes multiple matches where one is fully included in the other.*

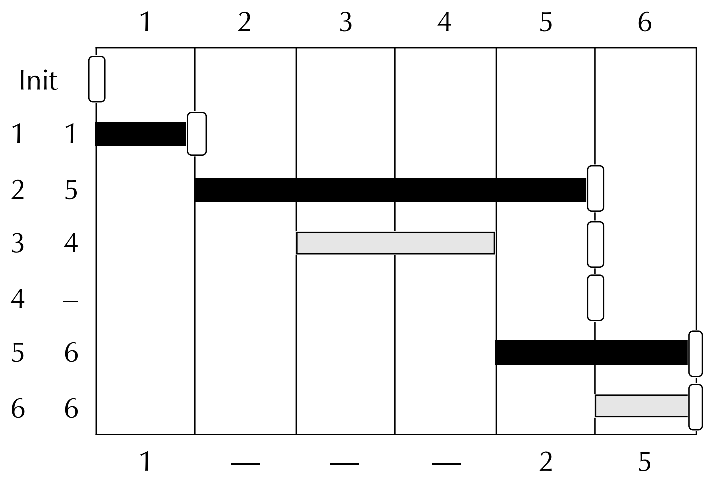


Fig. 4–4: *A visualization of the algorithm described in Fig. 4–3. The leftmost numbers indicate the value of i and acceptedEntry, respectively. For i = 4, no accepting state was reached in the automaton. With typical automata, this happens in most cases. The small white rectangles (▯) depict the value of maxAcceptedEntry. Those match ranges that are actually recorded are painted in black, while gray bars stand for match ranges that are not stored because acceptedEntry was less than or equal to maxAcceptedEntry. The bottom line indicates the final value of the "m" cells.*

4.3 Executing an Automaton

An eminent factor for the high performance of Patti is that automata are not compiled to tables which are interpreted by some driver routine. Instead, specific machine instructions are generated for each state and each transition. In order to facilitate understanding the algorithm, it however makes sense to specify the execution of automata in the traditional way.

The algorithm calculates the set of configurations which the current configuration yields in one step. One of them will become the current configuration of the next pass, while the others are pushed onto a stack. If the automaton can not consume any more symbols, either because no transition label can be unified with the current token or because the end of the sentence has been reached, the next configuration is retrieved from the stack. If this is not possible, because the stack is empty, the rightmost token is returned which was consumed when entering an accepting state.

In the worst case, the execution time grows exponentially with the input length. In practice, however, the non-determinism stack hardly ever gets utilized.

Algorithm: ExecuteAutomaton

Input:

- *matchArray* — an array of match slots, as described in section 4.1;
- *startToken* — number of token to be consumed first.

Output: number of rightmost token consumed upon entering an accepting state, or NULL if no accepting state was entered.

Side-Effects: none.

Method:

```
state := s; /* start state of automaton */
curToken := startToken;
lastToken := lengthOf(matchArray); /* index of last slot */
result := NULL;
loop
  if ($state \in F$) /* is current state accepting? */
        result := max(curToken, result);
  endif;
  if (curToken < lastToken)
     and $\exists\sigma\,\exists$ targetStateNow: ($\langle$state, $\sigma$, targetStateNow$\rangle \in \Delta$
              $\wedge$ BIT-AND(matchArray[curToken].oracle, oracle($\sigma$)) $\neq 0$
              $\wedge$ $\sigma \sqcup$ matchArray[curToken].token $\neq \bot$)
        for $\forall\tau\,\forall$targetStateDeferred: ($\langle$state, $\tau$, targetStateDeferred$\rangle \in \Delta$
              $\wedge$ $\tau \neq \sigma$
              $\wedge$ BIT-AND(matchArray[curToken].oracle, oracle($\tau$)) $\neq 0$
              $\wedge$ $\tau \sqcup$ matchArray[curToken].token $\neq \bot$) do
              push $\langle$targetStateDeferred, curToken + 1$\rangle$;
        enddo;
        curToken := curToken + 1;
        state := targetStateNow;
  elseif (StackNotEmpty())
        $\langle$state, curToken$\rangle$ := pop;
  else
        return result;
  endif;
endloop;
```

Fig. 4–5: *Although specific machine instructions are generated for every state, every transition and every feature unification of an automaton $\langle K, \Sigma, \Delta, s, F\rangle$, the algorithm is best described as if it was driven by tables for the states and transitions. The oracle mechanism is described below in section 5.3; the formal definitions of automata and feature structures were introduced in chapter 3.*

4.4 Unification

In constrast to most other feature-logic natural language processing systems, the actual result of the unification operation is not needed with Patti. To execute the automata, it is only necessary to know whether transitions can be taken, i.e. whether or not the unification fails. This can increase efficiency, since no data structures have to be built.

As already mentioned several times, Patti compiles machine instructions out of the transition labels. Basically, the run-time unification loops over the readings of the current token to check if one of them is satisfying the restrictions stated by the transition label. If none of the readings passes these tests, the unification fails.

Let the transition label be $FS = \langle Q, q_0, \theta, \delta \rangle$ and the current reading be $FS' = \langle Q', q_0', \theta', \delta' \rangle$. The first test is type unification: It is checked whether the type of the transition label is consistent with the type of the current reading, which is the case if either $\theta(q_0) \leq \theta'(q_0')$ or $\theta'(q_0') \leq \theta(q_0)$. Next, it is checked for every path $\pi \neq \varepsilon$ in FS whether FS' satisfies the description stated by π:

- If the first feature in the path is introduced by some subtype of $\theta(q_0')$, this means that the unification can not fail due to this description. While other unifiers would copy the value (i.e. $\theta(\delta(\pi, q_0))$) into the result, nothing has to be done in the case of Patti, since a mere copy operation can not cause the unification to fail.
- If $\theta(\delta(\pi, q_0))$ is subsumed by *Matrix*, the third and fifth appropriateness condition of section 3.1.4 together guarantee that the embedded feature structures in both FS and FS' are of the same type. The unification then can not fail due to this description, and no checks have to be performed.
- Otherwise, $\delta(\pi, q_0)$ is an atomic feature whose value is actually present in the current reading. The generated code loads the corresponding bits into registers and performs the check, depending on the run-time representation of the value. If the *isDisjoinable* flag (cf. section 3.1.6) was set upon feature introduction, the value is represented as a bit-vector. Unifiability is then checked with bit-AND instructions, similar to the method described in section 2.3.2. Otherwise, the value is represented as an integer number; unifiability is checked with a test for numeric equality.

5 Compilation Algorithm

This chapter explains how Patti compiles the finite-state automata and type hierarchies of chapter 3 into machine-independent Intermediate Code, so that the automata can be executed at run-time as described in chapter 4.

A first section discusses how the type and feature definitions are reduced to those parts which are actually required for the respective tasks. Then, a specialized mechanism to speed up unifications is introduced, before the generation of intermediate code is described in a final section.

5.1 Extraction of Used Parts

Seemingly, linguists tend to specify much more knowledge about the described language than is actually needed for a given task. For example, the type hierarchy which was used as input to Patti at Apple Computer[29] contains a large number of types for different kinds of anaphora, although none of the current automata ever require some specific kind. Although this might make sense from a purely linguistic perspective, it is not ideal for implementation: a typical grammar contains numerous types and features which are not required to determine the desired results. For this reason, the first compilation step is to reduce the input type hierarchy to a subset which is actually used by the automata.

[29] A concrete application of the Patti system was briefly described in section 1.3.

5.1.1 Mark Used Types and Appopriateness Conditions

The first step to determine the actually utilized subset of types and feature introductions is to mark those types which are specified in the transition labels of finite-state automata. Every type and feature appropriateness specification bears a flag named *inUse,* initially set to false. A *markUsage* method is invoked for every transition label in any automaton. Depending on the object class,[30] *markUsage* performs the following:

- in case of a feature structure, the *inUse* flag of its type is set. For every feature in the structure, the appropriateness condition is marked to be used, and the *markUsage* method of the specified feature value is called;
- in case of an atomic feature value, *inUse* of the corresponding type is set;
- in case of a negation, *markUsage* is called for the negated value;
- in case of a value disjunction, *markUsage* is called for every disjunct.

5.1.2 Mark Objects with Enforced Run-Time Presence

As described in section 3.1.5, the run-time presence of a type or a feature can be enforced, even if this type or feature is never accessed in the code generated by Patti. For each type whose *forcePresence* flag is set, the compiler sets its *inUse* flag. For each feature appropriateness specification whose *forcePresence* flag ist set, the compiler sets the *inUse* flag of the type which is appropriate for the values of that feature.

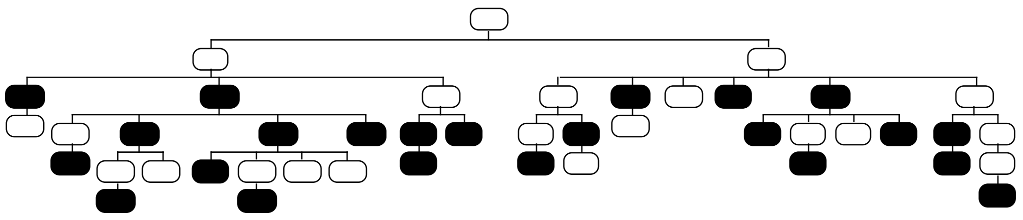


Fig. 5–1: *The steps described in 5.1.1 and 5.1.2 determine those types which will be part of the run-time representation (black rectangles) and those which will not (white rectangles).*

[30] Patti is implemented using object-oriented techniques. All classes for feature values — feature structures, atomic values, value negations and value disjunctions — provide a *markUsage* method.

5.1.3 Extract Marked Matrix Types

A pre-order depth-first search is performed on the matrix section of the type hierarchy. Types whose *inUse* flag is not set are ignored. For any other matrix type *M*, an object *CM* of class *CompMatrix* is created to hold the following information:

- A 16-bit integer as run-time identification number for the type. To the first type whose *inUse* flag is true, an ID of 1 is assigned. Each subsequent type gets an ID which is incremented by 1. Because *M* is encountered in the traversal before any of its subtypes, any subtype of *M* with a run-time realization will get a greater ID than *M*.
- An array with the feature definitions which are introduced by *CM*. Not only do these comprise the feature definitions introduced by *M,* but also the feature definitions introduced by any supertype of *M* up to, but not including, the closest supertype whose *inUse* flag is true.

The relation of matrix types to their subtype is preserved; the resulting *CompMatrix* objects are organized in trees as well. The described mapping process hence generates a forest of *CompMatrix* objects, where each tree (called *cluster*) corresponds to a subpart of the original type hierarchy.

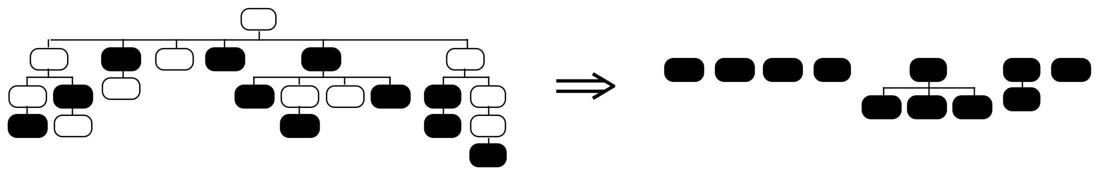


Fig. 5–2: *The matrix part of the type hierarchy is mapped into a forest of objects. Only those parts of the hierarchy that are actually in use (indicated by black color) will form a part of the result structure.*

5.1.4 Extract Marked Atomic Types

A pre-order depth-first search is performed on the atomic part of the type hierarchy as well. If a type is neither used in the patterns nor declared to be included at run-time, it is ignored. However, for any atomic type *A* whose *inUse* flag is set, a *CompAtom* object is created. Corresponding to the dominance relation of the original type hierarchy, the CompAtom objects are organized in a forest of trees or *clusters*.

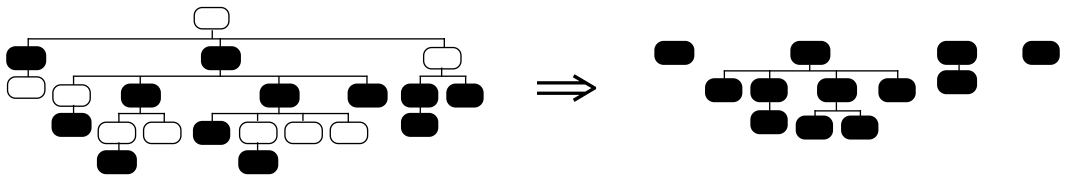


Fig. 5–3: *The atomic part of the type hierarchy is mapped into a forest of CompAtom objects. Only those types whose inUse flag is set (depicted in black) become part of the result structure; the others (depicted in white) will not be present at run-time.*

5.2 Calculation of Run-Time Representation

Having determined which types and appropriateness conditions will be present at run-time, the next step is to calculate the actual run-time representation for the surviving objects.

5.2.1 Count Leaves in Atomic Clusters

The just created forest of *CompAtom* objects is traversed by performing a pre-order depth-first search on every atomic cluster. For each *CompAtom* object *CA*, the following information is calculated:

- the number of leaves to the left of *CA* in the same cluster. For the top-most node in the cluster, this is zero;
- the number of leaves in the subtree dominated by *CA*. For a leaf, this number is 1, because every node is contained in the tree which is dominated by itself;
- the number of leaves to the right of *CA* in the same cluster. For the top-most node in the cluster, this is zero.

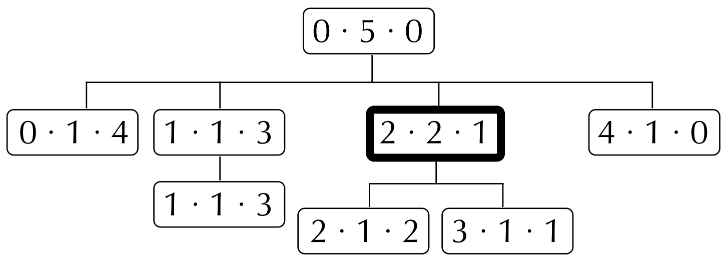


Fig. 5–4: *Counting the leaves in an atomic cluster. The node with the thick frame has two leaves to its left (first number); it dominates a subtree with two leaves (second number); to its right, there is one leaf (third number).*

5.2.2 Calculate Bit-Vector Size for Matrix Types

The run-time representation of a matrix type has been described in section 4.1. It includes a 16-bit integer type ID, and a bit-vector to hold the individual feature values. In section 5.1.3, it has been described how the type ID is obtained; the subsequent sections will now cope with the assignment of features to bits in this vector.

For every utilized appropriate feature, the compiler assigns some bits in the run-time vector. For this purpose, a method *calcRuntime* is called for every *CompMatrix* object *CM*. It checks first whether the object's run-time representation has been calculated previously. If this is not the case, the following actions take place:

- If *CM* is the topmost of its cluster and thus has no supertype, the size of the associated bit-vector is initially set to zero.
- If *CM* has a supertype, *calcRuntime* is called for that supertype, before the mapping from features to bit-vector positions is copied. This ensures that inherited features are stored at the same vector positions as with the type which has introduced them.
- For every feature *f* whose appropriateness is introduced by *CM* (i.e. *Intro*(*f*) = *CM*), a range of bits is reserved in the bit-vector. Let be τ = *Approp*(*f*, *CM*).
    - If τ is subsumed by *Matrix*, *calcRuntime* is called for τ, and the number of bits needed to hold a structure of type τ is added to the length of *CM*'s bit-vector.

- If τ though is subsumed by *Atom*, the number of bits to be reserved depends on the disjoinability of *f* (cf. section 3.1.6). Let *n* be the number of leaves in the atomic cluster to which τ belongs. If the *isDisjoinable* flag is set, *n* bits are reserved; otherwise, merely $\lceil \log_2 n \rceil$ bits are required to hold the value of *f*.

