Thesis

De nition and implementation of a

Front-end generator for Oz

of

Leif Kornstaedt

September 1996

Supervisor:

Prof. Dr. rer. nat. Gert Smolka
Dipl.-Inform. Christian Schulte
Prof. Dr.-Ing. Hans-Wilm Wippermann

Department of Computer Science

University of Kaiserslautern

Page 3

Explanation

I, Leif Kornstaedt, hereby declare that I have prepared the present thesis myself and did not use any aids other than those specified.

Kaiserslautern, September 16, 1996

Summary

In the present diploma thesis, a front-end generator is developed that tiparadigmatic language Oz used as target language. This confirms the suitability of Oz as an implementation language for compilers and the reimplementation of the Oz compilers prepared in Oz.

The tool is particularly suited to translating fully compositional languages. lays. This means that in addition to the lexical and syntactic analysis, the uber so-called production schemes can also be de ned by your own EBNF operators, the tool also covers the reduction into a core language. To the latter take care and simplify the implementation of tree transformations, in which no con icts of the variable names may occur, an au- Binding analysis carried out automatically with renaming of all bound identifiers offered by the tool.

contents

Page 8

Chapter 1

Introduction and goal setting

1.1 Motivation

If you consider the ubiquity of software in everyday life and the increasing
de tendency to entrust safety-critical tasks to software systems, this is how the
The importance of eight development tools is evident. Through their use, a
ne higher quality at the software products can be achieved, for example with regard to their
Ease of use and reliability. These tools also include the
deten programming languages as well as their programming systems. This is why the
Research in these areas is still very interesting.

To use a programming language reliably in the development of large software projects
To be able to use it, it should be precisely de ned and a correct compiler or
Interpreters for them exist. For this, the relationships between the de ning
Document of the language (the language report) and its implementation at source code level
be clearly recognizable. The formalisms used for de nition must therefore be very
can be transformed into an executable program at a high level.

For the formalisms that have proven their worth in the past {especially for the
Description of the syntactic structure {, is a variety of so-called compiler
Generators (or compiler-compilers) have been developed. From these the
mostly in the traditional imperative languages that emerged from Algol 60 N + 63]
genes are. In addition to the imperative languages, research is also carried out on languages that use
build other paradigms, for example purely functional ones with lazy evaluation PJ88],
logical War77] or multiparadigmatic languages Smo94]. For this often
new models are found.

More extensive tool support is therefore only available for imperative languages
available, but especially with these there is a particularly large gap between the speci
on and the expressiveness of the implementation language. Fully compositional languages

on the other hand, they are already well suited to implementing mathematically based formalisms.
suitable. Some arguments for the particular suitability of newer programming language concepts
for the compiler building can be found in War80] and JPB94].

The aim of this thesis is to use the tried and tested compiler generator technology to
to expand mechanisms that facilitate the translation of modern languages. To this end

a tool for the multiparadigmatic language Oz will be made available that is developed by the programming systems research division in Saarbrücken. The The developed tool is to be used later for a reimplementation of the Oz compiler will. In particular, tasks such as lexical linkage analysis and reduction into a core language that is rarely done automatically in traditional tools, should be supported.

This chapter describes the exact requirements for the Tool and its specification language are developed. Advance is in the following Sections for a better understanding of the task environment initially focus on the structure of Compilers as well as the special features of Oz.

1.2 The structure of compilers

A compiler transforms an input that is available in a certain source language, into a semantically equivalent output in a target language. Usually, but not necessarily, the input is in the form of a character string.

This task has traditionally been divided into several logical phases. They offer one good opportunity to structure the compilers, which are often very complex to make it understandable. Figure 1.1 shows the standard compiler model that this Work is based on (modified from ASU86, p. 10]). Here, rectangles represent phases, rounded rectangles represent components of the compiler; Arrows indicate data u, through Drawn lines indicate the use of a component.

The phases can be assigned to two groups, as shown in the figure the dashed line has been done. The front end takes over the analysis phases, i.e. the those who only speak of the source language, but not (or only slightly) of the target language These include the lexical, syntactic and semantic analysis as well as the Intermediate code generation, error handling and symbol table management tion. The back-end, on the other hand, does the synthesis tasks. The phases in this part depend very much on the target language or its execution engine.

The individual phases and components are briefly described in the following sections described. The implementation of the back end is not the subject of the investigation in the present work, therefore only the tasks of the front-end are more detailed and only for these possibilities to use compiler generators. given.

Source program

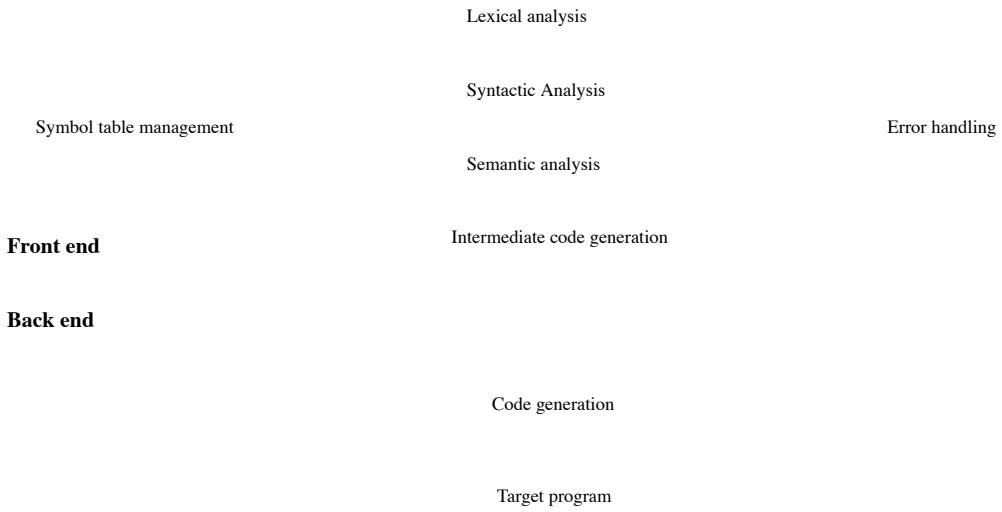


Figure 1.1: The phases of a compiler

1.2.1 Lexical analysis

The lexical analysis, in the literature often also scanning, lexing, tokenization or called linear analysis, combines partial sequences of the input character stream into tokens, each of which is assigned a token type or token tag (for example „Loading draftsman \) and possibly carries a token value (such as the representation of a Identifier). The output is the token stream, i.e. a sequence of tokens.

The exact representation of the token types is insignificant {is only important because everyone Token type is unique. Any sequence of data or events that have clear to- can be assigned to types without separate lexical analysis in the to be passed on to the next phase.

There are good options for tool support for this phase: If the operator If the user specifies the set of character strings belonging to each token type, then This information is automatically generated by a finite automaton that contains the token recognizes. Such sets can be described by so-called regular expressions.

1.2.2 Syntactic Analysis

This phase can also be found in the literature under the terms parsing and structured tural or hierarchical analysis. According to certain rules, partial sequences of the Token streams successively replaced by new symbols. If one looks at the replaced symbols le as the successor to the new one, the linear token stream is converted into a syntax tree (also:

Structure tree) transformed. Usually, however, only a more compact representation is used constructed from it, namely the abstract syntax tree that is the output of this phase.

Tool support is also available for the implementation of the syntactic analysis possible by specifying a context-free grammar. The concrete syntax tree depends on the exact formulation of the grammar. The automatic determination a suitable abstract syntax from the given grammar is an important aspect the translation, as the complexity of the subsequent phases can depend on this. There the parsing algorithm used additional restrictions on the form of the concrete Due to the syntax tree, a representation can differ from the subtleties in grammar is independent, but cannot be determined by the user without further information.

1.2.3 Semantic analysis

In this phase the input will be checked for validity. For this you have to the rules formally established by the language report for the selected abstract syntax can be formulated and applied when running through the abstract syntax tree will. Information derived from the context is stored in the tree nodes saved. Furthermore, constructs can be simplified or reduced to more general ones will. The output of the phase is an annotated abstract syntax tree.

If in the implementation language convenient matching and tree flow Strategies are easy to formulate, this phase can be implemented more easily. Tool Initial use here often only has the goal of adding this kind of functionality to a language. tern (Pattern-Based Transformation Tools, Tree Walkers). Also for simplifications is the provision of resources interesting; it is conceivable to specify rules like them be used in term rewriting systems.

Lexical attachment analysis

For the semantic analysis of almost all languages it is necessary to use the to be assigned to the corresponding de nition. This step is with the symbol described below bolt table management is closely coupled and can be used in most languages by and large can all be solved in the same way by a lexical linkage analysis. Plant- tool support would mean here, as some frequently required functions are simple can be made available, such as the creation of a fresh (un- used) identifier and the consistent renaming of all identifiers, so there are no two binding occurrences of the same identifier occur.

This is important for declarative languages, the semantic definition of which is often does not interpret the entered program, but rather its quotient with respect to alpha Renaming considered. Alpha renaming is the name given to consistent renaming of a bound identifier, i.e. at the locations of its de nition and of all its references limit. With this prerequisite, the implementation of transformations is facilitated,

as it is guaranteed from the outset that no identifier con icts (so-called name class hes) can occur as long as no program parts are duplicated.

1.2.4 Intermediate code generation

Often the representation in abstract syntax does not directly become the code of the target language generated, but intermediate code that is located between the source and target language is. This should be as independent of the target language as possible in order to allow for later translations into other target languages to facilitate. In this phase the annotated abstract syntaxbaum translated into the intermediate language. This transformation depends on both the abstract syntax as well as from the target language and is therefore difficult to use tools realizable.

1.2.5 Symbol table management

During the analysis, information on the in the source text is displayed in the symbol table occurring identifiers collected. These include, for example, the location of their deKlaration, the de nition of the object that is bound to the identifier, or be Visibility area. Typical operations on a symbol table are the entry of a an identifier with its de nition or looking up a de nition in the current one Surroundings.

1.2.6 Error handling

In the event of an error, information about this is output in legible form. It It must be decided whether and how to proceed with the translation process should.

1.3 The programming language Oz

This section introduces some of the features of the Oz programming language. which are important for an understanding of the work, and their importance for the compiler construction outlines.

1.3.1 Logical variables

A logical variable is a placeholder for a (cannot be changed) value. The variable can be used before its value is fully known {unlike radio tional programming languages in which at the time of the de nition of a variable

your complete calculation rule must already be specified, or too imperative
ven languages, which are usually used with an error before initialization
represents unpredictable consequences.

The value of a variable is specified using constraints. For example,
be known, since a variable X stands for a record with certain subtrees. Theirs
However, values can still be unspecified and can only be derived from later calculations.
give (or not at all). There are constructs that check the entailment of constraints
can. This can be used to achieve matching of values that determine logical variables.
hold what is very often required in compilers.

Logical variables can be used usefully in a compiler front-end: So
could, for example, contain logical variables in the abstract syntax tree, which will later be
tations, or in the symbol table could logical variables at the positions
can be used where not all information about an identifier is known
{as for example in the case of a declared identifier whose exact definition is still
pending. Another application is backpatching: If code is created at a point in time
must be generated for which the value of an operand has not yet been determined, then about
A placeholder can be inserted into a logical variable, which then has the final value
records.

1.3.2 Higher Order Procedures

Oz supports higher order procedures. This means that there are closures at runtime
generated (i.e. quantifications made using non-local variables) and procedures
Variables can be assigned.

Higher order procedures can be used in compilers just as diverse
as in other applications. On an application that has been proposed is supposed to
will be discussed in more detail here.

There are parsing techniques in which for the construction of a node of the abstract syntax
tree, only the successor nodes can be referenced (i.e. none that are linked to other
Digits of the input). This restricts attribute calculations to very local
le data. Higher order procedures can be used to remove this restriction
TA91], May81]: Instead of a node construction, its attribute
To determine values immediately, the calculation rule is based on the unavailable data
quantified and a procedural abstraction at the corresponding point in the abstract
Syntax tree entered. At the time when the missing information becomes accessible
the procedure can be applied to them.

1.3.3 Secondary option

Oz is a colloquial language; several calculations can be carried out in
different threads.

Since the structure of compilers is a simple chain of (possibly interlinked) analysis
This step with linear data u is for the rough structure of a compiler front-end
no sideline necessary. However, as an alternative to the above process, threads can
can be used to reset calculations that are still
cannot be carried out.

1.3.4 Object-oriented programming

Another paradigm that Oz supports is object-oriented programming.
Classes and objects are created at runtime, multiple inheritance is possible.
Private as well as public methods can be de ned and „friends \ class
sen comparable to C ++ Str87] exist. The state of an object can be constant
(Features) and changeable (attributes) proportions are divided; several objects one
Classes can share the same feature.

Object orientation can be used very well in compiler design. It enables
It is important to structure the compiler phases well and to provide them with clear interfaces.
So it is obvious to use the programs generated by a compiler tool
To encapsulate class nitions. Then, through inheritance, the behavior of a class
can be modified: For example, one and the same parser class can lead to a
Compiler front end as well as to a pretty printer can be extended by the method
the overloaded, which are carried out in the recognition of grammatical rules.

Furthermore, it has to be investigated which advantages of the structure of the abstract syntax-
tree of objects. An obvious disadvantage is that pattern-based trans-
Formation rules are no longer so easy to use through existing constructs of the language
can be learned.

1.3.5 Names

Many possibilities of the visibility control, which were mentioned in the previous section,
are made possible in Oz by so-called names. There is a procedure that creates a new
en, unique name that can be linked to a logical variable.
Since procedures, methods, and features can be named by name, they are all
Visibility regulations can be implemented, which are conceivable in the context of lexical scanning
are.

{as in all software systems {are used in favor of better maintainability. The possibility of information hiding Par77] given by this can be used in compilers

1.3.6 The levels of the de nition of Oz

The language definition of Oz is given in several levels: At the bottom is the kernel
Oz de niert Smo94], which is a simple but semantically complete partial

language acts. Thanks to its simplicity, it is axiomatically de ned. To be comfortable
To preserve language, the Oz notation is applied Hen95]. The constructs of this language
At the same level, they can all be translated into Kernel Oz. Come to the top level
the Standard Modules HMSW96], which provide the user with basic operations and often
Make used functions available.

The division of the language definition into levels offers the language developer great advantages. share and make even complex languages more accessible to the user. Can this
Structure can also be used for the compiler construction, so this also offers the
Compiler builder advantages: The reduction rules that use the notation in core language constructs translate only depends on the grammar, not on the abstract syntax
and therefore do not have to be subsequently adjusted if the latter is modified.
Furthermore, this de nition allows incremental development and testing of the compiler.
A slight changeability of these rules in the implementation has for the language development winder, in turn, has the advantage that it is very easy to experiment with new notations can.

1.4 Requirements for the front-end generator

After this overview, the requirements for the tool can be formulated.
First it is determined which phases are examined more closely and {if the scope of the
Work it perm at {should be supported:

Lexical analysis. This phase should as much as possible of the functionality be more widespread
Provide scanner generators for porting existing descriptions
to simplify.

Syntactic Analysis. The specification of the syntactic rules should be on a very high level
Level be possible. Frequently occurring syntactic patterns should be abbreviated
and abstract from the parsing algorithm used. To this
Way, a simple portability of existing descriptions should be made possible and

a high level of agreement with the language report can be guaranteed.

Lexical attachment analysis. The information necessary to carry out this phase mations should be able to be specified directly in the grammar. Furthermore should the lexical link analysis to a full-fledged symbol table management can be expanded to include the data already available at this point to be able to reuse.

Reduction into a core language. Syntactic simplifications are intended through this supported, since replacement rules can be specified.

Build an abstract syntax. The possibility of a (semi-) automatic derivation Use of abstract syntaxes from grammar should be taken into account.

Tools for syntax tree transformations and tree traversals do not need to be looking to be, since these tasks are already easy with the constructs available in Oz can be done.

Furthermore, some requirements for the interfaces are formulated:

Independent usability. The lexical analysis on the one hand and the speci tion of the remaining phases, on the other hand, should be used independently of one another can be. This allows the use of token sources other than the scanner generator and token consumers other than the parser generator as well as a free Combination of several scanners and parsers. There is also a clear structure in the lexical and syntactic parts of the analysis are required. This separation should also increase the clarity of the speci cation.

High level of integration. If all tools are to be used, they must (despite the required clear separation) be well integrated in order to to be able to be bound. For this purpose, simple, clear interfaces are used between the individual parts are required, which are coupled by the tool.

Embedding in Oz. The speci cation language should be well embedded in Oz in order to to be able to take full advantage of the printability of this language.

Information hiding. The generated analysis program should take into account the possibilities of Make use of abstraction and visibility schemes that Oz offers. It offers consider embedding the program in a class.

Interactive systems. The tool should be used for interactive systems. nen. The phases that depend on the syntactic analysis must therefore be interlinked can run with this.

1.4.1 Structure of the work

The requirements have stipulated which parts of the speci cation practice to be developed
which are independent of each other. Chapters 2 to 4 deal with this accordingly
Order the lexical analysis, the syntactic analysis and their extensions.
The focus is initially on the investigation and evaluation of related work.
whereupon the design decisions for the front-end to be developed
Generator to be met.

After the detailed requirements for the speci cation language have been established, the process is carried out
the next chapter with its design and implementation in a tool.
The components are discussed individually and their interaction is explained. Farther
a bootstrapping process is presented that enables the system to be used
of its own to implement. For readers who are only interested in the result of the work,
Chapter 5 is probably the most important.

Chapter 6 summarizes the knowledge gained and evaluates the new work
stuff. Reference is made to the requirements from Section 1.4. At the end of
Chapter gives an outlook on possible further investigations.

Chapter 2

Lexical Analysis { Scanner definition

The lexical analysis is the first step of the structural analysis, in which the entered bene character stream is converted into a token stream. Since this is consumed by parser

lexical analysis is often presented as a client process of syntactic analysis.
seen from which the parser successively requests the tokens; however, sometimes the
Tokenization completed before the parse process is started.

The task of the scanner consists of two parts. One reads the stream of characters
and determines the end of the next string to be processed, which generates the other
from this tokens that are appended to the token stream. Pr aprocesses can also be
basic tasks such as file inclusion (e.g. in Oz via \ insert), conditional comp
pilation or macro expansion can be done. Symbol table entries
generated or requested. This is necessary for languages in which the token class
one and the same character string changes in the course of the analysis. This is an example of this
Introducing a new type name in C and C ++ via typedef, whose token class is
changes from the time of de nition from 'identifier' to 'type name'. Is one of those
Adaptation not possible, then the language cannot be clearly parsed.

This chapter is structured according to the two subtasks: In Section 2.1
the recognition of a character string is described, in section 2.2 the token generation.
The solution approaches of existing tools are examined. In the last part
(2.3) it is determined how the scanner generator to be de ned should approach the tasks.
Its specification language and interfaces are de ned in Section 5.2.

Some of the designations used are taken from KVE94].

2.1 Speci cation of the input languages

As already mentioned in the introductory chapter, the recognition part of a scanner
described by the fact that for each token class the set of the character string

gen is specified. The character stream entered is based on this information
broken down and assigned token classes. For example, in Oz the input would be

```
case Xs of '|' (X1 Xr) then :::
```

divided into 14 parts: the keyword case, a space, the variable Xs, a space
chen, the keyword of, a space, the atom '|' and so on.

Technically, the decomposition is realized by a finite automaton. The amount
of all possible input characters is called a vocabulary or alphabet, a set of
Strings (for example, all of those belonging to a token type) are considered as one
formal language referred to above this vocabulary. Such sets can be given by regular
Expressions are described which are de ned in Section 2.1.1. Is a string
contained in the set described by a regular expression, one speaks of

a match. The character string is then called the sentence of the associated formal language or also {in the context of scanners {the matched lexeme.

Section 2.1.2 describes how scanners for languages can be implemented, which use different lexical rules in different sub-languages. For this purpose, the Lexical mode concept introduced.

In practice, the specified regular expressions do not represent a partitioning of the Set of all character strings that can be formed using the vocabulary. Two F all The following must be taken into account: On the one hand, inputs must be that do not have a regular expression can be assigned, can be reported as faulty, on the other hand can not empty There are intersections between the sets of two token classes. When the latter one Errors and how such con icts can be resolved is explained in Section 2.1.3 examined.

2.1.1 Regular expressions

Regular expressions, which represent a concise notation for the so-called regular languages. (grammars of type 3 in the Chomsky hierarchy GW85]) are a special case of context-free languages (type 2) and first examined by Kleene Kle56]. they have the property that they can be recognized by a finite automaton. Al- Algorithms for generating such an automaton can be found, for example, in ASU86, P. 113 {146] or Gro87].

Usually a set of regular expressions is given for the scanner generation. Only in rare cases are other techniques used for this purpose; Counterexamples are the Scanner and parser generator Mango Age94] for the programming language Self and the Grammar speci cation language GRAMOL, in which the techniques nd application which should be presented here in Chapter 3.

Most of the time, all required regular expressions are given by the user. The work zeug TXL CCH95] is structured in such a way that a default set of rules is given,

some of which can be modified. For example, a standard rule states which Characters may be used in identifiers; if you want to expand this set of characters tern, the other characters must be specified via a command line parameter will.

There are many different concrete syntaxes in the existing scanner generators for regular expressions. The most common is that of Lex Les75] and the ab- guided tool ex Pax95]). Therefore, its spellings are used here, to allow easy porting of speci cations. Alternatives can be found

for example in KVE94], Dor96], And95], Gro92], CCH95] and GS88].

Basic expressions

First of all, the basic constructs from which regular expressions can be built. The following section then expands this into an extensive prints heavier notation. Some practical aspects are included there.

De nition: A finite non-empty set V is called vocabulary . A consequence of the length n of symbols from V is a map f 1; :::; n g?! V . The set of all such consequences the length n is denoted by V_n ; the empty sequence is noted by ". The Kleene-
The conclusion of V is de ned by: V : = $\bigcup_{n \geq 0} V_n$. 2

De nition: Let V be vocabulary. A regular expression r and the formal language (menge of the sequences) L (r), for which it stands, are inductively de ned by:

r "is a regular expression with L (r) = f " g .