5.2.3 Compile Transition Labels

Having completed the steps described in the previous sections, Patti now compiles each feature structure *FS* which labels a transition in an automaton. First, it is determined which type IDs a reading can possibly have in order to be unifiable with *FS*. Assuming ζ to be the type of the reading and σ to be the type of *FS*, there are three cases to distinguish:

- ζ is subsumed by σ. Due to total well-typedness, all features in *FS* are specified in the reading. If ζ is a proper sub-type of σ, ζ might contain more features than σ does.[31] However, since Patti is only interested in whether the unification fails, the values of these additional features do not matter.
- ζ is a proper super-type of σ. In this case, it can happen that some features in *FS* do not have a counterpart in the current token's reading; the corresponding checks thus have to be omitted.[32] See fig. 5–5 for an illustration thereof.
- ζ and σ are inconsistent types. In this case, the unification fails.

Let *R* be *TypeID*(ζ), i.e. the numeric type ID of the current reading. Because the assignment of type IDs is based on a pre-order depth-first search (cf. section 5.1.3), the first of the above cases is indicated by $TypeID(\sigma) \leq R \leq TypeID(lastUnif(\sigma))$, where *lastUnif*($\sigma$) is the type with the largest ID among those consistent with σ. If *R* is in this range, all features in *FS* have a counterpart in the reading, and corresponding unificaton code will be generated for each feature.

If *R* does not fall between *TypeID*(σ) and *TypeID*(*lastUnif*(σ)), it is checked for every supertype τ of σ (i.e. for all types τ such that $\tau < \sigma$) whether $R = TypeID(\tau)$. In that case,

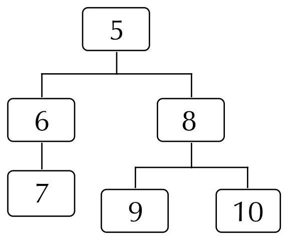
[31] This is the case if there exists a feature $f \in Feat$ such that $\sigma \leq \zeta$, $\sigma \neq Intro(f)$ and $Intro(f) \leq \zeta$.

[32] More formally, this occurs if $\zeta \leq \sigma$, $\zeta \leq Intro(f)$ and $Intro(f) \leq \sigma$ for a feature $f \in Feat$.

only a subset of the features in *FS* might have a counterpart in the reading to be unified with. If *R* is not equal to the ID of any super-type of σ which is also present at run-time, the unification fails.

Note that the root of the cluster to which σ belongs is the most general type consistent with σ and used at run-time; let *firstUnif*(σ) be that type. Since it makes sense to detect unification failures as quick as possible, one of the first actions that happen at run-time is to check whether $TypeID(firstUnif(\sigma)) \leq R \leq TypeID(lastUnif(\sigma))$.

$$\begin{bmatrix} \textit{8} & \\ & \\ g & \ldots \\ h & \ldots \end{bmatrix}$$



Feature *f*	**Intro(*f*)**
g	5
h	8
i	9

Fig. 5–5: *Not all features in a transition label need to be appropriate for the type of a unifiable run-time feature structure. Assuming the type hierarchy in the middle and the features being introduced as depicted in the table to the right, readings of type 5, 8, 9 and 10 are unifiable with the transition label AVM to the left. While both g and h have to be checked when unifying with a reading of type 8, 9 or 10, only g has to be considered upon unification with a reading of type 5. firstUnif(8) is 5, lastUnif(8) is 10.*

For the transition label for fig. 5–5, Patti would generate two versions of the code: a version to check both *g* and *h*, to be executed when $R \in \{8, 9, 10\}$, and a version to check *g* only, to be executed when $R = 5$.[33]

To test the unifiability of *FS* with a reading of type ζ, a number of machine words are loaded from memory into registers, and certain checks have to be performed on each word. An multi-dimensional array $accessedWord_{\zeta,\, i}$ indicates the *i*-th word to be loaded; $checks_{\zeta,\, i}$ holds the checks to be executed on that very word.

The compiler now processes every path π in *FS*:

- Due to the reasons stated above, π is to be ignored if ζ properly subsumes the type which introduces π.
- If $\theta(\delta(\pi, q_0))$ is subsumed by *Matrix*, π is ignored, since non-atomic embedded feature structures have no run-time counterpart.

[33] The **typeDispatch** intermediate instruction branches to the respective code section, depending on the value of *R*. The intermediate instructions are described in Appendix B.

- Otherwise, $\delta(\pi, q_0)$ is an atom — or a set of atoms, connected by disjunction and negation symbols. The position and length of the sub-vector holding the value of π have already been determined by the compilation step described in section 5.2.2.[34] Therefore, it is fairly easy to determine which machine words have to be loaded for π, and *accessedWords* is extended accordingly.
  - If the *isDisjoinable* flag is set, the run-time bit-vector is determined for each atom in $\delta(\pi, q_0)$ by concatenating *left* "0" bits, *below* "1" bit and *right* "0" bits (*left*, *below* and *right* being the number of leaves calculated by the method of section 5.2.1). If two atoms are connected by disjunction, their vectors are bit-ORed; if two atoms are connected by conjunction, their vectors are bit-ANDed; if an atom is negated, the complement of its vector is taken. *checks* is modified to hold an AND check (cf. section B.3.7) with the resulting vector for $\delta(\pi, q_0)$.
  - If the *isDisjoinable* flag is not set, all atoms in $\delta(\pi, q_0)$ are replaced by a disjunction of the atomic leaves they subsume. By applying the usual transformation rules of Boolean algebra, the value is then brought into conjunctive normal form. Single negated atomic leaves are realized as NEQ, non-negated as EQ checks (cf. section B.3.7). In case of a single (possibly negated) leaf, the EQ or NEQ object is added to *checks*; in case of multiple leaves (joined by Boolean connectives), a CNF object is created and added to *checks* to hold the structure of the expression.

After having processed all paths, *firstVWAcc*(σ) is set to hold the offset of the very first bit-vector word which has to be accessed at run-time. The PowerPC code generation uses this information to issue data cache control instructions.

5.3 The Unification Oracle

In a typical NLP system, the major part of the performed unification operations do fail. It would be desirable to have a kind of "oracle" which reliably predicts whether a unification will fail, so that the time to perform the unsuccessful unification can be saved.

This section presents such an oracle which greatly contributes to Patti's efficiency. Note that in general, the oracle does not replace unification: it merely determines whether

[34] Let be n the length of the bit-vector for ξ before reserving the m bits to hold the value for π. The sub-vector then starts at bit n and has a length of m.

or not a unification is worth trying. In the former case, the unification could still fail, despite of the oracle's prediction. In the latter case, however, it is sure that the unification will actually fail: The oracle's failure predictions are guaranteed to be correct.[35]

5.3.1 A Unification Failure Oracle for Flat Hierarchies

For the sake of simplicity, the reader might neglect for a moment the inheritance hierarchy and assume that two AVMs be only unifiable if they are of the very same type. The complications due to type unification will be discussed later in section 5.3.2.

Let the run-time representation of a token contain a bit-vector of size 32, called *oracle word*.[36] Further, let *oracle*(σ) be for any type σ a bit-vector of size 32, whose only bit set is the bit with the number (TypeID(σ) mod 32).

When the tagger output for a token is converted into the Patti format, all bits of its oracle word *OW* are initially set to zero. For each reading with type τ, *OW* is bit-ORed with *oracle*(τ); the result is then stored back to *OW*.

The oracle now works as follows: In the generated machine code for each state of the finite-state automata, one of the first actions is to load the oracle word of the current token into a register. Before performing the unification with a transition label of type ζ, the oracle word is bit-ANDed with *oracle*(ζ). If the result is an all-zero vector, the unification is cancelled immediately. Note that on many processors, performing a bit-AND on a register and comparing the result with zero is an extremely efficient operation. On PowerPC, this test consumes one single machine cycle.[37]

[35] A similar kind of "failure oracle" is common practice in Natural Language Processing to speed up syntactic analysis. For example, the *FIRST* set, employed in numerous Chart Parsing algorithms for context-free grammars, constitutes a comparable filter: membership of the subsequent terminal's category in the *FIRST* set for the category of an active edge does not assert that the edge actually forms a part of the final solution. Non-membership though guarantees that an edge can be safely omitted.

[36] Most current microprocessors have 32-bit integer registers. Since logical operations are most efficient if their arguments fit into a single register, 32 was chosen as size for the oracle word.

[37] This is only true if independent instructions can be scheduled between the bit-AND/comparision and the conditional branch, or if the compiler is able to predict correctly whether the conditional branch will be taken. Otherwise, a small delay can occur. See sections A.2.2 and A.2.3 for an introduction into Superscalar Instruction Dispatch and Branch Prediction.

Why does this work? Assume the current token had a reading of type ζ. Due to the calculation of *OW* described above, the bit number (*TypeID*(ζ) mod 32) would be set in *OW*. This bit would be common to both vectors *OW* and *oracle*(ζ). Thus, the result of a bit-AND operation of *OW* and *oracle*(ζ) would be a vector with this bit set. Hence, the bit-AND operation would lead to a non-zero result. But in that case, the oracle would not have predicted the unification to fail.

5.3.2 A Unification Failure Oracle for Simple-Inheritance Hierarchies

With a type *hierarchy,* a transition label feature structure of type σ can unify with a token even if none of the readings are of type σ — it suffices that the type of one of the readings be *unifiable* with σ. As mentioned above, this slightly complicates the oracle.

The run-time algorithm — perform a bit-AND of the current token's oracle word with *oracle*(σ), predict failure if the outcome is an all-zero vector — does not need to be modified. However, a different method is employed to calculate *oracle*(σ).

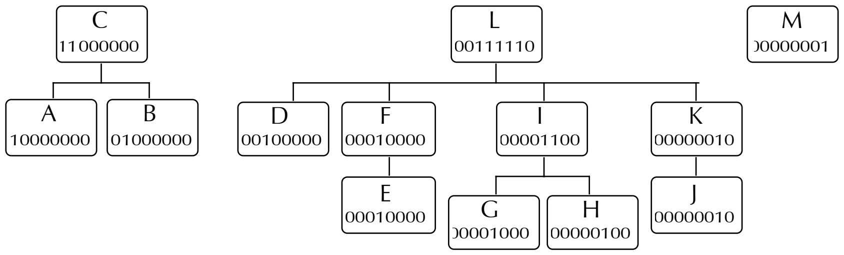
The algorithm is based on a commonly known type unification algorithm for multiple-inheritance hierarchies ([Aït-Kaci et al., 1989]; cf. section 2.3 for a brief description). In contrast to Aït-Kaci et al., Patti is not interested in the *result* of the type unification, but solely in the fact whether or not the unification fails. In addition, the problem is further alleviated because the current formalism merely allows for simple inheritance type hierarchies.

After setting an Integer variable *leavesEncountered* to zero, a post-order depth-first search is performed on the forest constructed in section 5.1.3 ("Extract Marked Matrix Types"). For each encountered matrix type σ, the 32-bit vector *oracle*(σ) is calculated as follows:

- If σ dominates no subtypes (i.e. σ is a leaf in the type hierarchy): *oracle*(σ) is a bit-vector whose bit number (*leavesEncountered* mod 32) is set, whereas all other bits are cleared. Afterwards, increment *leavesEncountered* by 1.
- If σ is not a leaf: Set *oracle*(σ) to the result of a bit-OR on all *oracle* vectors of the dominated subtypes.

The described algorithm establishes a mapping from leaves in the type hierarchy to positions in the vector (see right part of figure 5–5 for an example). If two matrix types are unifiable, they have at least one leaf in common, thus the corresponding *oracle* words have at least one commonly set bit. Therefore, bit-AND of two vectors can lead only to an all-zero result if two types are not unifiable (as with the algorithm by Aït-Kaci et al.).

However, it is in general not valid to deduce unifiability from non-zero results. If there are more than 32 leaves, several leaves will share the same bit in the vector.



eaf	Bit
A	0
B	1
D	2
E	3
G	4
H	5
J	6
M	7

Fig. 5–5: *The forest of used matrix types is traversed in a post-order depth-first search. In this example hierarchy, the type names correspond to the order in which the nodes are visited. For a leaf, the corresponding bit in its vector is set (see table to the right). For other types, the vector is obtained by bit-ORing the vectors of all sub-types.*

Another difference to the algorithm of Aït-Kaci et al., besides the restriction to simple inheritance, is that the result of a bit-AND is not indicative for the type of the unification result. In fig. 5–5, for example, the very same bit-vector is assigned to both E and F. Since the bit-AND is only used for oracle purposes and not to determine the actual unification result, this does not affect the usability of the algorithm.

Of course, it would have been possible to assign a bit to every type (as Aït-Kaci et al. do), not only to the leaves. This would enlarge the size of the bit vectors. However, since the *oracle* vectors are limited to 32 bits, this would make it more probable for a bit to be shared by several inconsistent types, hence decreasing the number of cases where the subsequently discussed Unification Success Oracle is applicable.

5.3.3 The Unification Success Oracle

The Unification Oracle, described in the previous section, is able to reliably predict most cases of unification failures due to type mismatches. However, as has been pointed out beforehand, it is not a reliable prediction for unification success: Even if the oracle predicts success, the unification might fail and thus still has to be tested before a transition can be taken. Nevertheless, there are specific conditions under which the oracle is indeed able to tell reliably whether a feature structure *FS* of type σ is unifiable with a token whose oracle word is *OW,* without performing the actual unification:

- *FS* is an *empty* feature structure. In this case, type inconsistency is the only reason for which the unification could possibly fail;
- from the fact that a non-zero result is obtained from bit-ANDing *OW* with *oracle*(σ), it can be deduced that the token has a reading whose type is indeed unifiable with σ. This condition is fulfilled if the variable *leavesEncountered* is less than or equal to 32 after being finished with the forest traversal described in section 5.3.2.[38]

These conditions are easy to verify at compilation time. If they both hold for the label of a transition, that unification can be safely omitted, since then, the oracle predicts unification success if and only if the unification would be successful. Thus, if a transition label feature structure is empty, and if not too many types are in use, the unification is compiled to a single bit-AND operation on a register, resulting in a tremendous impact on efficiency.

This optimization can be applied surprisingly often. In fact, most transitions were labeled with empty feature structures in the automata used at Apple Computer.

5.4 Intermediate Code Generation

For every automaton, a sequence of Intermediate Code instructions is generated. The purpose of the Intermediate Code, described in appendix B, is to keep the machine-dependent parts of the compiler sources in a small, designated place. With this separation, it is much easier to port Patti to a new target instruction set — only the code generation out of the intermediate code would have to be enhanced, but not the compiler itself.

[38] If the number of leaves in the matrix forest is between 33 and 63, certain bits are shared by several leaves, while others are unique. The Patti compiler is slightly more complicated than described — it determines which bits of an oracle word are unique and which are not — in order to allow for this optimization with hierarchies that use up to 63 matrix types.

5.4.1 Intermediate Code for Automata

The intermediate code for an automaton consists of the instructions listed below. Each automaton bears a distinctive identification number, which will be part of the function name of the final code. The start state always gets an ID of 1; its code thus immediately follows the **prolog** instruction. Let *startAccepting* be "yes" if the start state is accepting and "no" otherwise.

Once:

	declaration	automatonID = ⟨ID of Automaton⟩
	prolog	startStateAccepting = ⟨*startAccepting*⟩
	— code for for each state, according to section 5.4.2 —	
	epilog	
	declarationEnd	

5.4.2 Intermediate Code for States

Let *ID* be the identification number of the state *State* whose intermediate code is being generated. Let $Target_1$, …, $Target_n$ be the IDs of the target states of the transitions leaving *State*, let $Accepting_1$, …, $Accepting_n$ be "yes" if the respective target state is accepting and "no" otherwise, and finally let $Label_1$, …, $Label_n$ be the labels of the respective transitions.