Let x 2 V . Then rx is a regular expression with L (r) = f x g .

Let s regular expression. Then rs * is a regular expression with L (r) = L (s).

Let s regular expression. Then r (s) is a regular expression with L (r) = L (s).

Let s and t be regular expressions. Then r st regular expression with L (r) = f uv j u 2 s ^ v 2 t g .

Let s and t be regular expressions. Then r s | t regular expression with L (r) = L (s) L (t).

2

Some examples for regular expressions are shown in Table 2.1.

Since regular expressions must be notable with the usual ASCII character set and Metacharacters must be distinguishable from vocabulary symbols, there are some

r	L (r)
a	f a g
from	f from g
a | b	f a; b g

a *	f "; a; aa; ::: g
from*	f a; ab; abb; ::: g
(a | b) *	f "; a; b; ab; ba; ::: g

Table 2.1: Examples for regular basic expressions

Deviations between the above definition and the actual syntax. On the one hand
there is the option of using quotation marks or escape symbols as well as metacharacters
to use as symbols. If a regular expression r 6 "is still noted, then
containing "characters are usually left out, for example the expression a (b |"),
which describes the set f a; ab g , to a (b |).

Extended expressions

To make regular expressions more powerful or more expressive,
Often some of the extensions presented below have been implemented.

The constructs that make regular expressions more effective are those
Line start and line end conditions and the trailing context.

If a regular expression is preceded by the character ^, it will only be matched
if the current position in the input character stream falls exactly at the beginning of a line
(i.e. either the first character of the input or the one following a newline character-
de characters). Some scanner generators (such as lx Dor96] for Haskell)
generalize this construct to any set of characters instead of New-
line.

This construct is required, for example, if {as in some C-pr aprocesses
soren {directives must appear at the beginning of a line (e.g. the pound of
#include).

If two regular expressions s and t are connected by the character /, a
s matched, but only if the text of the input stream on top matches t
(Succession context).

The record labels in Oz are an example where this construct is needed: A
Atom or a variable is only recognized as a record label if the
Identifier is followed by opening round brackets. But this is not part of the lexeme.

The end-of-line condition created by appending the $ character to a regular
pressure r is an abbreviation for the regular expression r with a
Newline character as a successor context.

Other extensions are only used for a more compact notation. There are indications for this sets, complements of sets of characters, literal strings, mandatory characters repetitions, options, (i to j) -fold repetitions and agreements of named regul arer expressions (comparable to macros).

Some examples of extended regular expressions are shown in Table 2.2. Here was $V = f$ a; b; c g fixed.

r	L (r)
from]	f a; b g
^ from]	f c g
a +	f a; aa; ::: g
a?	f "; a g
a {2,4}	f aa; aaa; aaaa g

Table 2.2: Examples of extended regular expressions

Finally, a special regular expression should be mentioned, namely the end Of-file rule, noted by <<EOF>>. This is only matched by the end of the input and must be handled as a special case.

2.1.2 Lexical modes

Many languages have embedded sub-languages, for the others lexical conventions be valid. Examples of this are compiler directives or semantic actions in compiler construction tools that are noted in the target language and embedded in a metalanguage are. To distinguish different lexical conventions of the sublanguages several tools offer lexical modes And95], also called lexical classes PDC91], start conditions or start states Les75]. In principle, for every lexical mode (and each precontext, such as ^) generates its own recognizer (although these can have common states).

The principle is simple: every regular expression is lexical with the set Modes are annotated in which it can be matched (possibly also indirectly through the ren complement, as is possible with the scanner generator Rex Gro92]). To each At this point in time, the scanner is in exactly one of these modes. When recognizing, The token can be switched to another lexical mode. For example C-comments KR78, p. 197] can be scanned because when the Start sequence / * is switched to a comment mode in which the comment text until including the sequence * / is skipped. Thereupon the original returned to the normal mode.

A possibility to annotate the regular expressions with their associated valid To make modes more manageable is the inheritance of lexical modes. Inherits a

Mode B from mode A, so in mode B only those resolved with B match
gul aren expressions; in mode A, however, all annotated with A or B. (The inheritance
from <<EOF>> receives special treatment.)

With Lex, inheritance is only possible to a very limited extent: those regular expressions
Without an explicit mode specification, all modes are considered to be orig, including
the predefined initial mode called INITIAL.

With ex as well as in the POSIX-Lex speci cation it is possible to define a le-
xical mode to either inherit all non-annotated regular expressions (inclusive)
or not to inherit (exclusive start condition). From version 2.5.1 of ex you can start
condition scopes which allow a group of regular expressions
jointly assign a lexical mode; Inheritance here is through nesting
such Scopi realizable.

Another tool that allows the inheritance of lexical modes is SAGA And95],
the parser generator for the multiparadigmatic language AKL Jan94]. There she gets over
a directive of the form: - inherits (derived, bases) speci ed.

2.1.3 Resolution of Con icts

It can happen because there are two languages using regular expressions
have been de ned, have a non-empty intersection. For example:

$$L((a \mid b)^*) \setminus L((a \mid c)^*) = L(a^*)$$

If a sequence of, a`s appears in the input, it cannot be decided which one
Rule applies. In this case one speaks of a con ict. In a typical
Many such con icts occur in the language de nition; For example, keywords are often
excellent identifier. Since it is cumbersome and confusing, regular expressions
To formulate conflict-free, there is a set of rules with which con icts are resolved
will.

The longest match rule is used as the first criterion: Are there two matches for
a pr ax of the input, the one that includes the longer text is preferred (where-
where the length of the lexeme and the successor context are summed up). The implementation
this rule is mandatory, otherwise, for example, with a regular expression aa *
can be canceled with a match after the first read a.

The partial order on the matches de ned by the longest match rule is extended
Thereafter, lexicographically combined either with the rst- t or with the best- t rule:

First fit. If two regular expressions match the same pr ax of the input, the
first noted the expression selected. (Of course there are also other priorities -
assigned conceivably as the purely descending order in the order of the notation.)

R. r	S. s
f A g begin	f A g az] +
f A g a	f A; B g a
f A g a	f A; B g az] +

Table 2.3: Examples for r more specific than s

This strategy is used by most tools (including Lex, Rex Cocktail and DLG from PCCTS PDC91]) followed. All uniqueness can be resolved, but individual regular expressions can also be used are completely covered by the others and are therefore never used.

Best fit. Suppose two regular expressions and match the same pr ax the input. Let R and S be the sets of lexical modes with which perspective r or s have been annotated (after inheritance explicitly zit have been made). Then r is preferred to s if and only if

RS ^ L (r) L (s) holds {in other words: if all pr a xes of the input, the can be matched by r, can also be matched by s, but also additional pr a xes covers. So r is really more special than s. Some examples of this can be found in belle 2.3. The partial order thus de ned for an input is not the smallest Element this is how an error in the speci cation is reported.

This strategy can be found in SAGA, for example. Your advantage is there the order of the regular expressions in the speci cation is no longer important. is giving. Two languages intersect, but neither is a true subset another, then the intersection must be covered by other expressions. By the way, the inheritance of the modes has the effect that an expression in an Guided mode is preferred to the same expression in the basic modes. Realized This rule can be changed by changing the accepting states of the (deterministic made and minimized) generated machines are considered: The quantities the accepting states must be pairwise to the for all regular expressions Subset property to be tested.

If the scanner definition uses context-free instead of regular grammars, the These criteria are not easily applicable. A solution to the resulting Problems can be found in the GRAMOL language. The Ada Ada83] -Uniqueness with regard to the distinction between quotes and character literals elegant be solved.

2.2 Options for generating tokens

If the matches for the practice of input have been determined and one of them has been selected, the only information available is which regular expression was matched and which

Length of the pr ax concerned. The token type and value must now be determined from this will.

There are three different approaches:

The simplest option that gives the user little control over token generation lat, consists of directly associating a token type with every regular expression. ren. The token values are usually only the lexeme and possibly its position in the source text (file name, line number and possibly the column number, also ordinates) are available. Unless there are special exceptions are seen, no context information can be included and the match not be reworked. Switching lexical modes can only be can be made dependent on the matched expression (in SAGA this is for example by an attached, instead of a pr a xed, annotation of a lexical mode ').

This strategy is used, for example, by TXL CCH95], SAGA, the scanner generator for W-Lisp Kuh94] and that of Eli Com96a]. Since this procedure is in the The tools concerned have been found to be too restrictive, shows that here other mechanisms were sought in order to increase the power: In SAGA, the filter introduced (which will be explained in more detail below); the scanner generator for W-Lisp enables the lexeme textually by insertions or deletions to modify. Auxiliary scanners and tokens can be used in the Eli scanner generator Processors can be specified by the names of C functions.

With the second possibility, the body becomes a for every regular expression Function specified in the target language of the scanner generator, called semantic Action. For the calculation of the token type and value, arbitrarily complex ornamental expressions can be evaluated. The token is determined by the return value of the Function given. If no token is to be returned, no return will be made. failed (in imperative languages, like C in the case of Lex) or the generated Scanner function called again recursively (as in the scanner generator ML-Lex for standard ML AMT92]).

This procedure is not only relevant for the calculation of the token type and value m eight: In the semantic action, the match itself can also be manipulated the, which affects the follow-up behavior of the machine. For example, can the match can be shortened, with the excess characters being returned to the input electricity can be returned, additional characters can be requested for the Match can be rejected (after which the next best match is chosen) or the value of the line start ags can be modified.

The third option is very similar to the second. However, it allows, if necessary, to return several tokens instead of just one or none. This can help game macro expansion can be realized or literal strings can be used as lists supplied by integers, as is done in the Oz de nition.

The idea is to encapsulate the generated scanner in a class definition and to understand every semantic action as a method. A predefined method serves to append a token to the token stream; this can be done as often as required can be carried out per action. The interaction with the machine is also possible Realized via method applications.

When choosing the strategy, one must consider that the rest of the syntactic analysis analysis can be simplified if the scanner incorporates information from the context can pull; in addition, this is sometimes even a precondition for parsing. Since the semantic analysis is the most complicated phase anyway, you should do as much work as possible. beit can be removed from the semantic actions of the scanner by including the Token values no longer require post-processing.

Another advantage is that the interface to the scanner (the token stream) is simpler and easier Is to be de ned more precisely when the tokens have already been processed. In Oz for example, unquoted and quoted atoms no longer need to be differentiated if the escape characters of the latter have already been converted. Further Examples of non-trivial tasks that arise are given in Gro88b].

The tendency seems to be towards languages for which the scanner speci cation is integrated with the languages for other tool parts, simple token generation methods are chosen. This may be due to the fact that an attempt is made here to add every additional Move work out of or out of the area covered by the tool because more emphasis was placed on the subsequent phases during development. Man should bear in mind, however, that the simple token generation options are limited by the eight They can be emulated easily and almost as clearly, and there is something else Offer {since it does not affect e ciency {can only be an advantage.

2.2.1 Filters

Some tools, such as SAGA or Mango, offer an intermediate stage between between scanner and parser still a filter phase, as shown in the data u diagram in Figure 2.1 is shown. This searches the token stream for certain patterns (i.e. partial sequences with certain token types) and allows the corresponding places

post-processing by replacing them with newly generated token sequences. In this way
For example, spaces and comments can be eliminated or macro expansions
be performed.