The intermediate code for *State* then consists of the subsequent parts; together, they implement the algorithm of section 4.3.

Once:

state⟨ID⟩	**stateEntry**	accepting = ⟨yes | no⟩

For $1 \leq i < n$:

tr⟨ID⟩_⟨i⟩	code for $Label_i$	failAction = "**jump** target=tr⟨*ID*⟩_⟨*i* + 1⟩", label = "tr⟨ID⟩_⟨i⟩", successCode =	
		call	target = n⟨*ID*⟩_⟨*i*⟩
		jump	target = state⟨$Target_i$⟩

Once:

tr⟨ID⟩_⟨n⟩	code for *Label*$_n$	failAction = "**jump** target=pop", label = "tr⟨ID⟩_⟨n⟩", successCode = **jump** target=state⟨*Target*$_n$⟩

For $1 \leq i < n - 1$:

n⟨ID⟩_⟨i⟩	code for *Label*$_i$	failAction = "**jump** target=n⟨*ID*⟩_⟨*i* + 1⟩", label = "n⟨ID⟩_⟨i⟩", successCode = **jump** target=push⟨ID⟩_⟨i⟩
push⟨ID⟩_⟨i⟩	**push**	newState = ⟨*Target*$_i$⟩, newStateAccepting = ⟨*Accepting*$_i$⟩

Once:

n⟨ID⟩_⟨n–1⟩	code for *Label*$_{n-1}$	failAction = "**return**", label = " n⟨ID⟩_⟨n–1⟩", successCode = **jump** target=push⟨ID⟩_⟨n–1⟩
push⟨ID⟩_⟨n-1⟩	**push**	newState = ⟨*Target*$_{n-1}$⟩, newStateAccepting = ⟨*Accepting*$_{n-1}$⟩
	return	

5.4.3 Intermediate Code for Transition Labels

The routine to generate the intermediate code for a transition label takes two arguments: First, a branch instruction *failAction* to be executed on unification failure, and a sequence of intermediate instructions *successCode* to be executed on unification success.

The code for a transition label *FS* always starts with an invocation of the unification oracle. The oracle was described in section 5.3; its purpose is to predict reliably whether a unification will fail due to type inconsistencies. In the general case, the oracle filters out most candidates which will lead to unification failures, thus greatly saving execution time. In the code below, let *oracleWord* be the oracle word for *FS*, as determined by the algorithm of section 5.3.2.

Under certain conditions itemized in section 5.3.3, the oracle is even able to predict reliably whether a unification will succeed. In this special case, the actual unification does not need to be performed at all, and the **oracle** instruction is immediately followed by the code to be executed on unification success, which is passed as *success-*

Code parameter. The code for the transition label then looks as follows:[39]

Once:

```
                oracle            oracleWord=⟨oracleWord⟩,
                                  prediction=doesMatch,
                                  failAction=⟨failAction⟩
                ⟨successCode⟩
```

Otherwise, the unification oracle does filter out many mismatches, but nevertheless, the unification checks still have to be executed. The code then loops over all readings, until one is unifiable with *FS* (let σ be the type of *FS*):

Once:

```
                oracle            oracleWord=⟨oracleWord⟩,
                                  prediction=doesMatch,
                                  failAction=⟨failAction⟩
                readingLoopInit   firstVectorWordAccessed=⟨firstVWAcc(σ)⟩
⟨label⟩_do      readingLoopBody   failAction=⟨failAction⟩,
                                  firstVectorWordAccessed=⟨firstVWAcc(σ)⟩,
                                  firstUnif=⟨firstUnif(σ)⟩,
                                  lastUnif=⟨lastUnif(σ)⟩
```

As discussed in section 5.2.3, it depends on the type of the encountered reading whether or not certain checks have to be executed in order to verify unifiability. Let $\zeta_1, \ldots, \zeta_k$ be the realized super-sorts of σ in order of descending generality, i.e. $\zeta_k = \sigma$ and $\zeta_{i-1} \leq \zeta_i$ for $1 < i \leq k$.

[39] Please refer to section B.3.1 in the appendix for an explanation of the *prediction* parameter.

If $k = 1$, no type dispatch has to happen at all, and the code continues as follows:[40]

Once:

⟨*label*⟩_chk1	code for *FS*, σ	failAction = "**jump** target=⟨*label*⟩_nxt"
	⟨*successCode*⟩	
⟨*label*⟩_nxt	**readingLoopNext**	failAction=⟨*failAction*⟩, bodyLabel="⟨*label*⟩_do", predictMoreReadings=true

If $k > 1$, the code continues as follows:

Once:

	typeDispatch	failAction=⟨*failAction*⟩, typeIDs=[*TypeID*(ζ_k), …, *TypeID*(ζ_1)], targets=[⟨*label*⟩_chk⟨*k*⟩, …, ⟨*label*⟩_chk1]

For $1 \leq i \leq k$:

⟨*label*⟩_chk⟨*i*⟩	code for *FS*, ζ_i	failAction = "**jump** target=⟨*label*⟩_nxt"
	⟨*successCode*⟩	

Once:

⟨*label*⟩_nxt	**readingLoopNext**	failAction=⟨*failAction*⟩, bodyLabel="⟨*label*⟩_do", predictMoreReadings=true

5.4.4 Intermediate Code for Transition Labels, Assuming Some Type for the Current Reading

Let n be the number of words in the bit-vector holding the feature values which need to be accessed in order to check the unifiability given the assumption that the current reading of the current token is of type σ. Let now $accessedWord_{\sigma,\,1}$, …, $accessedWord_{\sigma,\,n}$ be the words which need to be accessed, and let $checks_{\sigma,\,1}$, …, $checks_{\sigma,\,n}$ be the checks which have to be performed on these words.[41]

[40] Please refer to section B.3.4 in the appendix for an explanation of the *predictMoreReadings* parameter.

[41] The computation of both multi-dimensional arrays has been described in section 5.2.3.

The straightforward implementation would be to load *accessedWord*$_{\sigma,\,i}$ immediately before executing *checks*$_{\sigma,\,i}$. However, this would cause numerous CPU stalls, because the result of a load instruction is not available at the time when the immediately following instruction is processed. A better implementation schedules independent instructions between a load and the calculations on its result, where this is possible. To achieve this, Patti applies a technique called *Software Pipelining* (cf. [Kacmarcik, 1995], p. 351): the respective load and check parts of two "cycles" are interleaved, as depicted in fig. 5–6.

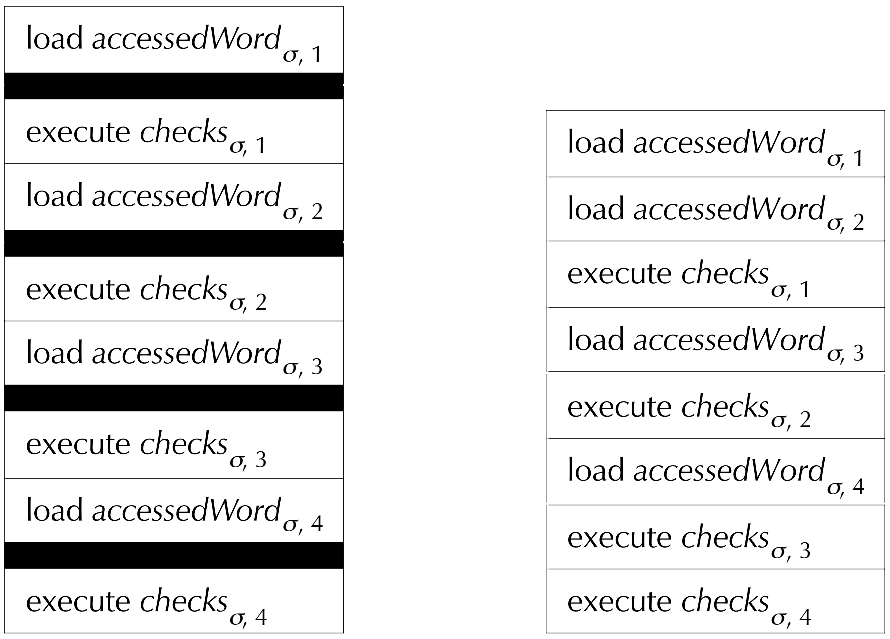

load $accessedWord_{\sigma,1}$
execute $checks_{\sigma,1}$
load $accessedWord_{\sigma,2}$
execute $checks_{\sigma,2}$
load $accessedWord_{\sigma,3}$
execute $checks_{\sigma,3}$
load $accessedWord_{\sigma,4}$
execute $checks_{\sigma,4}$
load $accessedWord_{\sigma,1}$
load $accessedWord_{\sigma,2}$
execute $checks_{\sigma,1}$
load $accessedWord_{\sigma,3}$
execute $checks_{\sigma,2}$
load $accessedWord_{\sigma,4}$
execute $checks_{\sigma,3}$
execute $checks_{\sigma,4}$

Fig. 5–6: *The standard approach would be to load a word from memory into a register, immediately followed by calculations on that register. Due to the relative slowness of the memory subsystem, this however causes stalls on many pipelined CPUs. In the left illustration, the CPU has to wait (thick line) for the first load to finish before the checks on that word can be performed. In the right illustration, the CPU does something useful (loading the second word) while waiting for the first word to be ready.*

The following intermediate code is generated in order to check unifiability with a feature structure of type σ:

Once:

```
loadVectorWord  offset=⟨$accessedWord_{\sigma, 1}$⟩, register=2
```

For $1 < i < n$:

```
loadVectorWord  offset=⟨$accessedWord_{\sigma, i}$⟩,
                register=⟨1 + ($i$ mod 2)⟩
checkWord       failAction=⟨$failAction$⟩,
                register=⟨1 + (($i$ – 1) mod 2)⟩,
                checks=⟨$checks_{\sigma, i-1}$⟩
```

Once:

```
checkWord       failAction=⟨$failAction$⟩,
                register=⟨1 + ($n$ mod 2)⟩,
                checks=⟨$checks_{\sigma, n}$⟩
```

6 Empirical Evaluation

The algorithms presented by this thesis were designed for high efficiency; many provisions have been taken to make the system as fast as possible. Therefore, it would be interesting to know about the actual performance. The subsequent chapter thus describes how the speed was measured and lists the outcome of those experiments.

Unfortunately, there exist no standard benchmarks to evaluate the speed of natural language systems. For this reason, it is hard to give a comparison or a ranking. Nevertheless, there exist some articles which compare different systems; these numbers are repeated as an illustration.

The chapter concludes with a thorough discussion of common objections to the utilized evaluation scenario.

6.1 Measuring the Speed

In the opinion of the author, it makes more sense to evaluate the performance of natural-language engineering systems with an actually utilizable task than with purely hypothetical "toy" examples. To evaluate the speed of Patti, the extraction of simple noun groups was chosen. Example of such noun groups include *tall and small dogs, National Cable TV Association* or *blue velvet*. The corresponding automaton is reproduced in appendix C.[42]

As outlined in section 1.3, the Apple Linguistic Analysis Library sends the input texts to a server over a TCP/IP network. The server performs Part-of-Speech Tagging and returns the tagged text to the client in a text-based representation. The Library then converts the tags into the bit-vector format of Patti. Before subsequent modules are invoked, the Pattern Matching routine (as generated by Patti) is called. From this description, it should be clear that the total time for linguistic analysis — between passing a Unicode text to the library and retrieving the most salient discourse referents — is of no concern for evaluating the Patti system on its own merits. For example, neither the speed of the Lingsoft Part-of-Speech Tagger, the delays of TCP/IP networking nor the Unicode conversion times are of any interest in the context of the present thesis.

Therefore, just the execution time for the routine generated by Patti was measured. This routine, reprinted in appendix C, consists of 144 PowerPC instructions which perform the necessary pattern matching and unification operations. As its input, it takes the bitvector-oriented representation of a single sentence;[43] as its output, it returns the longest ranges[44] of all matching noun groups.

The ***group***, called the ***Information Infrastructure Standards Panel*** (***IISP***), is sponsored by the ***American National Standards Institute*** (***ANSI***) and is open to all ***organizations*** actively working on ***NII*** and ***GII***, both ***members*** and ***non-members*** of ***ANSI***.

Fig. 6–1: *To evaluate the Patti system empirically, the execution time for the automaton of appendix C is measured. In the depicted sentence, the automaton detects eleven noun groups, printed in bold italics.*

[42] At Apple, a slightly more complicated automaton was used to extract the noun groups. However, as pointed out in section 6.4.4, neither the size of the automata nor the number of features do greatly affect the performance. Indeed, the same timing experiments were conducted with the actual automaton as well, with essentially equal results. Therefore, the more illustrative example automaton of appendix C was used for the timing experiments described in this chapter.

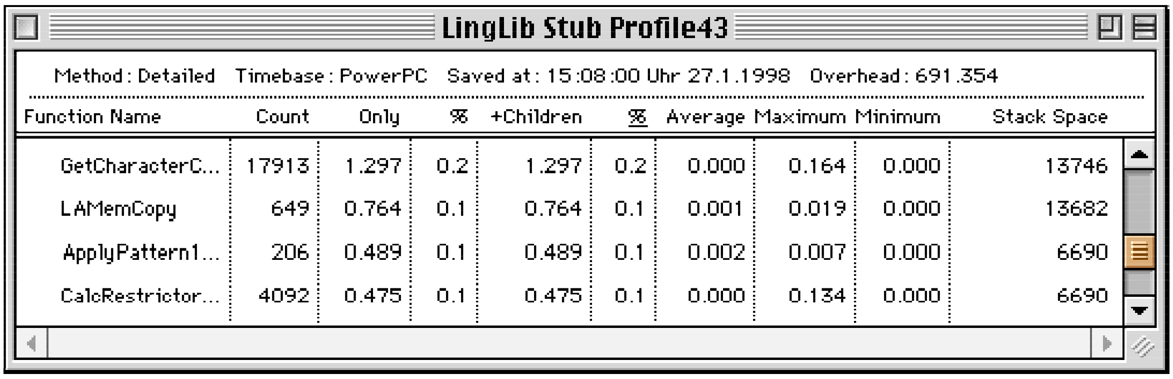
[43] One might object that the time for converting the tagger output into bitvectors is not measured. This and other common objections will be discussed in section 6.4.

[44] In other words, "greedy match" is performed. For example, *National Standards* would be recognized by the automaton as well, but this is not returned because it is fully contained within *American National Standards Institute*. Of course, the time to filter out non-longest matches *is* included with the measured time.

The execution time for this routine was measured with the profiling utility provided by Metrowerks CodeWarrior Professional Release 2. This development environment allows to measure the execution time for each invocation of a routine. A graphical utility, depicted in fig. 6–2, displays for each procedure:

- how often the routine was called;
- the total time for all invocations;
- the minimal, maximal and average time spent.

The timing utilizes the real-time clock facility of the PowerPC chip whose granularity is 128 nanoseconds. The results are displayed in milliseconds, with an accuracy of 1 microsecond.



LingLib Stub Profile43

Method: Detailed   Timebase: PowerPC   Saved at: 15:08:00 Uhr 27.1.1998   Overhead: 691.354

Function Name	Count	Only	%	+Children	%	Average	Maximum	Minimum	Stack Space
GetCharacterC...	17913	1.297	0.2	1.297	0.2	0.000	0.164	0.000	13746
LAMemCopy	649	0.764	0.1	0.764	0.1	0.001	0.019	0.000	13682
ApplyPattern1...	206	0.489	0.1	0.489	0.1	0.002	0.007	0.000	6690
CalcRestrictor...	4092	0.475	0.1	0.475	0.1	0.000	0.134	0.000	6690

Fig. 6–2: *The Metrowerks Profiling Utility displays the execution time for every routine in the Linguistic Analysis Library. For a text which consists of 4092 tokens, Pattern Matching (the third item in the list) was called 206 times (once for each sentence). Finding the noun groups took 0.489 milliseconds in total, 2 µs for the average sentence and 7 µs in maximum, while the minimal matching time was less than 512 ns, which is rounded to 0.000 ms. A maximum stack space of 6690 bytes was needed. These numbers translate to a (though theoretical, see text) matching speed of roughly 8.4 million tokens per second.*

A variety of input texts were used to evaluate the speed:

- **Forbes:** an article from Forbes magazine (cf. [Hutheesing, 1996]) about the strategy of Gilbert Amelio, CEO of Apple Computer at that time, to rescue his company. The article consists of 4092 tokens in 206 sentences.[45]
- **Desktop Printing:** the "About Desktop Printing" file which is put into the "MacOS Read Me Files" folder when installing most Printer Drivers on Macintosh System 7. The text consists of 2894 tokens in 224 sentences.[46]
- **ANSI:** a newswire text ("The American National Standards Institute Hosts the First National Conference of Information Superhighway Panel") from 1994. The text consists of 681 tokens in 22 sentences.[47]

These texts were analyzed with the Linguistic Analysis Library on the following machines:

- an Apple Power Macintosh 6100/60av[48] with a 60 MHz PowerPC 601 processor and 40 MB RAM, running MacOS 8.0 in the US-English version;
- an Apple Power Macintosh 9500/150 with one[49] 150 MHz PowerPC 604 processor and 88 MB RAM, running MacOS 8.0 in the US-English version;
- an Apple Power Macintosh G3[50] with one 334 MHz PowerPC G3 processor and 192 MB RAM, running MacOS 8.1 in the US-English version.

Each text was analyzed three times; fig. 6–3 below indicates the average results of this timing experiment.[51]

[45] The text is available at http://www.coli.uni-sb.de/~brawer/patti/forbes.txt

[46] The text is available at http://www.coli.uni-sb.de/~brawer/patti/desktop.txt

[47] The text is available at http://www.coli.uni-sb.de/~brawer/patti/ansi.txt

[48] This is one of the first released (and hence slowest) Power Macintosh models.

[49] Since the Apple Linguistic Analysis Library is not multi-threaded, the results would not benefit from multiple processors.

[50] As of January 1998, this the the fastest Power Macintosh model (and probably the fastest existing desktop computer at all).

[51] As expected, the results were not differing notably for individual runs on the same machine with the same data. For example, noun group extraction for the Forbes text took 489 μs, 495 μs and 487 μs on the PowerMacintosh 9500/150.

	6100/60av		**9500/150**		**G3**	
	Time	**Tokens/s**	**Time**	**Tokens/s**	**Time**	**Tokens/s**
Forbes	2182 μs	1.9 Mio.	490 μs	8.4 Mio.	192 μs	21.3 Mio.
Desktop Printing	1474 μs	2.0 Mio.	366 μs	7.9 Mio.	152 μs	19.0 Mio.
ANSI	332 μs	2.1 Mio.	101 μs	6.7 Mio.	39 μs	17.5 Mio.

Fig. 6–3 *The execution time for the code generated by Patti was measured on a number of different machines. Depicted are the average time, calculated from three independent runs. The value for "Tokens/s" is the number of tokens in a text divided by the number of seconds needed for extracting its noun groups. This number is however only of theoretical value; cf. section 6.4.3 for discussion of this topic.*

6.2 Interpretation of Results

These results were quite surprising to the author: although it was expected that the numerous optimizations would lead to a very fast system, the performance is an improvement of several orders of magnitude, compared to other systems (cf. section 6.3). This achievement was not prognosticated.

It is hard to type out the reasons for this accomplishment, and it is even harder to quantify them on an individual basis. It has been noted several times in the literature that "research directed towards improving the throughput of unification-based parsing systems [has to be] concerned with the unification operation itself, which can consume up to 90% of parse time" (cf. [Carroll, 1994]). While a number of different methods has been developed to speed up the unification operation, there exists probably no other system which compiles the unifications into machine instructions, scheduled to benefit from advanced computer architecture. Another factor of utmost importance, viz. the opposition of compilation and interpretation, has already been discussed in section 2.2.4.

An additional acceleration is due to the fine-tuned utilization of specialized cache control instructions. If all cache control instructions are removed from the generated code, the system slows down by roughly one third.[52]

It has yet to be explained why the achieved performance is different for the three input texts. However, it is not too surprising that the speed is increasing for longer texts: certain hardware provisions such as instruction caches or dynamic branch prediction buffers are more effective when the same code is executed more often. An additional, probably more important factor might be the initialization overhead which is constant for each sentence — texts with longer sentences are thus analyzed with a better performance.

6.3 Comparison to Other Systems

Unfortunately, it seems that no evaluation criteria are commonly accepted for comparing the speed of systems similar to Patti. At least, [Abney, 1997] enumerates the speeds for some systems which perform partial syntactic analysis:

- traditional chart parsers run at less than 1 token per second[53]
  — Tacitus: ~0.12 tokens/s;
- "skimming" parsers run at 20–50 tokens per second
  — Fastus: 23 tokens/s; Scisor: ~30 tokens/s; Clarit: ~50 tokens/s;
- deterministic parsers can be more than an order of magnitude faster
  — CG: 410 tokens/s; Fidditch: 1200 tokens/s;
  Cass2: 1300–2300 tokens/s; Copsy: ~2700 tokens/s.

Because of the differing hardware used by the respective systems, these numbers are

[52] As an experiment, all dcbt instructions were manually removed from the PowerPC code in appendix C. With this modified routine, extracting noun groups from the "Forbes" text took 661 μs (instead of 490 μs) on the PowerMacintosh 9500/150, which corresponds to a theoretical speed of 6.2 (instead of 8.4) million token per second.

[53] Other sources give better results for Chart Parsers. For instance, [Carroll, 1994] reports analysis speeds of ca. 30 tokens per second.

normalized to a Sun4/Sparcstation 1.[54] Assuming a coëfficient of roughly 10 for an Apple Power Macintosh G3, Patti's speed would outperform the fastest system of the above list by almost three orders of magnitude.

6.4 Common Objections

While discussing the preliminary ideas which finally led to the Patti system, a number of objections came up, most of them related to the evaluation of the system, as described in the current chapter. For this reason, the subsequent sections outline these arguments, and try to respond to the criticism.

6.4.1 "Merely Tuning Constant Factors"

One researcher (of Xerox PARC, Palo Alto, California) objected that the work along the lines presented by this thesis would not constitute a significant benefit to the field: it would merely minimize constant factors, while a major improvement would require the development of entirely new algorithms with better mathematical complexity properties.

In the opinion of the author, this argument is certainly true — from a theoretical point of view. From an engineering perspective, however, it nonetheless *does* make a difference whether the speed of a system is twenty million or barely two tokens per second. The timing experiments indicate that very fast language analysis is indeed feasible, even without a revolution in NLP by developing entirely new algorithms with different mathematical properties.

6.4.2 "Evaluating Caches, not Algorithms"

Another researcher (of DFKI GmbH, Saarbrücken, Germany) pointed out that this kind of timing experiment would not be meaningful, since the most important factor are the CPU and its caches. In other words, he argued that large parts of the efficiency of

[54] Unfortunately, Abney's normalization is not based on a standard benchmark such as SPECint (Abney, personal communication). Even worse, it is not clear whether these numbers comprise only the matching process, or whether Part-of-Speech tagging, conversion, file I/O etc. are counted as well.

the system are not due to the utilized algorithms, but due to the caching mechanisms of the PowerPC chip.

Again, this argument has its point — it is surely true that mechanisms such as superscalarity or a good caching are extremely influential on the results. On the other hand, the whole purpose of the Patti system is to utilize these mechanisms offered by modern hardware. Issuing cache control instructions, for instance, is something which could be done with other algorithms as well — but Patti seems to be the first NLP system actually doing so, with a considerable achievement in performance.

6.4.3 "I/O not Counted"

Yet another researcher (of the Department for Computational Linguistics, Saarbrücken, Germany) was objecting that only the time to find the noun group ranges is included in the timing experiments above. His desire was to know about the performance of the overall system, for example including disk I/O, instead of just one single component.

This objection is certainly valid from the perspective of a person who has to compare different system *architectures,* or who has to decide which system to buy. However, the author thinks these factors not being relevant in the context of the present thesis, whose topic is indeed a specialized component and not an overall system architecture. Including the time for disk I/O is, at least in the opinion of the author, highly problematic to evaluate the efficiency of a matching or unification algorithm. For example, it might very well be that a system does not involve any disk operations at all: it might be part of an Embedded System (where there is no hard disk), or it might not need to access the disk.[55]

However, it must be concluded that with a performance as high as achieved by Patti, other factors (such as disk I/O related to Virtual Memory) become more important. In a system that takes 20 minutes to analyze a two-page text, 140 milliseconds for disk access are neglectible. In a system that takes two seconds, the 140 milliseconds are more important in comparison. In a system that takes 500 microseconds, factors such as virtual memory are an utterly prominent factor. A "matching speed of 8 million tokens per second" is thus mostly of illustrative value; this number would only be true if pattern matching and unification were the *only* operations that happen in a system. If the 8 million tokens do not fit into memory and hence need to be paged in and out, it

[55] The Apple Data Detectors/Internet Address Detectors project is an instance of a pattern matching system which does not involve any disk I/O.

will take more than one second to analyze them, because of the interference with the Operating System.

Nonetheless, it can be stated that advanced compilation technology, of which Patti constitutes an instance, can speed up natural-language processing tremendously, so that this task becomes an entirely neglectible factor in the overall system performance.

6.4.4 "Conversion into Bit-Vectors not Counted"

A seemingly straightforward objection is that the time to convert the part-of-speech tags into the bit-vectors is not included in the above numbers.

The rationale for not comprising this conversion time[56] is that this step is essentially superfluous. The Apple Linguistic Analysis Library was designed to eventually include a proprietary PoS tagger, instead of utilizing an external one. There is no reason why this tagger could not directly emit the bit-vectors in the format required by Patti, instead of converting internal data structures into a text-based representation and vice versa.

6.4.5 "What about Larger Grammars?"

Several persons pointed out that the described timing experiment was done with a very small grammar. The question is the one about scalability: how would the system behave if presented with a substantially larger grammar?

There are different parts of the grammar to separate when discussing this topic:

- A larger number of *states* in the finite-state automaton is unlikely to affect the performance of the system to a significant extent. The time needed for traversing a finite-state automaton does not depend on the number of its states. A very slight degradation of performance could however occur since a larger number of states leads to more machine code. Larger code sections are not loaded at once into the Instruction Cache of the CPU, which is why a very short delay could occur on jumping from one state to another, if there exists a large number of states.

[56] On a PowerMacintosh 9500/150, the required conversion time is 0.2 s for the "ANSI" example text, 0.8 s for "Desktop Printing" and 1.2 s for "Forbes."

- A larger number of *types* can only affect performance if more than just a small subset of all types is actually used in the grammar. This was not true for the automata at Apple. However, if a large number of types *is* present at run-time, this possibly could lead to some degradation of performance. The predictions of the Unification Oracle (cf. section 5.3) would be wrong more often, so that the unification operation could be omitted in less cases.
- A larger number of *introduced features* is not expected to slow down the processing, since only those features which are specified in the transition label AVMs are compiled to code. Although more introduced features lead to larger memory needs, this is very unlikely to affect performance, because of the careful usage of cache control instructions.
- A larger number of *features in the transition label AVMs* potentially could alter the performance of the system for similar reasons as stated above, since the size of the generated code depends on the number of features in the automata. However, that code is only executed if the Unification Oracle predicts a match. Therefore, only minor degradation of speed is expected.
- The greatest concern about scalability is probably with a larger number of *automata:* Since each automaton runs independently from the others on the entire sentence, the author expects that the processing time will increase proportionally with the number of finite-state automata. A possible solution is to combine several automata into a single one (see the extension proposed in section 7.1.2). It might however be that the problem does not turn out to be severe, since a typical grammar probably does not consist of hundreds or thousands of automata.

6.4.6 "What about Memory Requirements?"

Some readers might miss information about memory requirements from the numbers given above.

The reason for this is related to the architecture of the Apple Linguistic Analysis Library: In most applications, there is no need to preserve all detected information until all text has been processed. For example, the information about gender or part-of-speech is not needed after a certain stage in processing; this allows to minimize memory needs. In addition, most linguistic modules do not have to operate on the entire text at once — with a carefully designed system architecture, it is possible to analyze a smaller chunk of a text (e.g. a paragraph) on its own. Then, only relatively few informations must be preserved; most memory can be released immediately after being done with the current chunk.

Therefore, it does not make too much sense to compare (partial) parsing systems with respect to their memory needs — this total memory requirement is almost meaningless with a well-designed overall system architecture.

There is, however, a mode in which the Apple Linguistic Analysis Library preserves all detected information. This is utilized for the debugging viewer which was mentioned in section 1.3. The total memory needs (including many data structures which normally would be released at an early processing stage) is 55.7 Kilobytes for the "ANSI" text, 191.0 Kilobytes for "Desktop Printing" and 279.5 Kilobytes for "Forbes."

7 Future Extensions

The subsequent chapter discusses some possibilities to improve the Patti system. However, certain extensions not related to Patti are considered more urgent. For instance, the inclusion of an efficient Part-of-Speech tagger into the Linguistic Analysis Library would allow to distribute the library as a stand-alone module that does not depend on any external parts to provide interesting functionality.

7.1 Extensions to the Formalism

It would be possible to extend the input formalism in several directions. However, one has to keep in mind that the primary aim of the Patti system is to build an extremely fast processing engine that is well suited for Natural Language *Engineering* purposes. It is not intended to provide a full framework which would be able to process HPSG-style grammars or similar high-level linguistic formalisms. Nevertheless, certain extensions could be included without greatly affecting efficiency.

7.1.1 Coreferences between AVMs

There is currently no way to model agreement phenomena, although this would be very useful, e.g. for detecting German noun phrases. The formalism could be enhanced to allow coreferences between features, even in different transition labels. This extension would be comparable to the notion of *registers* in Augmented Transition Networks.

Indeed, coreference of those atomic-typed features whose representation is a bit-vector could be efficiently implemented along the following lines:

- Depending on the size of the bit-vector, one or several integer registers hold the bit-vector corresponding to the unification result.
- These registers are initialized with all-1 vectors before entering the start state.
- For each occurrence of the same coreference index, the compiler generates code which performs a bit-AND operation of the register(s) with the feature value in the run-time representation. An all-zero result means unification failure.
- In case a value is specified in addition to the coreference, this leads to another bit-AND operation (with an immediate value). Again, zero means unification failure.
- The contents of the involved registers are saved on the non-determinism stack.
- To reduce the number of needed registers, an established method from compiler construction (register allocation by graph coloring, cf. [Wilhelm/ Maurer, 1992], p.561) might be adopted.

The run-time algorithm though gets more complicated with coreferences between matrices, and with coreferences between atomic-typed features whose representation is an integer. It is not clear how an efficient implementation would look like.

7.1.2 Output for Accepting States

Currently, the automata act as acceptors: their only output is the range of their longest match in the input string. It might be a useful enhancement to allow automata to build AVMs as side-effect of being in an accepting state. Associating an AVM with the entire automaton would be easy to implement, but an individual AVM for each accepting state would probably be more useful.