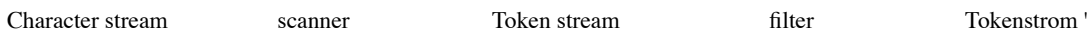


Figure 2.1: Data u diagram when using a filter

This scheme, coupled with one of the above options for token generation, is
no more than the third strategy: the method that a token is sent to the token stream
appended, can possibly be overloaded and replaced by one that has a
larger context captured. The simplest token generation strategy is when filters are de ned
can be, but more than the second method presented.

However, the third strategy from above is clearer for many applications: It
no additional token types need to be introduced that only contain one or
Force other post-processing if this is done directly in the associated method
can be.

2.2.2 Internal state

Another aspect of scanners with user-de ned semantic actions is that
Possibility to manage an internal state. This can be, for example
a symbol table or the current lexical mode. In Lex this is
simply possible due to the target language (C or C ++). Should be with ex however
a reentrant scanner is generated, the internal state must be used as an argument for the
Token request function must always be passed. (A reentrant scanner can be used from
Use several program parts at the same time for scanning different input texts.
This is similar in scanner generators with functional target languages,
like ML-Lex, handled; SAGA does this through the global threading of battery
mulators.

This way seems very cumbersome. However, if the generated scanner is converted into a
Embedded in the class, the management of an internal state is easy by means of the de nition
additional attributes possible.

2.3 Design decisions

From the previous analysis of existing tools and methods as well as the in
The requirements given in Section 1.4 are now the design decisions for the
Scanner part of the front-end generator for Oz met:

The syntax for regular expressions (excluding the lexical modes) is taken from
taken ex. It is (with the exception of variable precontexts) the eighth and
definitely the most common. On the one hand, this makes portability easy
existing descriptions and, on the other hand, the ability to learn the language
improved.

For the de nition of lexical modes the principle of the start condition scopes of
ex from version 2.5.1. With such a Scopus the corresponding
de lexical mode declared implicitly. You can specify which of them
other lexical modes he inherits.

The lexical modes are identified by variables. These are in
implemented local variables of the generated class, which are tied to whole numbers
will.

To resolve the con ict, it is possible to choose between the two strategies rst and best
be selected by compiler directives.

The method-based option is selected for the semantic actions. A
A large part of the possibilities of ex to control the machine should be a pre-
ned methods are made available.

Because the method that attaches a token to the token stream can easily be overloaded
can, there is no explicit mechanism for implementing filters. This can
but can be implemented by hand if necessary.

The token classes are identified by means of Oz atoms. This means that any
Drawings can be selected, especially those that contain punctuation marks or special characters.
There is no need to convert it into whole numbers (as is the case with Lex) by the user.
zer and the token types are accessible in the source code (even
symbolically calculable).

The concrete syntax, the interfaces and some comments about the implementation
are presented in Section 5.2.

Chapter 3

Syntactic Analysis { Parsing earth nition

This chapter develops the specification language of the tool that the parses the input. The goal is to structure the token stream to determine and to process accordingly, for example, an internal re- to build up a presentation.

The first few sections examine what exactly constitutes a parser generator. Here-
There are roughly four different aspects: 1) the de nition of the grammar, 2) the
corresponding parsing process, 3) in which way semantic values are calculated
net and 4) the form of the generated analysis program. Corresponding
the first parts of the chapter are structured.

Section 3.1 deals with the de nition of grammar, i.e. the specification of terms
le, the Nonterminale, the productions and the start symbol. For the productions are
As expressive languages as possible are desirable. It is also investigated which disparate
activities that the specification may contain and which additional information is required,
in order to be able to expose them.

According to these purely declarative aspects, it is considered how the speci cation for
Can be used to perform a parsing process. Section 3.2 gives a brief
zen insight into some of the many existing algorithms and tries to identify
what implications the choice of a parsing technique has, such as restrictions that it has
the shape of the grammar makes. In addition, not every parsing algorithm is suitable for every
suitable for use {in interactive systems, for example, backtracking is at most in
Desired to a limited extent. Possibilities are also given here as to how faulty
responsive input can be responded to.

With a grammar definition alone is only a check of the inputs
Correctness possible; in order to be able to process them further, semantic actions have to be performed
coupled with the recognition process and semantic values calculated. From-
Section 3.3 presents ways in which this can be done.

22nd

The fourth aspect is dealt with in Section 3.4. Of existing parser genes
rator generated code is considered, its interfaces examined and the form
the encapsulation assessed.

These considerations are used to describe the properties of the
Parser generator for Oz. The result will be presented later {in section 5.3 {
represents.

3.1 Specification of context-free grammars

A grammar describes a formal language, i.e. a set of sequences of symbols. To-
The following is a brief de nition for context-free grammars, indicated
rejects KVE94]. Some of the terms used have already been introduced in Section 2.1.1
been.

De nition: A context-free grammar G is de ned by a quadruple $G = (V; T; P; S)$. V denotes the vocabulary and TV denotes the set of symbols from which the sentences of the de ned formal language consist, called the terminal symbols le . $N := VnT$ stand for the remaining symbols, which are then called non-terminals . The symbol $S 2 N$ is an excellent non-terminal, called the start symbol of grammar.
P is the set of productions of the grammar. Every production from P has the form
(N; u) with $N 2 N$; $u 2 V$, also written as $N ! u$. 2

De nition: Let $G = (V; T; P; S)$ be a grammar. The derivative relation $)_G$ becomes de ned by:

$$uN v)_G uwv \text{ iff. } N ! w 2 P; \qquad \text{where } u; v; w 2 V \text{ and } N 2 N:$$

A sequence of derivatives denotes a sequence of theorems $u_1; :::; u_n 2 V$, for the $u_i)_G u_{i+1}$ applies. The formal language $L(G)$ de ned by G is given by:

$$L(G) := \{u \mid S)^+_G u \wedge u 2 T^*\}:$$

It denotes $)^+_G$ transitive Statements and from $)_G$. 2

With these designations, the job of a parser is to find a given
Sequence of terminal symbols determine whether this is a sentence by the grammar de ned formal language, and a corresponding derivation sequence to be determined.

The following sections deal with the speci cation of such grammar for practical use in compiler generators.

Page 32

3.1.1 Terminals

The input of the parser is a terminal symbol sequence. If this is pro-reduced, the terminal symbols correspond to its token types.

In general, however, any symbols or events can be used here. Example wisely there are tools that control the behavior of objects and their internal state describe by context-free grammar, such as MUSKOX does MAS96].
In this case, the terminals correspond to message types sent to an object can be.

As the formal de nition of a grammar already motivates, the terms must

nal symbols are declared. Terminals (or token types) are used internally by
represented by integers in most parser generators. This representation is
not always transparent for the user {it can be relevant, for example,
when a program for generating terminal symbols is written by hand.
Some tools allow the user to explicitly declare terminals
Assign numerical values; the others are then automatically numbered. Here oh
Lalr and Ell insist that there are no con icts with user-chosen numbers;
Yacc Joh75] and Bison DS95] assign consecutive values regardless of this.

The naming of the terminal symbols in the speci cation is also interesting. It
For reasons of readability, it makes sense to use names that are as descriptive as possible. To-
Some tools use the terminal names in error messages that appear at run time
such as PCCTS PDC91] or Bison. In Yacc and Bison up
Version 1.24 can have terminals either as single-character literals, using their ASCII code
have as a value, specify in the form '=' or name with identifiers with C syntax.
In version 1.25 the multi-character literal tokens from AUIS-Bison DSH95]
Accepted, for example, allowing ": =" instead of BECOMES.

Literal, i.e. single-element formal languages (notated verbatim by the element),
are declared implicitly by their use in many parser generators, for example
game in KVE94]. For systems in which scanner and parser speci cation are integrated,
a terminal can also be declared using a regular expression,
as in SAGA And95] or PCCTS.

3.1.2 Non-terminals

Non-terminals are the symbols used in the formation of a sequence of derivatives by others
Symbol sequences are replaced. In grammars they should be logically closed units
t correspond to the language described, such as expression, statement or
program. In MUSKOX, the context-free grammars for describing object
hold begins, correspond to non-terminal object states.

For naming the non-terminals, the identifier syntax of the target
language used. In contrast, it is usually not visible to the user of the parser
to the names of the terminals.

As in the formal de nition, one of the non-terminals must be marked as the start symbol.
net, also called axiom or sentence symbol. As with bison or ML-
Yacc TA91], done by a special directive (% start non-terminal name) or,

as with Yacc, be the first non-terminal for which a production is noted in the grammar becomes. In Eli Com96b] the start symbol is de ned as the (only) non-terminal, for that there is only one production and that it does not appear on any right-hand side.

The documentation of the ML-Yacc, which recommends not to let the start symbol appear on any right side, otherwise con icts with Parse could occur if production of the start symbol is detected.

3.1.3 Syntax rules

There are different spellings for the productions (or syntax rules). In this Section the common alternatives are presented.

The Backus-Naur form

A distinction can be made between two types of terminals, namely literals (element te single-element formal languages) and other named terminals (including generic Token GS88] or variable terminals Wad90]), as in the Backus-Naur form, or short BNF, is done. There are then two possibilities for the notation of the symbols:

> Literals are noted by simply writing them down (in print they can also add can be identified by bold print or typewriter character set). Consists the risk of confusion with meta symbols, they can be be set. To get the named terminals and non-terminals from the li
>
> To be able to distinguish teral terms, they are put in angle brackets h ::: i . (This The spelling goes back to the original BNF N + 63].)
>
> All literals must be identified by quotation marks. Then you can The named terminals and the non-terminals are named without additional metasymbols to be written. However, they must have an identifier syntax that is not Con icts with meta symbols.

The separators for the left and right side are, :: = `,, ! `and,:` common. Sometimes rules are terminated with a period or a semicolon. Blank right sides are notated by "or not at all. It should be noted that several productions are the same left side; sometimes their right sides can then become a single Rule can be summarized by separating them with ' j '

This or a similar notation is used, among other things, Yacc Joh75] and derivatives (Bison DS95], AUIS-Bison DSH95] and Bison ++ Co e93]), Happy GM96], ML-Yacc TA91]

or Gentle Sch89] is used. Also de nite clause grammars (DCGs) PW80] in Prolog are based on this.

The advantage of the BNF is that it is easy to process for the corresponding tools. is working. However, it also has some disadvantages: Speci cations are very long and thus lose legibility. Constructs that occur frequently, such as separated lists, must be used must be formulated every time {this is cumbersome and error-prone and obscures the behind list construct; it loses its recognition value and thus its Conciseness.

The extended Backus-Naur form

For these reasons, the EBNF, the extended Backus-Naur form, was introduced. the. Variants of it are used, for example, by Lalr and Ell GV92], Eli Com96b], SA-GA And95], PCCTS PDC91] and GRAMOL GS88] are used. However, it can be as their support is only patchy due to the advantages of the BNF mentioned above: So can only be used to a limited extent in Eli EBNF constructs if an abstract syntax is used which is specifically to be derived.