7.1.3 Cascades of Finite-State Automata

As has been pointed out in section 2.1, *cascaded* finite-state automata form a very promising approach to (partial) parsing. With output for accepting states, as described above in section 7.1.2, the formalism could easily be enhanced to allow for cascades. The necessary modifications to the compiler seem to be very straightforward and easy to implement.

7.1.4 Weakening Appropriateness Conditions

The appropriateness conditions of section 3.1.4 could be weakened to allow for non-finitely typable feature structures. The implementation could be realized along the lines of the AMALIA system. However, it is not clear whether this extension is actually needed for the linguistic engineering tasks covered by Patti. For instance, there seems to be no Part-of-Speech tagger which would emit cyclic feature structures.

7.1.5 Multiple Inheritance

Most parts of the compiler were designed with Simple Inheritance type hierarchies in mind. Nonetheless, an extension to Multiple Inheritance is feasible, since the current run-time unification is not necessarily restricted to Single Inheritance. However, this would constitute a major effort involving broad changes to the compiler code. In addition, it is not clear if Multiple Inheritance hierarchies are actually needed at all for the (engineering) applications of the Patti system.

7.1.6 Finite-State Approximation of Context-Free Grammars

One of the algorithms which were outlined in section 2.1.4 could be incorporated with the Patti system to allow for context-free grammars as input formalism. It might be a very interesting alternative to design yet another, new approximation algorithm which would approximate the input grammar with a *cascade* of finite-state automata. A finite number of recursive rule invocations would correspond to different cascade levels, while additional invocations would be covered by adapting one of the conventional techniques mentioned above.

7.2 Improving Code Generation

Although a large effort was made to implement good instruction scheduling, the optimal sequence is not emitted in every case. Certainly, the run-time performance of the system could benefit from improving the code generation module. However, this is certainly not the most important extension, given the already very high speed which is achieved now.

7.3 Support for Additional Platforms

The system in its current state is only functional for PowerPC machines running MacOS.[57] For greater usability, the system should be ported to additional platforms. The subsequent discussion has only to cope with the output generated by the Code Generation module, because the compiler itself has been developed in 100% Pure Java. Portability of the compiler itself is therefore granted.

7.3.1 Porting to Other Operating Systems

The Pattern Matcher does not use any functions provided by the Operating System. Since the generated code adheres to the PowerPC Application Binary Interface (cf. [IBM/Motorola, 1997]), it is supposed to work under all Operating Systems supported for PowerPC (MacOS, A/IX, Linux, Mach/OpenStep/"Rhapsody", Windows NT, BeOS and several embedded systems). However, it has only been tested under MacOS.

The rest of the Apple Linguistic Analysis Library is written in ANSI C. The only OS functionality needed are Memory Management and Networking; the latter is coded to the POSIX Open Transport libraries. Finally, all text is represented in Unicode. Very easy portability is hence granted.

7.3.2 Assembly Language for Other CPUs

To support additional processors, only the code generation module has to be enhanced. The compilation algorithm (cf. chapter 4) is machine-independent, and the

[57] Of course, any OS with MacOS emulation (BeOS, Rhapsody) is acceptable as well.

intermediate language (cf. appendix B) has been designed with portability in mind. Thus, it is assumed that the amount of work for porting the compiler to another CPU is rather limited.

The efficiency of Assembly code for different CPUs should be comparable, provided these share their most important characteristics with PowerPC, being superscalar pipelined RISC machines. Because typical CISC processors allow less fine-grained instruction scheduling, a slight degradation of performance is predicted there. However, these are merely speculations: to determine the actual speed on another CPU, the code generator will have to be ported.

7.3.3 ANSI C as Code Generation Output?

Instead of porting the code generation to a number of different instruction sets, one could argue that it might make sense to port it once to a language which is available on virtually every machine, for example ANSI C. Of course, this would solve the problem of portability, but a rather large degradation of performance is expected, even when using highly optimizing C compilers.

The reason for this has been pointed out several times in the present thesis: A compiler which operates on a high-level formalism has a more explicit knowledge of the problem, thus it can base its decisions on more accurate heuristics. Two concrete examples might serve as an illustration thereof:

- Static Branch Prediction (cf. section A.2.3) is a means for the compiler to inform the CPU whether or not a particular branch is likely to be taken. Patti knows in many cases about the branch probability and can express this knowledge in the generated Assembly code, whereas a C compiler is restrained to extremely simple heuristics. For example, data can only be taken from a stack if the stack is not empty. Patti's heuristic ("automata written by linguists are mostly deterministic, thus the stack is probably empty") is better than that of a C compiler ("forward branches are likely to fail")[58]. In this particular case, equivalent C code will lead to a mis-

[58] Metrowerks CodeWarrior Professional Release 2 for PowerPC on its highest optimization level seems to use this heuristics, which is a reasonable one for the general case (cf. [Hennessy/Patterson, 1994], section 3.5).

predicted branch and cause a delay for several clock cycles whenever the stack is checked for emptiness.[59]

- Cache instructions (cf. section A.4.4) are a means for the compiler to inform the CPU that a specific memory address will be accessed soon. The memory subsystem can then initiate a transfer from memory into the cache while useful calculations are executed in parallel. When the address is later actually accessed, the processor does not need to wait for the data to arrive. It would be rather difficult for a C compiler to emit this instruction.[60] Experiments indicate a performance loss of roughly one third if these cache control constructions are omitted from the generated code (cf. section 6.2).

For these reasons, generating C code would loose many of the advantages with regards to efficiency. As in the previous section, however, a definitive decision can not be made on merely theoretical grounds. To determine how much efficiency is lost when producing C code as output of the system, the code generator will have to be ported.

7.4 Completion of Graphical User Interface

Certainly the most urgent extension to the Patti system is in the area of the specification of the input grammars. As pointed out in section 3.2, the author believes for several reasons that a text-based input language would not be a promising path. Instead, the direct-manipulative Graphical User Interface should be enhanced to cover all parts of the input grammar formalism.

[59] This does not entirely hold on processors with *Dynamic Branch Prediction* (e.g. PowerPC 604, but not 601) where the initial compiler-specified branch probablility is corrected depending on the actual branch behavior. Nevertheless, the static prediction is still important for the first time a branch is encountered, or if a branch has been flushed from the Branch Prediction Buffer.

[60] *Metrowerks CodeWarrior Professional Relase 2 for PowerPC* does not seem to generate cache control instructions, even on the highest optimization level.

A PowerPC Instruction Set

A certain awareness of the basic principles of PowerPC Assembly Language Programming is required in order to understand both the code generation module of the Patti compiler (as described in Appendix B) and the example output (as described in Appendix C). Since it can't be presumed that the typical reader of the present thesis is familiar with those topics, the subsequent chapter will give a very brief introduction to some basic concepts before tuming to that subset of PowerPC instructions which is actually emitted by Patti.

However, it is neither possible nor intended for this thesis to form a textbook on Assembly Language Programming or to cope in general with PowerPC optimization issues. For the former, the reader might want to consult a Computer Architecture textbook such as [Hennessy/Patterson, 1994]. For the latter, [Kacmarcik, 1995] gives an excellent introduction. The PowerPC instruction set is described in [IBM/Motorola, 1995] and [IBM/Motorola, 1997]. For each model in the PowerPC series, a separate volume has been published to describe its peculiarities (e.g. [IBM/Motorola, 1993], [IBM/Motorola, 1994]). A thorough discussion of efficiency-oriented PowerPC Assembly coding, seen from the viewpoint of the compiler writer, can be found in [Hoxey et al., 1996].

A.1 RISC vs. CISC

The PowerPC instruction set adheres to the general principles of *Reduced Instruction Set Computing* (RISC). Other examples for RISC architectures include the Intel 860, MIPS R3000, Motorola M88000 and the SPARC chips, whereas e.g. the Intel

80x86/Pentium and Motorola 680x0 series constitute instances of the more traditional *Complex Instruction Set Computing* (CISC) architecture.

The following is a list of features that are commonly associated with RISC architectures:

- a large uniform register set;
- a load/store architecture: A small, distinguished set of instructions transfers data from memory into the registers and back; all calculations operate on registers exclusively. In contrast, a typical CISC instruction set contains numerous instructions that directly perform calculations on memory;
- a minimal number of addressing modes;
- a simple fixed-length instruction encoding — CISC instructions can be of variable length;
- only minimal support for misaligned memory accesses.

This set of "rules" is designed to make fast processors easier to implement. Note that this does not necessarily mean that RISC technology would *always* be faster and cheaper than CISC. Certain efficiency-increasing mechanisms such as instruction pipelines, a superscalar instruction dispatch and hardwired instructions are typical for RISC microprocessors, but are sometimes applied with CISC instruction sets as well.

A.2 Processing Machine Instructions

As has been mentioned at the beginning of the current chapter, any reasonable introduction to the concepts of Computer Architecture would exceed the scope of this thesis. However, certain mechanisms are especially important to understand when dealing with (RISC) Assembly code: Pipelining, Superscalar Instruction Dispatch and Branch Prediction. Therefore, the subsequent section briefly explains these methods that are used for reducing the execution time on the PowerPC, and on most other current CPUs as well. The Patti Code Generation is designed to exploit these mechanisms to a large extent, which constitutes a major reason for the efficiency of the described system.

A.2.1 Instruction Pipeline

The execution of every machine instruction takes some time, typically several clock cycles. By splitting the processing into a pipeline consisting of several stages, each stage can work on a different instruction at the same time. Thus, the overall processing time is reduced.

For instance, the master instruction pipeline of the PowerPC 604 has six stages. All instructions executed by the machine flow through these stages, although some instructions combine several stages into a single cycle, while others flow through additional execution pipeline stages.

A.2.2 Superscalar Instruction Dispatch

A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same pipeline stage at the same time. For example, the PowerPC 604 contains two separate Integer Units.[61] Under certain circumstances, the CPU's Instruction Dispatch Unit will dispatch two independent integer instructions to these two units. Hence, those two instructions are actually executed in parallel, although they appear sequentially in the program.

Some highly optimizing compilers for high-level languages consider these instruction scheduling issues, although many others do not. The code generation module of Patti is designed to utilize this parallelism, for instance to execute certain unifications in parallel. For each individual Intermediate Code instruction, the corresponding PowerPC machine code has been hand-optimized with regards to Superscalar Instruction Dispatch. In addition, the design of the Intermediate Code (cf. appendix B) has been influenced by the desire for taking advantage of pipelining and parallelism.

A.2.3 Branch Prediction

Branch Prediction is a mechanism for the processor to guess whether or not a particular branch will be taken. The ability to generate some type of reasonable prediction is quite useful since a pipelined processor is usually not able to completely resolve a

[61] In fact, there exist three Integer Units in the PowerPC 604, but two of them cope with instructions that can be executed in a single clock cycle, while the third is responsible for multiple-cycle instructions.

branch before it needs to be executed. Thus, the processor *speculatively* executes the instructions along a predicted path until the branch is later resolved. If the prediction was correct, the processor can continue the execution; otherwise, a backtracking process takes place.[62]

Static Branch Prediction allows the Assembly language programmer (or the compiler) to express their assumptions about the more probable behavior. For example, it is necessary to test if a stack is full before pushing additional data onto it — but in most cases, this test will fail. The Patti compiler expresses this knowledge by setting an appropriate prediction flag in the generated branch instruction. Then, the processor knows that it is not probable for this branch to be taken, and will therefore speculatively execute the instruction path for the non-overflowing case.

As has been mentioned in section 2.2.4, there is no means to express this kind of knowledge in high-level languages. In comparison to a C compiler, Patti's branch predictions will typically be closer to the actual behavior, since they can be based on more appropriate heuristics.

A.3 Register Set

The PowerPC architecture defines 32 general-purpose registers, called r0 – r31. The PowerPC Run-Time Environment [IBM/Motorola, 1997] assigns special meanings to r1 and r2; function parameters are passed in r3 up to r15. The architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode.

Eight condition register fields (cr0 – cr7) hold conditions that can be used for conditional branching. Arithmetic, Boolean and Compare instructions set the value of a specified condition register field; cr0 can be set implicitly as a side-effect of an integer instruction. Conditional branches test a specified condition register field and branch if the respective condition is fulfilled.

[62] Actually, speculative execution works on *shadow registers,* i.e. copies of the real registers. When it is known that the branch was predicted correctly, the real registers are overwritten with their shadow values, and the execution continues. In case the branch was mispredicted, backtracking involves no further actions than setting the Program Counter, because the original registers were never changed during speculative execution.

The link register (LR) is used for branching to subroutines: Branch instructions include the option of placing the address of the instruction following the branch instruction in the LR. Other branch instructions set the program counter to the content of the LR (i.e. return to caller).

There exist numerous other registers (floating-point registers, count register, time base facility, etc.), none of which are used by the code currently generated by Patti.

A.4 Instructions

Below, only those instructions actually emitted by the current Patti code generation module are discussed, and only with those options that are used by Patti. For the full instruction set, see [IBM/Motorola, 1995]. For instruction latencies and other concerns with regards to scheduling, see [IBM/Motorola, 1993] and [IBM/Motorola, 1994].

A.4.1 Load and Store

lhz r_T, d(r_A)	Loads two bytes from memory into the lower 16 bits of r_T, starting at address (r_A) + d. The other bits of r_T are set to 0.
lwz r_T, d(r_A)	Loads four bytes from memory into the lower 32 bits of r_T, starting at address (r_A) + d. The other bits of r_T are set to 0.
stw r_T, d(r_A)	Stores the lower 32 bit of register r_T into four bytes in memory, starting at address (r_A) + d.

A.4.2 Integer Calculations

addi r_T, r_S, V	The contents of r_S are added to V, and the result is placed into r_T.
andi. r_T, r_S, V	The contents of r_S are ANDed with 0x000000000000 ‖ V, and the result is placed into r_T. The result is compared to zero, and the condition register field 0 is affected correspondingly.[63]
andis. r_T, r_S, V	The contents of r_S are ANDed with 0x00000000 ‖ V ‖ 0x0000, and the result is placed into r_T. The result is compared to zero, and the condition register field 0 is affected correspondingly.

[63] The ‖ symbol is a common notation for concatenating bit sequences.

cmplwi cr, r_S, V	The contents of r_S are compared with the 16-bit *unsigned* constant V. The condition register field cr is affected, depending on the result.[64]
cmpwi cr, r_S, V	The contents of r_S are compared with the 16-bit *signed* constant V. The condition register field cr is affected, depending on the result.[65]
li r_T, V	Loads the 16-bit signed immediate value V into r_T.[66]
mr r_T, r_S	Moves the contents of r_S into r_T.[67]
subi r_T, r_S, V	V is subtracted from the contents of r_S, and the result is placed into r_T.
xori. r_T, r_S, V	The contents of r_S are XORed with 0x000000000000 ‖ V, and the result is placed into r_T. The result is compared to zero, and the condition register field 0 is affected correspondingly.
xoris. r_T, r_S, V	The contents of r_S are XORed with 0x00000000 ‖ V ‖ 0x0000, and the result is placed into r_T. The result is compared to zero, and condition register field 0 is affected correspondingly.

A.4.3 Branch Instructions

Some of the instructions below have a suffix which is either + or –. This serves to express the Static Branch Prediction (cf. section A.2.3 above). Plus stands for "predict branch to be taken," minus stands for "predict branch not to be taken."

b label	Jumps unconditionally to the instruction indicated by label.
beq± cr, label	Branches to the instruction indicated by label if the condition register field cr has a certain value.[68] The branch is taken if the outcome of a compare instruction was "is equal", or if a Boolean instruction had an all-zero result, or if an arithmetic calculation lead to zero.
beqlr± cr	Similar to **beq±**, but jumps to the address indicated by the Link Register instead of a fixed location. This can be used to return to the caller of a subroutine if certain conditions are fulfilled.