The EBNF is no more eighty than the BNF, but it is easier to read and, if necessary, allows a more e cient implementation. Their semantics are often determined by tracing them back to equivalents Grammars are de ned in BNF, which often corresponds to their implementation. With some Tools like the Eli mentioned above, the exact implementation is also for the user zer relevant when the relationships between abstract and concrete syntax are established will.

In EBNF, right sides of productions are nothing more than sequences of symbols seen, but as expressions. In the following, the common operators are guided; where A is a new (i.e. previously unused in grammar) non-terminal, a terminal, x, x i EBNF expressions). Only one common notation is described at a time. ben. There are many alternative concrete syntaxes for the same operators, for example wise in And95], PW80], PDC91], Com96b], KVE94], GS88] or CCH95] can be.

Sequence. The sequence corresponds to the symbol sequence in the BNF. Your operator will sometimes written as a comma, but mostly not noted at all.

Alternative. An alternative describes different right sides for the same left Page and is usually notated by, j `. So it will

x 1 j ::: y x n

replaced by A with the rules

A ! x 1
...
A ! x n

Option. One option is 0 or 1 repetition. It is here by angular Brackets, ...]. The BNF equivalent of

x]

is the fresh non-terminal A with the new rules

A ! "
A ! x

Optional repetition. Optional repetitions cause 0 to n-fold repetitions repetition of an EBNF expression. They are enclosed in curly brackets- sen. The construct

f x g

is equivalent to the fresh Nonterminal A with the rules

A ! "
A ! A x

Compulsory repetition. The mandatory repetition is 1 to n times Repetition, notated by, f ::: g + `. The construct

f x g +

is equivalent to the fresh Nonterminal A with the rules

A ! x
A ! A x

Separate repetition. By means of a separated repetition, for example comma-separated lists are described. Here this operator is through , f ::: == ::: g `, where the element to be repeated is separated from the separator separated by two dashes. The construct

f x == a g

is equivalent to the fresh Nonterminal A with the rules

A ! x
A ! Aax

Bracketing. In order to be able to take full advantage of the operators, a grouping construct, usually noted in round brackets. So can

(x)

can be transformed into BNF by replacing it with a fresh non-terminal A. with the rule

A ! x

These constructs are still very general. On- on the other hand, for some languages it can also reduce the legibility and conciseness of the speci - Improve cation if application-specific constructs can be de ned. A Technology with which this can be done are the parsing combinators Fai87], which are tional languages. There can be on a very low level with that Parsing technique: In principle, the user writes the entire parser itself, but has a set of predefined functions. Applying this Concept with C as the target language has been tried with PRECC BB92]. Another The possibility of realizing this is described in Section 5.3.3 under the name Production schemes proposed.

Structured grammars

Another suggested notation is the structured grammars. They too can translated into equivalent BNF rules. The advantages are increased legibility gen via Yacc, easier debugging of the speci cation and the implicit definition of the abstract syntax. Their components are objects and use inheritance mechanisms.

Structured grammars are de ned by the following two points:

There is exactly one production for each non-terminal.

Every production is structured, so it has one of the forms shown in Table 3.1. Let A be a non-terminal, X i , E symbols, and S be a terminal. The table shows that too EBNF equivalent of any production.

Structured grammars have several disadvantages. For one thing is the argument their greater readability to the BNF lapses because structured productions are inferior to the EBNF. The reason is that many partial expressions are named must, even more so than in the BNF. However, these rarely have a special meaning for a programmer in the language and are therefore not easy to come by with good names Mistake. Furthermore, it is concealed which non-terminals are actually important.

Page 37

construct	Structured form	EBNF equivalent
construction	A :: = X 1 ::: X n	A: X 1 ::: X n
alternative	A :: j X 1 ::: X n	A: X 1 j ::: y X n
option	A ::? E.	A: E]
Optional repetition	A :: * E	A: f E g
Optional separate repetition	A :: * ES	A: f E == S g]
Compulsory repetition	A :: + E	A: f E g +
Mandatory separate repetition	A :: + ES	A: f E == S g

Table 3.1: De nitions of the productions of structured grammars

Another disadvantage concerns the implementation of structured grammars. For your Implementation, they are transformed into BNF constructs, whereby the resulting gram under certain circumstances has properties that make it difficult to parse (so-called The con icts are dealt with in Section 3.2). So it is necessary to be (semi-) automatic Implement transformations that correct these errors. Mango offers "- Elimination, elimination of non-terminals with only one production, inlining determined ter non-terminals and expansion of non-recursive non-terminals.

The start production

This section is intended to look at some of the intricacies of what the startup production handling is concerns.

In the section on non-terminals (3.1.2) it was requested that one of them should be used as the Start symbol must be marked. But it can also be useful to use the Parsevor- start with several different symbols. For example, could be a System with integrated compiler and debugger for the compilation process only Lich accept complete programs, but single statements or when debugging Parse expressions in order to execute or evaluate them.

A known technique that makes this possible proceeds as follows: Let S 1 ; :::; S n the desired start symbols. Continue to be a 1 ; :::; a n new (i.e. previously unused) Terminal symbols and S a new non-terminal symbol. Then the following production becomes too added to the grammar:

S: a 1 S 1 y ::: y a n S n

When the parser is called, the desired start symbol S i must then be specified den, whereupon the corresponding symbol a i as the first terminal in front of the input stream is inserted.

This technique is used, for example, in the instructions for ML-Yacc TA91, Section 10.1] suggested. Multiple start symbols are rarely supported directly, for example

from GRAMOL GS88, section 5.3.1].

Another question related to the initial production does not concern the
Beginning of the parsing process, but rather its completion. Usually a new non-standard
minal S_0 with the following production added to the grammar (for example from Bison
or Lalr Gro88a]):

$S_0 : S \$$

Let S be the former start symbol and $ stand for the end marking of the input (
excellent terminal, represented in Yacc by an integer 0. In bison can
this cannot be noted due to a program error and thus remains an internal one
Terminal). So when the start production has been recognized, the end of the input stream must
be achieved.

This is not desirable behavior for all languages. With Oberon-2 MW91]
for example, after the sequence, END., which concludes a module, any
Data that can follow in the Oberon system, among other things, as command activations
or test data can function and should be ignored by the compiler. For this
For this purpose, Yacc and Bison offer the command YYACCEPT, which starts the parsing process immediately
a success message is canceled.

At ML-Yacc, the% eop directive (for end-of-parse symbols) can be used to set the amount of
Terminals can be specified which may follow the start symbol in the entry.

3.1.4 Operators and handling of ambiguities

With all the presented formalisms it is possible to construct ambiguous grammars.
ieren. The following de nition describes what this means.

De nition: A grammar is called unambiguous if for every symbol sequence from the
the formal language de ned by it gives exactly one sequence of derivation from which it arises.
If this is not the case, it is called ambiguous . 2

The problem with ambiguous grammars is that for the same input text
several syntax trees can exist, which is what the language developer and the
pilerbauer is seldom wanted. There are equivalents for many ambiguous grammars
unique grammars. However, the transformation into a unique grammar is
not trivial and often only feasible at the expense of clarity or e ciency.

Classic problems for ambiguous grammars are:

That 'dangling else'. This problem is named after an ambiguity that exists in many
Languages exist, for example in Pascal, C and Haskell, but in the language report
is often only dissolved informally (of course in the language). But it also occurs in others

Page 39

Contexts, for example in Refus EKVW94] in the de nition of a lambda
Function.

The following grammar is given for instructions with a special focus
on the if statement:

Statement: if Expression then Statement
Statement: if Expression then Statement else Statement
Statement: Others

'Others' stands for all other types of instructions. This grammar is inconsistent
Clear: For example for the sentence, if x then if y then s 1 else s 2 `makes the grammar
no statement about which if the else belongs to. The rule from speech
port usually means that every else should be associated with the last if,
that doesn't have any.

One way to express this grammar clearly is to use between
„open \ and „to distinguish closed \ if statements ASU86, p. 175]:

Statement: Matched
Statement: Unmatched
Matched: if Expression then Matched else Matched
Matched: Others
Unmatched: if Expression then Statement
Unmatched: if Expression then Matched else Unmatched

Obviously, this becomes very confusing.

Printouts. Another ambiguity arises from the syntax of expressions
the following is often described in the language report:

Expression: Expression Operator Expression
Expression: operand
Operator: +
Operator: -
...

The precedents and associativities are then given in a table.

Obviously, this grammar is ambiguous because it is used, for example, when entering
'3 + 4 5' is not de ned whether it is interpreted as '(3 + 4) 5' or as '3 + (4 5)'
shall be; both syntax trees corresponding to the brackets are possible.
The syntax tree should, however, contain the precedents and associativities of the arithmetic
Reflect expression.

Unambiguous grammars for indent expressions can be relatively easy and clear
be constructed according to the following procedure. Assess the authors of PCCTS
this is the clearest, but there are controversial opinions about it.

i	A i	o i	Associativity
0	expr	compop: = j <>	non-associative
1	sum	addop: + j -	left-associative
2	term	mulop: * j /	left-associative
3	factor	expop: ^	right-associative

Table 3.2: An exemplary operator table

Assign a non-terminal A i to each precedence level i with operators o i . (One
of which can stand for "invisible" operators, such as the application
on in the functional programming language Haskell HPJW92]. Then o i : ".)
Smaller i correspond to lower precedents.

Associativity is assigned to each level of precedence. Write the
rule

A i : A i +1

and {depending on the associative {one of the following:

non-associative: A i : A i +1 o i A i +1
left-associative: A i : A i o i A i +1
right-associative: A i : A i +1 o i A i

For example, choose the operator table from Table 3.2 and A 4 = prime. To
The following rules result from the above procedure:

expr: sum j sum compop sum
sum: term j sum addop term
term: factor j term mulop factor

factor: prim | prim expop factor

The disadvantage is that with many parsing techniques this leads to ine cient solutions.
In the case of LR parsers, for example (see Section 3.2), the effort for processing
processing of an input as the sum of shift and reduce actions.
In the C language, up to 15 reductions can be necessary for a simple assignment
{compared to a single one if the ambiguities are resolved differently.
(In addition, the memory consumption of the generated parser with this method
bigger.)

An in-depth study of precedence and associativity in speci
cation and implementation of programming languages can be found in Aas92].

Special case productions. Sometimes languages should have special uses of
Constructs are not processed by a general case but a special
experience treatment. For example, operators with constant opera-
margins that can thus be optimized, or constructs for the general

Case does not deliver the desired result. As an example for this are descriptions
of printed mathematical formulas: Is both an index for an expression
as well as a subsequent exponent given (as in TEX Knu91] by x_i ^ 2), see above
should index and exponent be on top of each other (x_{i^2}) and not standing next to each other (x_i^2).
This case can be handled via a special case productions (modified
from ASU86, p. 251]):

term: term index
term: term exponent
term: term index exponent

Obviously, there is now an ambiguous grammar. The special case product
tion should be preferred in any case in which it is applicable. Becomes
If an ambiguous grammar is required, the special case must instead be in the
semantic analysis can be explicitly queried.

The examples reveal some of the advantages of ambiguous grammars
(with a separate mechanism to resolve the ambiguities). How
these ambiguities can be handled is very much dependent on the par-
depends on the seal algorithm and is therefore only dealt with in Section 3.2.1.

3.2 Parsing Techniques

In this section some of the many existing parsing technology
sketches that analyze an input based on a grammar specification. Since the
The focus of the work is on the de nition of the speci cation language and not on its
rer implementation, it is first examined what implications the choice of
Has parsing technology. The advantages and disadvantages are mainly from the user's point of view
explains.

M ightiness of the recognized language. It is not desirable to use the full M eight-
ability of the BNF to allow {so tools should always use ambiguous grammars
reject {but most parsing techniques have additional restrictions
to the form of the grammar. For example, some algorithms have difficulties
with left-recursive productions like the following:

A: x j A x

In this case the user has to rephrase the grammar before using the tool
can work with her.

There can be a multitude of grammars for the same formal language.
These can, however, be suitable for the subsequent processing in different ways

because the shape of the syntax tree depends largely on the formulation of the
Productions. For example, it is a great relief when the process
dences and associativities of the operators of a language in the construction of the
Syntax tree have already been received. Likewise, the scoping should be
The rules of the language can be found again: The visibility area of an identifier
should correspond to one subtree as closely as possible.

It becomes clear that a tool-related reformulation of the grammar problems
me can prepare.

Resolution of ambiguities. To choose from several alternative syn-
It is very difficult to specify formal rules for tax trees. For this reason
The solutions found in existing tools depend very much on the
parsing technique used from {the ambiguities are resolved by adding the
The runtime behavior of the parser at the implementation level is influenced.

Theoretical basis. In order to be able to prove the correctness of a parser, must
a formal description from the parsing program or its specification
spelling of the recognized language (in BNF) can be derived. this is not
always easy.

Error handling options. As well as the handling of ambiguities
Finds reacting to an error in the input at a very low level
the parsing process takes place.

Powerfulness of the semantic actions. When executing a semantic activity
on, only part of all information from the syntax tree is accessible. To this
include global variables and some neighboring nodes that have already been evaluated,
sometimes information from the parent nodes. From the parsing technique used
depends on which of this information is included in the calculations of an action
can be.

For example, an S-attributed grammar has the property that all attri
bute (node annotations of the syntax tree) in a single bottom-up through
run can be evaluated. For the calculation of the attributes only
the information of the successor nodes can be used. On the other hand, it has to be with one
L-attributed grammar it is possible to combine all of its attributes in a single top
evaluate down, left-to-right sweep. So the information is out
the higher lying nodes and those lying further to the left in the parse tree.
tangible. (See Wil79].)

The power of the semantic actions is further restricted if you
the exact time of execution is not known (e.g. due to baking
tracking or certain transformations of the grammar). In this case you can
no side effects are used to control the scanner (so-called lexical
tie-ins DS95]).

Page 43

E ciency. As was motivated in the section on ambiguity, can
the e ciency of different parsing techniques differ solely in how the
Grammar is implemented (in addition to the complexity
analyzes). For example, the repeat constructs of EBNF can be
can be e ciently mapped directly, with others it has to be done by BNF
Equivalents to be replaced. Thereupon it can no longer be of its known structure
be bene ted.

Debugging. The debugging process inherently runs at a very low level. De-
Here bugging includes both the debugging of the grammar specification (e.g.
wise the elimination of ambiguities) as well as the user-written
semantic actions. In the case of different procedures, this has to be done in other places
be set.

LL parsing

LL (k) parsing refers to a top-down parsing technique that involves input from left to right reads right and forms an inverse left derivative. (A left derivative is a derivative sequence in which the leftmost non-terminal is always replaced.) The Value k indicates how many terminals are used as a so-called lookahead, that is, read in advance and included in parse decisions. Mostly is k = 1, but there are arguments to support larger values for k PQ95].

LL (k) parsing is a special recursive descent parsing algorithm that does not have a back-tracking required (i.e. a predictive parser). For this, however, the grammar must determine Sufficient criteria: There must be no (direct or indirect) left recursion and two alternative productions each for a non-terminal may not derive a symbol sequence, that start with the same terminal. Therefore, time-consuming reformulation may be necessary until the generator accepts the speci cation, which leads to unnatural grammars can. This can be remedied by semantic predicates (see Section 3.2.1).

Semantic actions can be carried out at any point and rule-local variables variables (which are reallocated each time a production is called) are de ned. L-attributed grammars can be implemented.

LL parsers are easy to implement and EBNF constructs can be easily and effectively be implemented efficiently. For example, separated repetitions are more efficient than with most other techniques. But there are only a few possibilities for ambiguity to dissolve (see section 3.2.1). In particular, operator precedences can only be through Formulate it or an embedded parser that uses a different technique to be set.

As an argument for LL parsers is often given, since these are easy to debug, because the generated code looks very similar to the input grammar: For every non-terminal a function is generated; Sequences are converted into sequences, alternatives

and options in conditional statements, repetitions in loops. Actually should Parser, however, not at the level of the generated program code, but at the specification level be debugged. Also, this argument nullifies the reason a tool to use to generate the parser: If the code of the grammar is like this anyway similarly, it could have been written by hand.

LR parsing

So-called LR parsing is a bottom-up parsing technique without backtracking that ubli-

is usually implemented as a table parser (one of the few hard-coded variants
is described in BP95]). The actual parsing algorithm can, depending on the table con
instruction procedure (SLR (k), LALR (k) or LR (k), where k is again for the number of
Lookahead terminals) recognize a more or less large class of grammars.
With LALR (k) and LR (k) this is larger than with the corresponding LL (k), which is often
more legible and natural grammar formulations are permitted.

The grammar is transformed into the description of a push-down automaton.
One of the actions is based on the current status and the lookahead terminals
Shift or Reduce selected. With Shift (s) a lookahead terminal is set to the parse
stack pushed and the machine changes to the state s. With Reduce (n) is with
of the (BNF) production with number n reduced: There are so many symbols from the Parse-
stack replaced by the non-terminal symbol, as the right side of the production contains n.
It follows from this that EBNF can only be converted into an equivalent BNF representation
can be supported. This parsing technique can hardly be used for a given grammar
Implemented by hand {a tool is essential here.

There are restrictions on the grammar with this parsing technique because there are not
Simultaneously a shift and a reduce action or two different reduce actions
may be applicable. There are several methods of finding such con icts. In-
special operator precedences can be elegantly included (more on this
can be found in Section 3.2.1).

Finding the causes of ambiguous grammars often turns out to be very
expensive, since most tools only dump the generated machine as an aid
to offer. If desired, ML-Yacc can provide complete (but
do not generate more eight) LR (1) tables, which make debugging the machine easier
should do. Lalr supports the even more helpful output of a derivation tree
where the emergence of the ambiguity is easier to recognize.

Semantic actions can only be coupled with Reduce (n) actions, which
indicates that in the generated parser there is an extensive case distinction about the value
is made by n. To debug the semantic actions, a single break-
point to this statement. Only S-attributed grammars are easy to implement. Ver
inherited attributes, i.e. those that come from nodes that are further left in the syntax tree
can be simulated to a limited extent by not only inserting the content of the parsestack in

with reference to the current production, but also to elements below.
is pulled. Attributes can therefore be read, their relative position in the parsestack
is constant and known. This is exemplified by Yacc and derivatives by negative
Array indexes allowed. A check as to whether this reference is correct (correct index,
Conversion to the correct type), however, none of the examined tools offers.

Operator precedence parsing

Operator precedence parsing is a not very effective and not very widespread technique that
but for the implementation of grammars for In x expressions simple and e cient
is applicable. However, it may be that the language actually recognized is derived from the speci
tion can only be formally derived with difficulty. Hence, this procedure is used at best
Implementation of embedded parsers used and also rarely offered by tools.

De nite Clause Grammars

De nite Clause Grammars denote an extension of Prolog, with which BNF-Pro-
ductions can be speci ed. They are implemented directly in Prolog clauses.
Therefore, they use backtracking with all the disadvantages that come with it:

> Interactive systems are not possible, as these can be used quickly and straight away.
> It must be determined which production is used to parse the input
> must be (e.g. user interfaces or object protocols).
>
> The generated parsers are very ine cient if the grammar is poorly formulated.
> is lated.
>
> Backtracking parsers often produce very bad error messages.

The backtracking by the „ ! \ Instruction can be restricted by which all decisions
Calls that have been used up to that point are considered to be de nitive. This allows some
these disadvantages are eliminated.

Just because of their implementation, these grammars are very close to the target language Prolog
bound. They are mainly useful when there is no backtracking
can be dispensed with.

Combinator parsing

Combinator parsing is a parsing technique from the field of functional program
mier languages. It is a set of functions that the standard operat-
Implementing ren for the description and recognition of EBNF grammars. Since no
Backtracking is possible, a list of the possible parse results is always kept,

with alternatives removed according to input and according to grammar
to be added. In the end, exactly one element is left if the entry is correct
and the grammar is clear. This technique was carried out with the tool PRECC BB92]
suggested for target language C.

The implementation of EBNF operators through functions allows the user to
to de ne gene operators that can be arbitrarily complex. The target language can
when used in their full power and any expressions (and partial parsers)
can be used as arguments. The price for this flexibility is that the actually
Known language can under certain circumstances be formally very difficult to derive. Furthermore you can
Parsers are written whose termination is not secured {in general, auto-
Matic analysis of the grammar is only possible if strict guidelines are adhered to. The
Technology is also moderately to very ine cient. The advantage, however, is that there are
all formal languages can be recognized BB92].

However, this completeness can also have a negative impact on the language design
to have. The PRECC authors themselves give an example of legal but unclear ones
Syntax of your own tool: For formulations like @ a =) foo (TRUE) (b) (bar) (
must be determined whether it is

a =) foo (TRUE) (b) (bar) (

or to

a =) foo (TRUE) (b) (bar) (

acts, since both the brackets through (...) and those through) ... (one meaning
Has. This syntax is difficult to parse, not just for programs. Tools that speak the language
A little more restrictions can prevent such syntaxes from arising.

Summary

Many parsing techniques impose restrictions on the grammar that are not always included
are compatible with the desired properties of the syntax tree. For example, has
a left factorization of the grammar, which is required for LL parsers, results because
different constructs of language can hardly be distinguished. Becomes
an abstract automatically generated from the concrete syntax, this distortion
zahnung through to semantic analysis and possibly even code generation.
One solution is to use an eighty parsing technique that uses the full BNF
can implement {with the corresponding disadvantages. But it is more elegant from the
To abstract parsing technique more. Since many many transformations are based on grammars
which do not change the recognized language, the user should use the
Specify grammar at a high level and leave it to the tool
to bring them into a parsable form. These transformations must be made for the user

be transparent. In particular, references in semantic actions must be automatic
be adapted {for the user the concrete structure of the syntax tree
unimportant since the semantic actions relate to the speci ed form. A
Tool that takes a step in this direction is Mango Age94].

3.2.1 Resolving conflicts

The purpose of this section is to provide solutions to how grammars with discrepancies
clearances can be implemented. In the first part, static resolutions are
of con icts, i.e. those that were already carried out during the parser generation.
be guided. A second part describes possibilities, only at runtime {possibly
context-dependent {to make parse decisions. The examples from Section 3.1.4 are
the one picked up here again.