[64] The Assembler assumes cr0 if the cr parameter is missing.

[65] The Assembler assumes cr0 if the cr parameter is missing.

[66] Actually, this is translated by the Assembler to **addi** r_T, 0, V.

[67] Actually, this is translated by the Assembler to **or** r_T, r_S, r_S.

[68] The Assembler assumes cr0 if the cr parameter is missing.

bl label	Sets the Link Register to the address of the immediately following instruction, before jumping unconditionally to the instruction indicated by label. This can be used to call a subroutine; the return address will be in the Link Register.
blr	Jumps unconditionally to the instruction whose address is in the Link Register. This can be used to return to the caller of a subroutine.
bne± cr, label	Branches to the instruction indicated by label, if the condition register field cr has a certain value.[69] The branch is taken if the outcome of a compare instruction was "is not equal", or if a Boolean instruction had an non-zero result, or if an arithmetic calculation lead to something else than zero.
bnelr± cr	Similar to **bne±**, but jumps to the address indicated by the Link Register instead of a fixed instruction. This can be used to return to the caller of a subroutine if certain conditions are fulfilled.

A.4.4 Cache Control

dcbt 0, r_A	Transfer memory addressed by r_A into data cache.
dcbt r_A, r_B	Transfer memory addressed by $(r_A + r_B)$ into data cache.

The **dcbt** is a hint that performance will possibly be improved if the block containing the byte addressed by r_A (or $r_A + r_B$) is fetched into the data cache, because the program will probably soon load from the addressed byte. The time between the **dcbt** instruction and the actual load can be used by the memory subsystem to transfer the data into the cache while other instructions are executed in parallel.

A.4.5 Miscellaneous

mflr r_T	Move contents of Link Register into register r_T
mtlr r_S	Move contents of register r_T into Link Register

[69] The Assembler assumes cr0 if the cr parameter is missing.

B Intermediate Code

This appendix lists all elements of the intermediate representation language. The Intermediate Code is actually implemented as a set of Java classes, with member methods to perform the code generation for specific target platforms. Currently, two platforms are supported: PowerPC and, for debugging purposes, an "intermediate" platform whose instruction set is just a text-based representation of the instructions described in this chapter. This was used to generate the listing of section C.4.

B.1 Miscellaneous

The following instructions are intended for interfacing with the Assembler and various initialization and clean-up tasks.

B.1.1 declaration *automatonID*

This is emitted as very first instruction for an automaton whose numeric ID (an integer) is *automatonID*. The PowerPC code generation translates this to a C "asm" function declaration, followed by an opening curly bracket. This tells the C compiler that the subsequent lines are in-lined Assembly code.

B.1.2 declarationEnd

This is emitted as very last instruction for an automaton. The PowerPC code generation translates this into a closing curly bracket, which indicates to the C compiler that the end of in-lined Assembly code has been reached.

B.1.3 prolog *startStateAccepting*

The compiler emits this instruction, intended for general initialization upon entry, immediately after **declaration**. On PowerPC, the contents of non-volatile registers are saved on the stack, as required by the PowerPC Application Binary Interface, and a number of variables are set to their initial value. The PowerPC instruction sequence has been optimized for maximal throughput.[70]

Since the initialization of variables might be different if the start state of the automaton is accepting, this is passed as parameter.

B.1.4 epilog

The compiler emits this instruction, intended for general clean-up tasks upon exit, immediately before **declarationEnd**. On PowerPC, the contents of non-volatile registers are restored to their original values. Again, the instruction sequence has been optimized to minimize CPU stalls.

B.1.5 stateEntry *accepting*

This instruction is intended for general initialization tasks upon entering a new state. The compiler emits it as first instruction for every state. On PowerPC, the pointer to the current token (r8) is incremented, and the information needed by the Unification Oracle (cf. section 5.3) is loaded into register r9.[71]

Because the initialization of variables is different if the entered state is accepting, this information is passed as parameter.

[70] A `mflr` instruction (used to load the contents of the Link Register into a General-Purpose Register) causes a delay of one cycle in the Execute Stage if the next instruction is dependent. Therefore, independent instructions are scheduled between `mflr` and the storage of its contents on the stack.

[71] Actually, r9 is loaded *before* incrementing r8 by the size of an array element, but with an additional constant displacement of just this size. This avoids a one-cycle stall that would occur in the code for the immediately following **oracle** instruction.

B.2 Branches

The instructions discussed in this section control the program flow. In contrast to all other types of instructions of the Intermediate Code, branches can be passed as parameters to other instructions. For example, the *checkWord* instruction (cf. section B.3.7) takes a branch as *failAction* parameter which is executed whenever a check fails.

B.2.1 jump *target*

Jumps to the label *target*. For PowerPC, the single instruction **b** *target* is emitted.

B.2.2 call *target*

Calls the subroutine which starts at the label *target*. For PowerPC, the single instruction **bl** *target* is emitted.

B.2.3 return

Returns from a subroutine to its caller. For PowerPC, the single instruction **blr** is emitted.

B.3 Unification Checks

The instructions described in this section check if a transition in the finite-state automaton can be taken — they determine if the label of a transition is unifiable with the current token. Most of these checks take a *failAction* parameter: this is a branch which is to be executed if one of the checks fail. Depending on the context, the compiler passes either a **jump** or a **return** instruction for *failAction*.

If the check succeeds, the program flow continues without branching. The compiler hence emits a sequence of unification check instructions for a transition label, followed by those instructions to be executed upon unification success.

B.3.1 oracle *oracleWord, prediction, failAction*

The compiler emits the **oracle** instruction as first member of the code sequence which is generated for a transition label. Its purpose is to implement the Unification Oracle, as described in section 5.3.

The bit-vector which was loaded upon entering the current state is ANDed with *oracleWord*. If the result is an all-zero vector, it is sure that the unification will fail due to inconsistent types. In this case, the branch *failAction* is executed.

With the *prediction* parameter, the compiler can indicate its speculation about the outcome. The PowerPC code generation uses this for Static Branch Prediction (cf. section A.2.3). However, the current version of the compiler has no reasonable heuristics about what the oracle will predict and always assumes that the oracle will detect unification success.[72]

B.3.2 readingLoopInit *firstVectorWordAccessed*

The **readingLoopInit** instruction is intended as an opportunity to initialize variables before looping over all readings of the current token. The compiler emits it immediately before the **readingLoopBody** instruction.

The only parameter, *firstVectorWordAccessed*, indicates which word of the bit-vector will first be accessed in the subsequent unification checks. Some CPUs allow for specialized machine instructions to inform the memory subsystem that a specific memory location will be accessed soon in the future. This initiates then a transfer from the RAM into the data cache, so that the data is already in the cache when it is actually needed.

On PowerPC, register r10 is loaded with a pointer to the first reading of the current token. In addition, register r14 is loaded with a displacement that corresponds to *firstVectorWordAccessed*.

[72] Unification failure corresponds to the path where the branch is taken. In the absence of any other heuristics, it is better to predict a conditional PowerPC branch not to be taken, because the instructions following the branch are already dispatched at the time of speculative branch execution.

B.3.3 readingLoopBody *typeName, failAction, firstVectorAccessed, firstUnif, lastUnif*

The **readingLoopBody** instruction performs a number of tasks at the same time. It would have been possible to split it into a number of smaller instructions, each one performing a more specialized task. However, the Intermediate Instructions are the Basic Blocks for the code generation: individual machine instructions can only be moved within the boundaries of a Basic Block. Since instruction scheduling is a crucial factor for high-performance Assembly-language coding, efficiency of the final code was preferred to better readability.

The compiler emits this instruction immediately after **readingLoopInit**; it is reached every time before a reading is tried to unify with that feature structure which labels a transition in the finite-state automaton.

The *typeName* parameter is a Unicode string for the type of the compiled feature structure. It is useful to generate neat comments, but has no actual purpose besides debugging.

As with most other unification check instructions, *failAction* indicates what should happen if the unification fails. The compiler will always pass a **jump** to the corresponding **readingLoopNext** instruction (cf. B.3.4), since it might still be possible that the next reading matches.

firstVectorAccessed has the same value and purpose as with **readingLoopInit**; cf. B.3.2 for a description.

Due to the ID assignment algorithm described in section 5.1.3, all types greater than or equal *firstUnif* and less than or equal *lastUnif* are unifiable with the type of the compiled AVM. These two numbers, determined for each type upon compilation of the type hierarchy, are passed as arguments. The machine instructions generated for **readingLoopBody** compare the type ID of the current reading with *firstUnif* and *lastUnif;* if the type ID of the current reading falls outside this range, *failAction* is executed.

A minor optimization can be performed if *lastUnif* is the same number as assigned to the type with the greatest number of all types. In this special case, it is not possible that the current reading has a type ID which is greater than *lastUnif,* because there does not exist such a type. Therefore, this check will never fail and thus can be safely omitted. To indicate this special situation, the compiler passes 0 for *lastUnif* as a flag for the code generation to omit the corresponding checks.[73]

[73] The least assigned type ID is 1, which is why 0 can't be the ID of any existing type.

A very similar situation occurs with *firstUnif* being equal to the least type ID; again, a zero value for *firstUnif* serves to indicate this situation.

The PowerPC code generation tries to achieve high performance by a highly optimized scheduling of the emitted machine instructions. The instruction order has been tuned to minimize stalls. In addition, the **dcbt** instruction (cf. A.4.4) is emitted to ensure that the subsequent reading is transferred from RAM into the faster CPU cache, while the current reading is processed in parallel.

B.3.4 readingLoopNext *failAction, bodyLabel, predictMoreReadings*

The **readingLoopNext** instruction is the branch target for unsuccessful unification checks. Its purpose is to prepare a number of variables for the next loop before jumping to **readingLoopBody** again. The label of the **readingLoopBody** instruction is passed as *bodyLabel* parameter. If all readings of the current token have been processed without success, *failAction* is executed.

The compiler can pass its speculation about the likelihood of additional readings in the *predictMoreReadings* parameter. The PowerPC code generation utilizes both Static Branch Prediction and a specialized instruction scheduling to optimize for the case indicated by *predictMoreReadings*. However, the compiler currently lacks a reasonable heuristics, which is why *predictMoreReadings* is always set to true.[74]

B.3.5 typeDispatch *failAction, typeIDs, targets*

Depending on the type of the current reading, certain unification checks have to be omitted.[75] In accordance to the various situations that can occur, the **typeDispatch** instruction jumps to a target depending on the ID of the current reading. The *targets*

[74] The compiler could "know" that certain types (e.g. nouns and verbs) are likely to be part of ambiguous readings, while others (e.g. determiners) are not. It would probably make sense to set *predictMoreReadings* to false in the latter case.

[75] If the type of the current reading is more special than the compiled AVM, the compiled AVM might specify features which are not appropriate for the current reading. While in standard frameworks, those feature values would have to be copied into the unification result, those unifications succeed by definition with Patti. For this reason, the checks for the features in question must be omitted. Cf. sections 5.2.3 and 5.4.3 for details.

parameter specifies which label to jump to when the type of current reading is in *type-IDs*.

B.3.6 loadVectorWord *offset, register*

The **loadVectorWord** instruction loads a machine register with a 32-bit word (specified by the *offset* parameter) from the bit-vector which is part of the data structure of the current reading.

It is certainly not the compiler's job to determine which machine registers will actually be used to hold calculation results, since the register files of the target platforms might substantially differ. However, to exploit the parallelism enabled by superscalarity (cf. section A.2.2), the existence of two different "logical" 32-bit registers is assumed. These registers, which are part of the abstract machine whose instruction set is described in the current chapter, are called "1" and "2."

On PowerPC, the general-purpose registers r11 and r12 are used for "1" and "2," respectively. A single **lwz** instruction is emitted to load the corresponding vector word into the register.

B.3.7 checkWord *failAction, register, checks*

Upon execution of **checkWord**, a part of the bit-vector of the current reading is assumed to be already present in the logical register passed as parameter *register.* The checks indicated by the array *checks* are performed on this register. As soon as any of these checks fails, *failAction* is executed.

A number of different types of objects can be passed in the *checks* array:

- AND checks perform a bit-AND of a 32-bit argument with the value of *register.* The check fails if the result is an all-zero vector.
- EQ checks first mask out certain bits of the value of *register*. If the result is equal to a 32-bit argument, the check succeeds and nothing happens. If the result however is not equal, *failAction* is executed.
- NEQ checks first mask out certain bits of the value of *register*. If the result is not equal to a 32-bit argument, the check succeeds and nothing happens. If the result however is equal, *failAction* is executed. NEQ is hence the opposite to EQ.

- CNF checks allow to specify number of sets of disjoined EQ and NEQ checks; *failAction* is executed unless one disjunct from every set succeeds.

The PowerPC code generation has been carefully tuned to take use of the optimal code sequence in most cases. For example, the specialized masking instruction **rlwinm.** (which is not described in Appendix A for the sake of brevity) is emitted for EQ/NEQ if this allows to replace longer sequences of bit-AND and compare instructions.

B.4 Nondeterminism Stack

Since the underlying formalism is basically a non-deterministic finite-state machine, a stack is needed to hold machine configurations when several transitions can be taken. This stack is controlled by the instructions described in this section.

B.4.1 push *newState, newStateAccepting*

The purpose of the **push** instruction is to save the current machine configuration onto the non-determinism stack.

The parameter *newState* indicates which state will be current when popping. It is passed with *newStateAccepting* whether this state is accepting, since this allows for another nano-optimization.[76]

In the (albeit very rare) case when the stack is already full, the **push** instruction has no effect. The PowerPC code generation reflects the low probability of this event by appropriately setting a flag for Static Branch Prediction.

[76] If the new state is accepting, it is not necessary to store the longest match on the stack — this value will be overwritten as soon as the code for the new state is entered after popping. In this case, it is thus possible to save one machine cycle.

B.4.2 pop *startStateLabel, automatonID, states, targets, probab*

The task of the **pop** instruction is to remove an entry from the non-determinism stack and to change the current machine configuration accordingly. After loading a number of registers from the stack, the new state is retrieved from the stack, and the Program Counter is set to the location which corresponds to the entry of that state.

In the (highly probable) case of an empty stack, the matching starts over from the next token by changing a number of registers before jumping to *startStateLabel*.

If there is no next token, because the end of the sentence has been reached, control is passed to the caller by executing the code which is corresponds to the **epilog** instruction described in section B.1.4.

The *states, targets* and *probab* parameters indicate for each state its ID, branch target and probability to be the topmost element on the stack. Of course, the probabilities of all states sum up to 1. The constructor of the pop instruction sorts the states according to their probability in descending order.

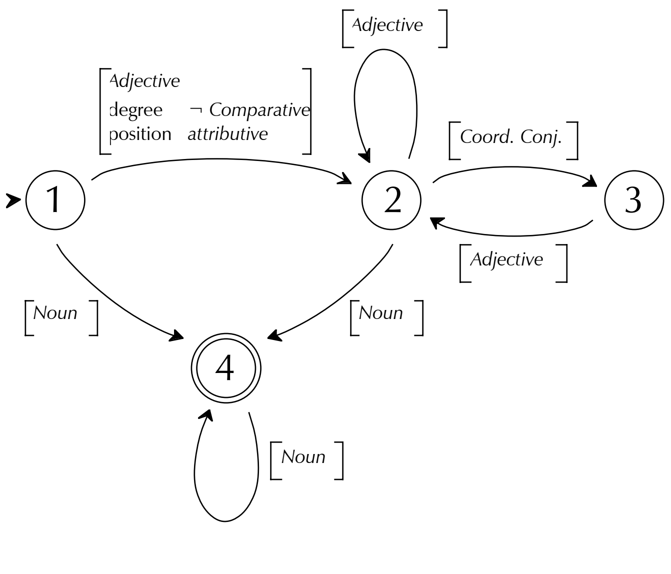
The code generation for PowerPC benefits from this, even if the current version of the compiler assigns equal probability to all states. After each non-taken branch, the probability increases that the next test will succeed. A statistical model has been developed to calculate the probability that one state is topmost, conditional to the fact that a set of other states is known to be not the topmost element on the stack.