Static resolution of con icts

In LR parsing, ambiguities are well studied. The following options were available for
existing tools found:

Operator precedence and associativity. Yacc, SAGA and Lalr, among others, offer
this procedure, which is used to eliminate typical ambiguities when parsing in-
to eliminate x-expressions. For this purpose the following (about the BNF productions
additional) prerequisites required:

Each terminal has a maximum of one precedence 2 IN and one associativity 2
f non-associative; left-associative; right-associative assigned to g . (High price
zen correspond to stronger ties.)

A maximum of 2 IN precedence is assigned to each production . (As default
that of the last terminal to appear in production is often used.)

If a con ict occurs, the decision as to whether a shift or a reduce
Action is to be performed by comparing the precedence of the lookahead
Terminals met with the production. If the values are the same, then is based on
the associativity decided DS95].

Expressions can then be derived from the rule, for example

Expression: Expression Operator Expression

can be parsed by specifying an operator table as in Table 3.2.

This procedure can also be used to solve the dangling else explained above. Be the following rules apply:

Statement: if Expression then Statement
Statement: if Expression then Statement else Statement
Statement: Others

Now else has a higher precedence than then, the first rule the precedence of then and the second that of else. Then the ambiguity is correctly resolved {in case of doubt the else is shifted.

Shift annotations. In the event of a shift / reduce con ict in SAGA, the user can ex- explicitly request that a terminal be shifted by appending $ shift to it:

Statement: if Expression then Statement
Statement: if Expression then Statement else $ shift Statement
Statement: Others

This annotation can only be implemented for this parsing technique.

Modi cation of the exact right context. This procedure can be found, for example, in Eli. In the event of a con ict, certain symbols from the set of Terminals that are allowed to follow in the input during the reduction with a production, removed. For example, the dangling else can be solved as follows:

Statement: if Expression then Statement $ else
Statement: if Expression then Statement else Statement
Statement: Others

The $ else in the first production means here that there {if with this production should be reduced {the Loohahead terminal must not be an else.

The disadvantage of all of these methods is that there is a BNF version of the recognized language under Circumstances difficult to derive formally. The user must register the generated Look at the machines and check exactly how the con icts were resolved.

There are far fewer options for LL parsing. The Ell Gro89 tool, Section 3] uses the following standard set of rules: If there is a con ict between a Option or a repetition and the subsequent terminal before the option or the repetition is selected. Is there a conflict between two alternatives? the first one noted is selected. Arise here „dead \ branches in the grammar, that are never used, an error is reported.

For combinator parsing, Hil94] investigated the question of how special combinators for the implementation of operator precedences can be implemented. The The operator table is then transformed into these combiners.

Dynamic resolution of con icts

If con icts are only resolved when they are carried out, the decision of con icts can be
text information can be made dependent. For example, user-de ned
Operators are handled as suggested in PVGK93]. The one described there
Algorithm is based on LL and LR parsers and can handle languages with dynamic
parse de nable operators that have the following properties:

- Any identifier can name an operator. You may also be able to
predefined operators (such as +, -) can be newly de ned.
- The syntactic properties of an operator change in the course of the parse
process.
- Operators can be passed as arguments to other operators. Consequently
is for example the sentence „_ ^ \ legal if „_ \, „ \ and „ ^ \ In x operators
represent.
- Operators can be overloaded, i.e. the same symbol for several syntactically
different operators can be used. For example is „- \ often at the same time
Pr a x and In x operators.

The only thing that is required in addition to the normal speci cation is a declaration
on, which symbols represent operators. Every shift / reduce con ict into the dynamic one
Operators are involved, is then converted into a so-called Resolve action,
which makes its decision based on the runtime operator table.

With LL parsers there is also the option of de ne semantic predicates, such as
they are supported, for example, by PCCTS Par95, Section 2.6.1]. This means,
because Boolean expressions are embedded in the grammar, which are expressed at runtime.
be evaluated. If they deliver the truth value incorrectly, the associated branch becomes the
Grammar not chosen. So decisions are not just based on lookahead
possible, but also realizable depending on the context.

PCCTS offers another construct, the so-called syntactic pr adika-
te Par95, Section 2.6.2]. Special case productions as well as ambiguous
for example in C ++ where declarations of initialized variables are resolved
cannot always be distinguished from functional prototypes. The solution leads
targets restricted backtracking in an LL parser.

3.2.2 Error handling

The error handling deals with the behavior of the parser when the input does not correspond to the grammar, i.e. if the lookahead terminals do not parse action is associated (or even if a non-associative operator is used twice in a row

is applied). Such errors should be recognized as early as possible in order to to be able to deliver this error message. It should also be possible to determine which Productions describe the further input, so in a well-de ned internal State found and the parsing process can be resumed. In the- This section is only intended to give a brief overview of the possibilities, such as this can be proceeded.

Error handling includes the following {structured {tasks:

Reporting. This describes the subtask of reporting parse errors that have occurred. Additional information can be provided:

the coordinates of the lookahead token in the source code,

the text of the line in which the error was detected, with a pointer to the current characters,

the lookahead terminal and

the possible (expected) terminals. These can still be used in „Classes \ group be pated as PCCTS allows: For example, if a class is de ned „Number \: = f Integer; Real g , you get instead of the message „Integer or real expects \ then „Number expected \. If the tool has the (un- natural) naming of literals by means of identifiers in this way also token names are replaced by the corresponding character strings be, for example, by de nition of a class „! = \: = f NOTEQUAL g .

Recovery. Error recovery refers to the attempt to take over so many input terminals. jump until parsing continues in a de ned internal state can. This phase does not need to be implemented if after the first one Input error should be canceled.

The parse process can be continued at so-called restart points. Either the tool determines these from the grammar, such as Ell and Lalr, or they are the specified explicitly by the specifier, as is context-dependent in Yacc the reserved terminal error is possible. For example, could be in Modula-2 at a parse error in a procedure starting from the next semicolon in expectation of a Statement to be parsed further.

If the semantic actions allow side e ects, it is problematic again

to return to a de ned state. When using Yacc step
easy memory leaks when skipping terminals, if explicitly
Carrying allocated values. AUIS-Bison supports the release of this in the event of an error
Values. This difficulty arises if the target language offers a garbage collector
not on (for example with JavaCup Hud96] for Java).

Repair. Some tools try to correct the input instead of part of the input
be discarded. An attempt is made to use a minimal number of insertions to

Deletions or replacements of terminals to construct a correct input
Ren. Either only the lookahead terminal is modified (as with Eli) or it
the last n terminals are also included (e.g. 15 terminals in
ML-Yacc). So that the values of the newly created terminals are not unde ned, can
a default value can be specified for each token type.

To control the possible modi cations, a list of the key words can be created in ML-Yacc.
words that are risky to correct, as well as a list of
Terminals to be preferred for insertions or replacements. Eli allowed
Furthermore, the specification of the permitted compounding constructs (for example begin
... end, „(:::) \). A repair is considered successful if in the next n terms
If no further error occurs (Eli: n = 4). The semantic actions are allowed
This type of error correction does not have any side e ects or has to be reversed
can be done (backtracking).

With insertions it is important to guarantee termination. The advantage of
driving is because the subsequent compiler phases always a consistent input
received, since syntactically incorrect entries are not possible at all.

With interactive systems in particular, it is essential to intercept errors and continue to parse them, because the internal status is lost with a simple abort.

3.3 Semantic values and semantic actions

In order to be able to combine the recognition of the input with calculations, for all
Terminal and non-terminal slots provided for so-called semantic values. In sta-
Table-typed target languages still have to be assigned types to these values. The
The values of the terminals are {if a scanner is used {the token values; the who-
te of the non-terminals must be calculated (attributed). This section will
investigates how this task can be done.

A simple possibility is the automatic construction of concrete or abstract
the syntax trees during the parsing process, which only appear in a second run.
be animalized. ML-Yacc, for example, delays the execution of the semantic actions
if these have side e ects until the parsing is complete so that there are none
There are problems correcting errors. The disadvantage of this is that there is no context information
functions can be used to resolve ambiguities in scanners or parsers,
for example the typedef problem of C and C ++ or the ambiguity of desi-
gnators between qualified identifiers and record field selection in Oberon-2. A
KVE94] suggests restricted inclusion of context dependencies]: Here can
at most the special action can be coupled with a reduction, since a single
A new token type is assigned to a lexeme. A scanner that supports this will be
in Kuh94].

For this reason, semantic actions are often written directly into the syntax rules
embedded. These are to be carried out exactly in the order in which the entries are made
derive the rules of grammar. As with the discussion of the parsing techniques
was mentioned in section 3.2, these actions are only allowed at certain points in the
Grammar and parsing con icts can be introduced with their insertion.

Probably the most interesting aspect of the semantic actions is the question of which
Non-terminal values you can access and how. For the calculation are of course exact
those attributes can be used whose calculation has already been completed and those from
the action can be referenced. Appends the calculation rule for an attribute
depends exclusively on the descendants of a node, so the attribute is called synthesized;
if it is calculated from values that come from parent or sibling nodes, then it is called
it inherits ASU86, p. 280].

With Yacc, the values of previous symbols are referred to via their relative position in the
(LR-) parsestack accessed. This access to indices has the disadvantage that the mo-
di cation of productions all references in their semantic actions after-
must be tested; this is easily forgotten. This gets worse with the
Use of inherited attributes that can only be used without checking for correct referencing
is possible. For example, in a grammar for C-like declarations
inherit the type name, as in the following Yacc source code:

```
declaration: type declaredVariables ';';
declaredVariables: declaredVariable;
declaredVariables: declaredVariables ',' mark declaredVariable;
declaredVariable: Variable {enterVarDeclaration ($ 1, $ <Type> 0); };
mark: {$$ = $ <Type> -2; };
```

The indices $ 0 and $ -2 can be used to access previous elements in the parsestack. to grab. Here, however, no legality check is carried out, and neither are the casts in the corresponding type (<type>) are not checked for correctness. So that the value of the non-terminal "type" has a constant relative position in the parsestack, it must be from the so-called marker non-terminal "mark".

If one also wants to allow EBNF constructs, the linear designation unclear by indices, as in Lalr. A moderate improvement is the two-step ge designation by $ scope.position. So it seems essential, the non-terminal values to be denoted by name. With Ell, these names are assigned automatically; in the rule

```
term: fact ('*' fact j '/' fact)
```

For example, the values can be accessed under the names "term0", "fact1", "fact2" and "fact3". bar. An individual naming allows for example JavaCup Hud96], which has an LR-

Parser generated, but no inherited attributes can be named. Names are assigned here by the construct expr: e1.

The most comfortable solution is offered by LL parser generators by specifying for- paint and current parameter lists, which also specifies easily inherited attributes can be. For example, the type inheritance on the example above would be in PCCTS formulated as follows:

```
declaration: << Type t; >> type> t] declaredVariables t] ';';
declaredVariables Type t]: declaredVariable t];
declaredVariables Type t]: declaredVariables t] ','
                                        declaredVariable t];
declaredVariable Type t]: << Variable v; >> Variable> v]
                                    << enterVarDeclaration (v, t); >>;
```

If you choose a spelling that is closer to Algol, the legibility is also better.