C Example

This chapter gives an example input to Patti, shows the intermediate processing steps in the compiler, and lists the Assembly code generated as output.

C.1 Automaton

The subsequent automaton is a simplified version of the one currently utilized in the Linguistic Analysis Library to extract simple noun groups. A number of states and transitions have been removed in order to limit the length of the listings in this chapter. The restriction on attributive, non-comparative adjectives has no linguistic motivation; it serves merely to illustrate how feature unification is performed.



C.2 Compiled Type Hierarchy

Only very few types survive the steps of section 5.1 (“Extraction of Used Parts”). Therefore, it would not make sense to depict the entire original type hierarchy used at Apple, consisting of 80 types and 184 features.

The reduced type hierarchy includes the following:

- Matrix Sorts: *Adjective, Noun, Verb, Adjective, Adverb, Preposition, Pronoun, To, Determiner, Cardinal, Negation, Interjection, Coordinating Conjuction, Subordinating Conjunction, Agreement* and *Tense/Aspect*.
- Atomic Sorts: *Boolean* (sub-types: *False, True*), *Person/Number (Singular (Sg1, Sg2, Sg3), Plural (Pl1, Pl2, Pl3)), Gender (Masculine, Feminine, Neuter), Case (Nominative, Genitive, Dative, Accusative), Degree (Absolute, Comparative, Superlative), Adjective Position (Attributive, Predicative), Tense (Finite, Present, Past, Future)* and *Consistency (Countable, Mass)*.

Most of these types are present at run-time because of the *forcePresence* flag, not because they are utilized in the automaton of C.1.

C.3 Intermediate Code

```
                declaration          automatonID=1
                prolog               startStateAccepting=no
state1:         stateEntry           accepting=no
tr1_1:          oracle               oracleWord=1, prediction=doesMatch,
                                     failAction=<jump tr1_2>
                call                 target=n1_1
                jump                 target=state4
tr1_2:          oracle               oracleWord=2, prediction=doesMatch,
                                     failAction=<jump pop>
                readingLoopInit      firstVectorWordAccessed=0
tr1_2_do:       readingLoopBody      typeName="Adjective", failAction=<jump tr1_2_nxt>,
                                     firstVectorWordAccessed=0, firstUnif=3, lastUnif=3
tr1_2_chk1:     loadVectorWord       offset=0, register=2
                checkWord            failAction=<jump n1_1_nxt>, register=2,
                                     checks=[ EQ(⟨0001 0000 0000 0000 0000 0000 0000 0000⟩,
                                                 ⟨0011 0000 0000 0000 0000 0000 0000 0000⟩),
                                              AND(⟨1000 0000 0000 0000 0000 0000 0000 0000⟩)]
                jump                 target=state2
tr1_2_nxt:      readingLoopNext      failAction=<jump pop>, bodyLabel=tr1_2_do,
                                     predictMoreReadings=true
n1_1:           oracle               oracleWord=2, prediction=doesMatch, failAction=<return>
                readingLoopInit      firstVectorWordAccessed=0
n1_1_do:        readingLoopBody      typeName="Adjective", failAction=<jump n1_1_nxt>,
                                     firstVectorWordAccessed=0, firstUnif=3, lastUnif=3
n1_1_chk1:      loadVectorWord       offset=0, register=2
                checkWord            failAction=<jump n1_1_nxt>, register=2,
                                     checks=[ EQ(⟨0001 0000 0000 0000 0000 0000 0000 0000⟩,
                                                 ⟨0011 0000 0000 0000 0000 0000 0000 0000⟩),
                                              AND(⟨1000 0000 0000 0000 0000 0000 0000 0000⟩)]
                jump                 target=push1_1
n1_1_nxt:       readingLoopNext      failAction=<return>, bodyLabel=n1_1_do,
                                     predictMoreReadings=true
push1_1:        push                 newState=1, newStateAccepting=no
                return
state2:         stateEntry           accepting=no
tr2_1:          oracle               oracleWord=2, prediction=doesMatch,
                                     failAction=<jump tr2_2>
                call                 target=n2_1
                jump                 target=state2
tr2_2:          oracle               oracleWord=4, prediction=doesMatch,
                                     failAction=<jump tr2_3>
                call                 target=n2_2
                jump                 target=state3
tr2_3:          oracle               oracleWord=1, prediction=doesMatch,
                                     failAction=<jump pop>
```

```
            jump            target=state4
n2_1:       oracle          oracleWord=4, prediction=doesMatch,
                            failAction=<jump n2_2>
            jump            target=push2_1
push2_1:    push            newState=1, newStateAccepting=no
n2_2:       oracle          oracleWord=1, prediction=doesMatch, failAction=<return>
            jump            target=push2_2
push2_2:    push            newState=2, newStateAccepting=yes
            return
state3:     stateEntry      accepting=no
tr3_1:      oracle          oracleWord=2, prediction=doesMatch,
                            failAction=<jump pop>
            jump            target=state2
state4:     stateEntry      accepting=yes
tr4_1:      oracle          oracleWord=1, prediction=doesMatch,
                            failAction=<jump pop>
            jump            target=state4
pop:        pop             startStateLabel=state1, automatonID=1, states=[1, 2, 3, 4],
                            targets=[state1, state2, state3, state4],
                            probab=[0.25, 0.25, 0.25, 0.25]
            epilog
            declarationEnd
```

C.4 Generated PowerPC Code

The following lists the Assembly code, as generated by the PowerPC code generation for the Intermediate Code of section C.3. Since comments greatly facilitate debugging, Patti includes them with its output, although they are ignored by the Assembler.

The 144 PowerPC Assembly statements are translated by the Assembler[77] into 576 bytes of machine code in the final executable object file.

```
/************************************************************************/
/*                                                                      */
/*  PatternMatcher.c                                                    */
/*  ================                                                    */
/*                                                                      */
/*  Linguistic Analysis Library                                         */
/*  Pattern Matching                                                    */
/*                                                                      */
/*  Compilation Date: Tue Dec 30 08:39:06 GMT+01:00 1997                */
/*  Copyright 1997 by Apple Computer, Inc.                              */
/*                                                                      */
/************************************************************************/
```

[77] For assembling, Metrowerks CodeWarrior Professional Relase 2 was used. Neither size nor contents however depend on the assembler, since assembly code is just a mnemonic representation for machine language.

```
// Warning
// =======
// The entire content of this file has been generated by a compiler. Do not
// change this file, or your changes will be overwritten at the next run
// of the pattern compiler.
// If you have any questions, please send an e-mail to Sascha Brawer
// <brawer@coli.uni-sb.de>.

#include "PatternMatcher.h"
#include "TextAnalyzerPrivate.h"

#if defined(powerc) || defined(__powerc)

// ----------------------------------------------------------------------------
//   ApplyPattern1 (PowerPC)
// ----------------------------------------------------------------------------
//  Applies pattern #1 to a single sentence.
//
//  Register Usage On PowerPC
//  ========================
//  r0  ---            intermediate calculation results
//  r3  stopMatch      end of match arena
//  r4  startMatch     slot before that one fed into first transition
//  r5  stack          first free item in nondeterminism stack (grows upwards)
//  r6  freeStackSpace number of free slots in nondeterminism stack
//  r7  acceptedSlot   last slot that was consumed by a transition into an
//                     accepting node
//  r8  curMatchSlot   current match arena slot
//  r9  oracle         oracle vector of current token
//  r10 reading        pointer to current reading
//  r11 ---            intermediate calculation results
//  r12 ---            intermediate calculation results
//  r13 nextReading    pointer to next reading
//  r14 ---            intermediate calculation results
//
//  cr0 ---            intermediate boolean calculation results
//  cr1 ---            intermediate boolean calculation results
//  cr5 wasLastReading true if nextReading is NULL pointer
//  cr7 ---            intermediate boolean calculation results

asm void ApplyPattern1(
    register LAMatchSlot*       stopMatch,      // last match slot
    register LAMatchSlot*       startMatch,     // first match slot
    register LAMatchStackEntry* stack)          // memory block for stack
{
            mflr      r0                             // fetch content of link register
            mr        r8,r4                          // curMatchSlot = startMatch;
            li        r7,0                           // acceptedSlot = NULL;
            li        r6,kStackSize                  // freeStackSpace = kStackSize;
            stw       r13,-76(sp)                    // save content of r13
            stw       r14,-72(sp)                    // save content of r14
            stw       r0,8(sp)                       // save content of link register
state1:     lwz       r9,sizeof(LAMatchSlot)(r8)     // oracle = ++slot
            addi      r8,r8,sizeof(LAMatchSlot)      //   ->oracleWord;
tr1_1:      andi.     r0,r9,1                        // if (!Oracle(Noun))
            beq-      tr1_2                          //   go to <transition 2>
            bl        n1_1                           // check other transitions
            b         state4                         // go to <state 4>
```

```
tr1_2:      andi.    r0,r9,2                        // if (!Oracle(Adjective))
            beq-     pop                            //   go to <failure>
            lwz      r10,LAMatchSlot.token(r8)      // token = curMatchSlot->token;
            li       r14,32                         // <offset of 1. accessed word>
            lwz      r10,LAToken.firstReading(r10)  // reading = token->firstReading;
                                                    // do {
tr1_2_do:   lhz      r11,LAReading.typeID(r10)      //   type = reading->typeID;
            lwz      r13,LAReading.nextReading(r10) //   nextReading
                                                    //      = reading->nextReading;
            cmplwi   cr0,r11,3                      //   if (type != <Adjective>)
            cmplwi   cr5,r13,0                      //   bool wasLastReading
                                                    //        = (nextReading == NULL);
            dcbt     0,r13                          //   cache *nextReading;
            dcbt     r14,r13                        //   cache nextReading->vec[0];
            bne-     tr1_2_nxt                      //     next_reading;
                                                    //   /* Check Adjective */
tr1_2_chk1: lwz      r12,32(r10)                    //   vector = reading->vec[0];
            xoris    r0,r12,0x1000                  //   if (Adjective.degree
            andis.   r0,r0,0x3000                   //       == Comparative)
            beq-     tr1_2_nxt                      //     next_reading;
            andis.   r0,r12,0x8000                  //   if (Adjective.position
                                                    //       != Attributive)
            beq-     tr1_2_nxt                      //     next_reading;
            b        state2                         // go to <state 2>
tr1_2_nxt:  mr       r10,r13                        //   reading = nextReading;
            bne+     cr5,tr1_2_do                   // } while (!wasLastReading);
            b        pop                            // go to <failure>
n1_1:       andi.    r0,r9,2                        // if (!Oracle(Adjective))
            beqlr-                                  //   return;
            lwz      r10,LAMatchSlot.token(r8)      // token = curMatchSlot->token;
            li       r14,32                         // <offset of 1. accessed word>
            lwz      r10,LAToken.firstReading(r10)  // reading = token->firstReading;
                                                    // do {
n1_1_do:    lhz      r11,LAReading.typeID(r10)      //   type = reading->typeID;
            lwz      r13,LAReading.nextReading(r10) //   nextReading
                                                    //      = reading->nextReading;
            cmplwi   cr0,r11,3                      //   if (type != <Adjective>)
            cmplwi   cr5,r13,0                      //   bool wasLastReading
                                                    //        = (nextReading == NULL);
            dcbt     0,r13                          //   cache *nextReading;
            dcbt     r14,r13                        //   cache nextReading->vec[0];
            bne-     n1_1_nxt                       //     next_reading;
                                                    //   /* Check Adjective */
n1_1_chk1:  lwz      r12,32(r10)                    //   vector = reading->vec[0];
            xoris    r0,r12,0x1000                  //   if (Adjective.degree
            andis.   r0,r0,0x3000                   //       == Comparative)
            beq-     n1_1_nxt                       //     next_reading;
            andis.   r0,r12,0x8000                  //   if (Adjective.position
                                                    //       != Attributive)
            beq-     n1_1_nxt                       //     next_reading;
            b        push1_1                        //   goto success;
n1_1_nxt:   mr       r10,r13                        //   reading = nextReading;
            bne+     cr5,n1_1_do                    // } while (!wasLastReading);
            blr                                     // return;
push1_1:    cmpwi    r6,0                           // /* push onto stack */
            li       r0,1                           // if (freeStackSpace == 0)
            beqlr-                                  //   return;
            stw      r8,LAMatchStackEntry.slot(r5)  // push curMatchSlot
            stw     r7,LAMatchStackEntry.acceptedSlot(r5) // push acceptedSlot
            stw     r0,LAMatchStackEntry.state(r5) // push <state 1>
            subi    r6,r6,1                      // freeStackSpace--;
            addi    r5,r5,sizeof(LAMatchStackEntry) // stack++;
            blr
state2:     lwz     r9,sizeof(LAMatchSlot)(r8)   // oracle = ++slot
            addi    r8,r8,sizeof(LAMatchSlot)    //   ->oracleWord;
tr2_1:      andi.   r0,r9,2                      // if (!Oracle(Adjective))
            beq-    tr2_2                        //   go to <transition 2>
            bl      n2_1                         // check other transitions
            b       state2                       // go to <state 2>
tr2_2:      andi.   r0,r9,4                      // if (!Oracle(CoordConj))
            beq-    tr2_3                        //   go to <transition 3>
            bl      n2_2                         // check other transitions
            b       state3                       // go to <state 3>
tr2_3:      andi.   r0,r9,1                      // if (!Oracle(Noun))
            beq-    pop                          //   go to <failure>
            b       state4                       // go to <state 4>
n2_1:       andi.   r0,r9,4                      // if (!Oracle(CoordConj))
            beq-    n2_2                         //   try <transition 2>;
            b       push2_1                      //   goto success;
push2_1:    cmpwi   r6,0                         // /* push onto stack */
            li      r0,1                         // if (freeStackSpace == 0)
            beqlr-                               //   return;
            stw     r8,LAMatchStackEntry.slot(r5)  // push curMatchSlot
            stw     r7,LAMatchStackEntry.acceptedSlot(r5) // push acceptedSlot
            stw     r0,LAMatchStackEntry.state(r5) // push <state 1>
            subi    r6,r6,1                      // freeStackSpace--;
            addi    r5,r5,sizeof(LAMatchStackEntry) // stack++;
n2_2:       andi.   r0,r9,1                      // if (!Oracle(Noun))
            beqlr-                               //   return;
            b       push2_2                      //   goto success;
push2_2:    cmpwi   r6,0                         // /* push onto stack */
            li      r0,2                         // if (freeStackSpace == 0)
            beqlr-                               //   return;
            stw     r8,LAMatchStackEntry.slot(r5)  // push curMatchSlot
                                                 // /* acceptedSlot not pushed:
                                                 //    <state 2> is accepting */
            stw     r0,LAMatchStackEntry.state(r5) // push <state 2>
            subi    r6,r6,1                      // freeStackSpace--;
            addi    r5,r5,sizeof(LAMatchStackEntry) // stack++;
            blr
state3:     lwz     r9,sizeof(LAMatchSlot)(r8)   // oracle = ++slot
            addi    r8,r8,sizeof(LAMatchSlot)    //   ->oracleWord;
tr3_1:      andi.   r0,r9,2                      // if (!Oracle(Adjective))
            beq-    pop                          //   go to <failure>
            b       state2                       // go to <state 2>
state4:     lwz     r9,sizeof(LAMatchSlot)(r8)   // oracle = ++slot
            addi    r8,r8,sizeof(LAMatchSlot)    //   ->oracleWord;
            mr      r7,r8                        // acceptedSlot = startMatch;
tr4_1:      andi.   r0,r9,1                      // if (!Oracle(Noun))
            beq-    pop                          //   go to <failure>
            b       state4                       // go to <state 4>
pop:        cmpwi   r6,kStackSize                // if (freeStackSpace==kStackSize)
            beq+    nextToken                    //   goto nextToken;
            subi    r5,r5,sizeof(LAMatchStackEntry) // stack--;
            lwz     r0,LAMatchStackEntry.state(r5) // pop state
            lwz     r8,LAMatchStackEntry.slot(r5)  // pop slot
            cmplwi    r0,1                            // /* p(s1) = 0.25 */
            lwz       r7,LAMatchStackEntry.acceptedSlot(r5) // pop acceptedSlot
            addi      r6,r6,1                         // freeStackSpace++;
            beq-      state1
            cmplwi    r0,2                            // /* p(s2) = 0.25,
            beq-      state2                          //    p(s2|~s1) = 0.3333333333 */
            cmplwi    r0,3                            // /* p(s3) = 0.25,
            beq+      state3                          //    p(s3|~s1,~s2) = 0.5 */
            b         state4                          // /* p(s4) = 0.25,
                                                      //    p(s4|~s1,~s2,~s3) = 1.0 */
nextToken:  cmplwi    cr7,r7,0                        // cr7 = (acceptedSlot == NULL);
            addi      r11,r4,2*sizeof(LAMatchSlot)    // slot* next = startMatch + 2;
            addi      r4,r4,sizeof(LAMatchSlot)       // startMatch = startMatch + 1;
            cmplw     cr1,r4,r3                       // cr1 = (startMatch < stopMatch);
            beq+      cr7,nextToken1                  // if (cr7) goto nextToken1;
            lwz       r0,(LAMatchSlot.m1-sizeof(LAMatchSlot))(r7)
                                                      // slot* x = acceptedSlot->m1;
            dcbt      0,r11                           // cache <next slot>
            cmplwi    cr0,r0,0                        // bool cr0 = (x == NULL);
            bne       cr0,nextToken1                  // if (!cr0) /* x != NULL */
            stw       r4,(LAMatchSlot.m1-sizeof(LAMatchSlot))(r7)
                                                      //   acceptedSlot->m1 = startSlot;
nextToken1: mr        r8,r4                           // curMatchSlot = startMatch;
            li        r7,0                            // acceptedSlot = NULL;
            blt       cr1,state1                      // if (cr1) goto <start state>;
epilog:     lwz       r0,8(sp)                        // restore link register
            lwz       r13,-76(sp)                     // restore register r13
            mtlr      r0
            lwz       r14,-72(sp)                     // restore register r14
            blr
}


#else
    #error "Only PowerPC platforms are supported for now."
#endif
```

D References

[Abney, 1996] Steven Abney: Part-of-Speech Tagging and Partial Parsing. In *Ken Church/Steve Young/Gerrit Bloothooft [eds.]: Corpus-Based Methods in Language and Speech. Dordrecht [et al.]: Kluwer Academic Publishers, 1996.*

[Abney, 1997] Steven Abney: Partial Parsing via Finite-State Cascades. In *Natural Language Engineering* **2** (4), pp. 339–346.

[Aït-Kaci et al., 1989] Hassan Aït-Kaci/B. Boyer/P. Lincoln/R. Nasr: Efficient implementation of lattice operations. ACM Transactions of programming languages and systems, 11.1, January 1989. *Cited after [Pulman, 1993].*

[Apple, 1994] Apple Computer, Inc., Cupertino, California, USA: Inside Macintosh: PowerPC System Software. Reading MA [et al.]: Addison-Wesley, 1994. ISBN 0-201-49727-2.

[Black, 1989] Alan W. Black: Finite state machines from feature grammars. In *Masaru Tomita [ed.]: International Workshop on Parsing Technologies. Pittsburgh PA: Carnegie Mellon University, 1989. pp. 277–285. Cited after [Pereira/Wright, 1996].*

[Boguraev/Kennedy, 1997a] Branimir Boguraev/Christopher Kennedy: Salience-based content characterisation of text documents. In *Proceedings of the Workshop on Intelligent, Scalable Text Summarisation at the Annual Meeting of the Association for Computational Linguistics, 1997.*

[Boguraev/Kennedy, 1997b] Branimir Boguraev/Christopher Kennedy: Technical Terminology for Domain Specification and Content Characterisation. In *Maria Teresa Pazienza [ed.]: Information Extraction — a Multidisciplinary Approach to an Emerging Information Technology. Springer Lecture Notes in Computer Science #1299. Berlin [et al.]: Springer, 1997. ISBN 3-540-63438-X.*

[Boguraev et al., 1998] Branimir Boguraev/Yin Yin Wong/Christopher Kennedy/Rachel Bellamy/Sascha Brawer/Jason Swartz: Dynamic Presentation of Document Content for Rapid On-Line Skimming. In *Proceedings of AAAI Spring Symposium on Intelligent Text Summarization, Palo Alto CA, March 1998.*

[Brawer, 1995] Sascha Brawer: Treating German with a Provable Context-Free Grammar. Coping with Subcategorization, Unbounded Dependencies and Partially Free Word-Order. In *Proceedings der 5. Tagung der Computerlinguistik-Studenten (5. TaCoS), Saarbrücken, May 1995.* pp. 5–17.

[Carl/Schmidt-Wigger, 1998] Michael Carl/Antje Schmidt-Wigger: Shallow Post Morphological Processing with KURD. In *Proceedings of NeMLaP (New Methods in Language Processing), Sydney, January 1998.*

[Carpenter, 1992] Bob Carpenter: The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution. Cambridge Tracts in Theoretical Computer Science, vol. 32. Cambridge: Cambridge University Press, 1992. ISBN 0-521-41932-8.

[Carpenter/Penn, 1995] Bob Carpenter/Gerald Penn: Compiling Typed Attribute-Value Logic Grammars. In *Harry Bunt/Masaru Tomita [ed.]: Current Issues in Parsing Technologies, vol. 2. Dordrecht [et al.]: Kluwer Academic Publishers, 1995.*

[Carroll, 1994] John Carroll: Relating Complexity to Practical Performance in Parsing with Wide-Coverage Unification Grammars. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics, Las Cruces, New Mexico, USA, June 1994.* pp. 287–294.

[Erbach, 1994] Gregor Erbach: ProFIT — Prolog with Features, Inheritance and Templates. CLAUS Report #42. Saarbrücken, Germany: University of the Saarland (Computational Linguistics), July 1994. ISSN 0941–083X.

[Gazdar/Pullum, 1985] Gerald Gazdar/Geoffrey K. Pullum: Computationally Relevant Properties of Natural Languages and their Grammars. In *New Generation Computing* **3** (1985), pp. 273–306. Reprinted in [Savitch et al., 1987], pp. 387–437.

[Hearst, 1994] Marti A. Hearst: Multi-Paragraph Segmentation of Expository Text. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics, Las Cruces, New Mexico, USA, June 1994.*
http://xxx.lanl.gov/abs/cmp-lg/940603

[Hennessy/Patterson, 1994] John L. Hennessy/David A. Patterson: Rechnerarchitektur. Analyse, Entwurf, Implementierung, Bewertung. Braunschweig/Wiesbaden, Germany: Vieweg, 1994. [German Translation of: Computer Architecture: A Quantitative Approach. San Mateo CA [et al.]: Morgan Kaufman Publishers, 1990. ISBN 1-55880-069-8].

[Hobbs et al., 1997] Jerry R. Hobbs/Douglas Appelt/John Bear/David Israel/Megumi Kameyama/Mark Stickel/Mary Tyson: FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text. http://xxx.lanl.gov/abs/cmp-lg/9705013.

[Hoxey et al., 1996] Steve Hoxey/Faraydon Karim/Bill Hay/Hank Warren: The PowerPC Compiler Writer's Guide. [Published for IBM Microelectronics Divison.] Palo Alto CA: Warthman Associates. 1996. ISBN 0-9649654-0-2. http://www.chips.ibm.com/products/ppc/documents/compiler/cover.html

[Hutheesing, 1996] N. Hutheesing: Gilbert Amelio's Grand Scheme to Rescue Apple. In *Forbes Magazine, December 16, 1996.*

[IBM/Motorola, 1993] IBM Microelectronics Divison/Motorola Semiconductor Products Sector: PowerPC™ 601: RISC Microprocessor User's Manual. IBM Technical Document #MPR601UMU–02; Motorola Technical Document #MPC601UM/AD.

[IBM/Motorola, 1994] IBM Microelectronics Divison/Motorola Semiconductor Products Sector: PowerPC™ 604: RISC Microprocessor User's Manual. IBM Technical Document #MPR604UMU–01; Motorola Technical Document #MPC604UM/AD.

[IBM/Motorola, 1995] IBM Microelectronics Divison/Motorola Semiconductor Products Sector: PowerPC™ Microprocessor Family: The Programmer's Reference Guide. IBM Technical Document #MPRPPCPRG–01; Motorola Technical Document #MPCPRG/D. http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/PRG.pdf

[IBM/Motorola, 1997] IBM Microelectronics Divison/Motorola Semiconductor Products Sector: PowerPC™ Microprocessor Family: The Programming Environments. IBM Technical Document #G522-0290-00; Motorola Technical Document #MPCFPE/AD.

[Joshi/Hopely, 1997] Aravind K. Joshi/Phil Hopely: A Parser from Antiquity. In *Natural Language Engineering* **2** (4), pp. 293–296.

[Kasper/Rounds, 1986] Robert T. Kasper/William C. Rounds: A Logical Semantics for Feature Structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, 1986,* pp. 257–266.

[Kazmarcik, 1995] Gary Kacmarcik: Optimizing PowerPC code. Reading MA [et al.]: Addison-Wesley, 1995. ISBN 0-201-40839-2.

[Kennedy/Boguraev, 1996a] Christopher Kennedy/Branimir Boguraev: Anaphora for Everyone: Pronominal anaphora resolution without a parser. In *Proceedings of COLING-96 (16th International Conference on Computational Linguistics), Copenhagen, Denmark, 1996.*

[Kennedy/Boguraev, 1996b] Christopher Kennedy/Branimir Boguraev: Anaphora in a wider context: Tracking discourse referents. In *Wolfgang Wahlster [ed.]: Proceedings of the 12th European Conference on Artificial Intelligence, Budapest, Hungary, 1996. London/New York: John Wiley and Sons, 1996.*

[Kornai, 1997] András Kornai: Extended Finite State Models of Language. In *Natural Language Engineering* **2** (4), pp. 289–292.

[Lewis/Papadimitriou, 1981] Harry R. Lewis/Christos H. Papadimitriou: Elements of the Theory of Computation. London [et al.]: Prentice-Hall, 1981.
ISBN 0-13-273426-5.

[Motorola, 1997] Motorola, Inc.: PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors. Motorola Document #MPCFPE32B/AD.

[Nakazawa/Neher, 1987] Tsuneko Nakazawa/Laura Neher: Rule Expansion on the Fly: A GPSG Parser for Japanese/English using a Bit Vector Representation of Features and Rule Schemas. Studies in Linguistic Sciences, Volume 17, Number 2, Fall 1987. Urbana-Champaign, Illinois, USA: University of Illinois, Dept. of Linguistics, 1987.

[Nakazawa et al., 1988] Tsuneko Nakazawa/Laura Neher/Erhard W. Hinrichs: Unification with Disjunctive and Negative Values for GPSG Grammars. In *Proceedings of the European Conference on Artificial Intelligence, Munich, Germany, August 1–5, 1988.* pp. 467–472.

[Naumann/Langer, 1994] Sven Naumann/Hagen Langer: Parsing: Eine Einführung in die maschinelle Analyse natürlicher Sprache. Stuttgart: Teubner, 1994.
ISBN 3-519-02139-0.

[Neumann et al., 1997] Günter Neumann/Rolf Backofen/Judith Baur/Markus Becker/Christian Braun: An Information Extraction Core System for Real World German Text Processing. In *Proceedings of ANLP-97, Washington, March 1997.* pp. 208–215.

[Pereira/Wright, 1996] Fernando C. N. Pereira/Rebecca N. Wright: Finite-State Approximation of Phrase-Structure Grammars. Revised and extended version of a paper with the same title, printed in *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics, Berkeley CA, 1991,* pp. 246–255. http://xxx.lanl.gov/abs/cmp-lg/9603002

[Pulman, 1993] Stephen G. Pulman: Unification Encodings of Rich Grammatical Formalisms. *Unpublished manuscript.*

[Rood, 1996] C. M. Rood: Efficient Finite-State Approximation of Context Free Grammars. In *András Kornai [ed.]: Proceedings of the Workshop on Extended Finite-State Models of Language at the European Conference on Artificial Intelligence, 1996.*

[Russi, 1990] Thomas Russi: A Framework for Syntactic and Morphological Analysis and its Application in a Text-to-Speech System. Diss. #9328 Swiss Federal Institute of Technology, Zürich, Switzerland, 1990.

[Samuelsson, 1994] Christer Samuelsson: Fast Natural-Language Parsing Using Explanation-Based Learning. Diss. Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 1994. SICS Dissertation Series 13. ISBN 91-7153-146-7.

[Savitch et al., 1987] Walter J. Savitch/Emmon Bach/William Marsh/Gila Safran-Naveh [eds.]: The Formal Complexity of Natural Language. Studies in Linguistics and Philosophy, vol. 33. D. Reidel, Dordrecht: 1987. ISBN 1-55608-046-8 and 1-55608-047-6.

[Traber, 1995] Christof Traber: SVOX: The Implementation of a Text-to-Speech System for German. Diss. #11064 Swiss Federal Institute of Technology, Zürich, Switzerland. Zürich: vdf Hochschulverlag, 1995. ISBN 3-7281-2239-4.

[Voutilainen et al., 1992] Atro Voutilainen/Juha Heikkilä/Arto Anttila: Constraint Grammar of English — A Performance-Oriented Introduction. Helsinki, Finland: University of Helsinki, Department of General Linguistics, 1992.

[Voutilainen, 1993] Atro Voutilainen: NPtool, a Detector of English Noun Phrases. In *Proceedings of the Workshop on Very Large Corpora, Ohio State University, June 22, 1993.*

[Voutilainen, 1995] Atro Voutilainen: A Syntax-Based Part-of-Speech Analyser. In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics, Dublin, 1995.*

[Wilhelm/Maurer, 1992] Reinhard Wilhelm/Dieter Maurer: Übersetzerbau: Theorie, Konstruktion, Generierung. Berlin [et al.]: Springer, 1992. ISBN 3-540-55704-0.

[Wintner/Francez, 1994] Shuly Wintner/Nissim Francez: Abstract Machine for Typed Feature Structures. Technical Report #LCL 94–8. Haifa: Technion (Israel Institute of Technology), 1994.

[Wintner/Francez, 1995] Shuly Wintner/Nissim Francez: Abstract Machine for Typed Feature Structures. In *Proceedings of the 5th International Workshop on Natural Language Understanding and Logic Programming, Lisbon, Portugal, May 1995*. http://xxx.lanl.gov/abs/cmp-lg/9504009

[Wintner et al., 1997] Shuly Wintner/Evgeniy Gabrilovich/Nissim Francez: Amalia — A Unified Platform for Parsing and Generation. In *R. Mitkov/N. Nicolov/N. Nikolov [eds.]: Proceedings of "Recent Advances in Natural Language Processing" (RANLP '97), Tzigov Chark, Bulgaria, September 1997, pp. 135–142*. http://xxx.lanl.gov/abs/cmp-lg/9709014