Another question is whether the tool supports rule-local variables. In Yacc local variables can only be implemented by a software stack. PCCTS however, as in the last example, it allows so-called init actions for allocation and initialization of local variables on the hardware stack. In general, local Variables are difficult to handle in table parsers, but in a hard-coded parser can be easily implemented.

3.4 The generated program

The most important property of the generated analysis program is its encapsulation. This strongly depends on the possibilities of the target language for program structuring. The following solutions can be found in the common tools:

Program based. In Yacc, the global symbols used for the parse functions are set on etc. Because of this, it is not possible to generate more than one Use parsers in a program; at the latest in the link phase there are symbols bolkon icts.

File based. The bison, for example, falls into this category. What makes him unis the possibility to use the standard pr ax for the prede ned global To change symbols (yy-). However, these renaming are cumbersome: they require a detailed knowledge of all global symbols of all programs in which which parsers should be used. (This is a basic problem of C.)

Module-based. The generator Ell for Modula-2 creates its own module in which the Parser l opens. For this reason, the coexistence of several generated parsers is in the same program nothing in the way.

Object or class based. Mango for the programming language Self, Bison ++ for C ++ and JavaCup for Java create objects or classes that use the parimplement ser. The ability to de nition multiple parsers in the same programs are the same as in the module-based solution; this procedure offers but the additional advantage that reentrant parsers can be implemented. Per input strom an object can be cloned (Self) or a class can be instantiated (C ++, Java).

Another advantage of the class-based variant is the behavior of the parser determined by inheritance and overloading when productions are recognized can. For example, the same grammar specification could be used to add an interpreter, a compiler and a pretty printer for a language to implement. The only generator that has a mention of this possibility was found was PCCTS. There this procedure is called the application of trigger functions.

With some generators it is not clear from the documentation which restrictions the generated program is incumbent. It can be assumed that relatively often the program

based solution is used.

3.5 Design decisions

After examining existing tools, you can make decisions regarding
the specification language of the tool to be developed must be met. Orientate you
also comply with the requirements from Section 1.4. The properties of the parser generator
are:

The user should not be concerned about the implementation of the terminals in an internal re-
presentation must take care of. For this reason, they are not represented as whole numbers.
posed, but as atoms. This means that meaningful names are also possible (such as ': =').
This corresponds to the representation already given in Section 2.3 for the scanner generation.
rator was elected. One-character literals correspond to atoms of length 1; for this
the whole numbers between 1 and 255 are also accepted. One-character literals
are declared implicitly, all other terminals must be declared explicitly.

The de nition of several start symbols is supported. Nonterminals can
denoted by atoms and variables. According to the distinction
between public and private methods in Oz, the atoms are the starting symbols.
The desired start symbol is displayed by specifying its name each time it is activated.
selection of the parser selected.

The user can create his own as well as application-specific EBNF constructs
de kidneys. The mechanism for this is the production scheme: become operators

Page 55

replaced by new expressions when they are used, with the arguments corresponding
be substituted accordingly. Variables are clearly renamed. New Re-
rules can thus be introduced. Common EBNF notations are given by
Predefined production schemes made available in the Prelude.

After the expansion of production schemes, some simplifying and
Optimizing transformations applied to the grammar. Together with the
Possibility to de ne your own production schemes can easily differ from the actual
parsing algorithm used can be abstracted.

Although LALR (1) is selected as the parsing algorithm, the rest of the design was
de, however, paid attention to the dependencies on this in the speci cation language
to be kept as low as possible. The reasons for this choice were the following properties:

{ LALR (1) is more eighty than LL (1) and thus leads to more natural grammar specifications. LL (1) is a real special case of LALR (1).

{ LALR (1) is a very popular parsing algorithm. This makes the portable existing grammar descriptions for tools like Yacc. fold.

{ Operator precedences can be implemented directly.

{ The algorithm can easily be extended to include dynamic operator tables.

Precedence and associativity information can be used with the explicit declaration can be specified by terminals. Rules are precedent by the reserved fourth non-terminal prec assigned with a positive integer as an argument; In the case of simplifications, this is treated like a non-terminal.

For fully compositional languages for which the parser generator is designed, a Good recovery in the event of an error is difficult because there are few suitable restart points gives. For this reason, no complex error correction method is sought. There there are error handling options that have little or no additional information practice in the speci cation (which actually have nothing to do with the language speci cation, but only have to do with the runtime behavior), such a drive to be assumed.

Values from non-terminals and terminals are accessed using variable names. fen. Non-terminals can have any number of partial values. The syntax for Nonterminal de nitions and applications of grammar symbols is the method application very similar to dung; in particular, there is no distinction between synthesized and inherited Differentiated attributes. This information is obtained from the use of the attribute te derived; thus, correct references to inherited attributes can be verified possible (marker non-terminals are inserted automatically).

When the parser is called, a nonterminal application is transferred as a record with ben, which in addition to the start symbol, the initial values of all inherited attributes of the Start production speci ed. The synthesized attributes are processed after the Parse process unified with the corresponding subtrees of the record.

Local variables are supported in rules. These are not on the hardware stack, as with hard-coded parsers, but on the already existing sting software stack {the parsestack. All in nested EBNF expressions

variables used have a constant relative position in the parsestack, therefore can be accessed without additional marker non-terminals. Consequently do not restrict the class of recognizable languages of the algorithm.

The generated program should {according to the already de ned scanner generator {be class-based. The coupling of scanner and parser takes place through De nition of both parts in the same grammar or by de nition in different grammars and inheritance.

Concrete syntax, interfaces and implementation are presented in Section 5.3.

Chapter 4

Extensions

This chapter examines which other tasks of a front end are carried out by
witnesses can be supported. The phases already considered are used here.
builds.

Section 4.1 deals with a special form of tree transformations, n am-
Lich the so-called replacement rules. You eliminate certain sentences of the language by adding
they replace them with equivalent constructs of the language that are easier for the compiler
than the original input are to be handled.

Section 4.2 looks for possibilities that are common to all lexically copied languages.
to automate the analysis of the variable bindings, which runs in the same way. Simultaneously
the unambiguous renaming of all bound names, already motivated in section 1.2
Draftsman are carried out.

In many applications the only result of structural analysis is internal
Tree representation of the input. Section 4.3 examines how deriving and instantiating
suitable structures are created by the tool instead of explicitly provided by the user.
can be taken.

Are the relations between the concrete and the abstract syntax formally described,
in this way, the tool can convert the abstract representation of the input back into a concrete one
translate. This is the subject of investigation in Section 4.4. This is in applications
genes that transform programs into others in the same language (for example, into a
Reduce core language), of importance.

The last section (4.5) deals with an extension that, strictly speaking, is not
should support further compiler phases, but is relevant for the design. Here will
examines how the entire speci cation for a tool that includes one or more of the
sub-tasks under consideration are supported and can be modularized.

The structural analysis from the previous chapters is a prerequisite for each
the extensions described here. It will each be highlighted like this
to interact. At the end of each section, the design decisions are made as to whether and how

the phase under consideration should be implemented in the tool to be developed, presents.

4.1 Substitution rules

The Oz programming language is de ned in several layers. The lowest level is Kernel Oz Smo94]. This core language contains all the concepts of Oz. Notation Hen95], which for common programming paradigms has expressive constructs offers. For example, in the core language there is no elsecase that instead goes through nested case statements must be implemented. The Oz notation gives for one Substitution rule like the following:

```
case EL 1                    case EL 1
elsecase EL 2          )     else
...                            case EL 2
                               ...
end
                               end
                             end
```

Languages de ned in this way should receive special support in their translation Experienced.

The advantages that are expected of this are presented in Section 4.1.1. At the same time it is explained which properties a tool that is used for the concrete project blem should offer support, must have. Section 4.1.2 examines existing ones Tools on what approaches they have to contribute to solving this task. The The resulting solution is outlined in Section 4.1.3.

4.1.1 Goals

The following points explain the advantages of using substitution rules for the compiler builder. The necessary prerequisites are met in each case that are made to the tool.

1. If it is possible to write down substitution rules directly, a better ensuring traceability between the de ning document and the implementation can be performed than would be possible if formulated by hand. That makes the implementation is less error-prone and improves maintainability.

2. Replacement rules promote the gradual program development. In a first Step only needs to be written a compiler for a core language. Four le constructs can first be made ine cient, but semantic, using replacement rules

be translated correctly. Only in later refinements are e cient implementations added mentions for these constructs.

3. For the notation of the substitution rules, instead of an abstract syntax, can the con- the specific syntax of the language to be implemented is used, the replacement Rules of speech themselves are invariant to changes in the de nition of the abstract syntax.

4. Under the same conditions as in point 3, the parse process can be performed with others Actions are coupled with the otherwise necessary construction of the abstract syntax tree.

5. Under the same conditions as under point 3, a representation does not Less to be formally documented: The abstract syntax is not required before and to be described after the simplifying transformations. Furthermore can under certain circumstances a more skilful (simpler) abstract syntax can be chosen, if there are no separate node types for constructs that are eliminated anyway need to be provided.

6. If a special notation is provided for replacement rules, a static check can be made whether the transformation is correct, because only correct (according to the syntax relevant) program fragments are accepted.

7. Can the substitutions coupled with the structural analysis be speci ed and are carried out, changes to the notation remain very local, namely on the parser speci cation is limited. You need each other in the rest of the front end Not even aware of the constructs that are eliminated by the substitution rules to be. Of course, this only applies as long as the basic language remains unchanged.

8. Under the same conditions as in point 7, interactive systems are to be implemented. because the replacements are not carried out after the parsing process is complete will.

9. Under the same condition as in point 7, replacements can be more e cient implemented, since the abstract syntax tree has the same definitive form can be built. It is not a separate complete run-through of the abstract Syntax tree is necessary because the reduction with certain productions results in the Applicability of substitution rules can be inferred directly.

After this preliminary examination, the requirements made are repeated again summarized:

A special notation for replacement rules, similar to the intuitive notation should be made available (specification by example).

The specification of the replacement rules should not be based on the abstract, but only depends on the specific syntax.

The replacements should be carried out coupled with the analysis process, so that the result of parsing is no different from that if directly the Normal form (i.e. the maximally simplified form) of the input has been edited would.

At this point, the connections and differences between replacement rules as described here and term rewriting systems Ave95] will. Formally, substitution rules are also de nitions of equivalences on terms, so that As with term rewriting systems, the termination is important for the implementer; if necessary, he must also guarantee the Church-Rosser property. Must in practice these properties, however, for replacement rules {because of their undecidability {not be checked by the tool.

In contrast to the term rewriting systems, whose rules are asentation (i.e. an abstract syntax), the concrete syntax is to be taken as a basis here will. In addition, it is not required that replacement rules have the full power of Have term replacement systems: Only replacement rules should be supported that Its applicability can be deduced from the reduction with a certain production can. (In particular, the parsing technique used can make the generality of the Restrict substitution rules.) Fulfill most substitution rules in Oz notation this criterion, so it is not a major limitation.

4.1.2 Approaches

This section examines existing mechanisms by which replacement have rules of language formulated. It is highlighted which of the above described Requirements of the respective approach are met.

Most of the languages designed specifically for rule-based transformations were (like Gentle Sch89], Sch], Rigal Aug93] or puma Gro91a], Gro91b]), work at the level of an abstract syntax. In principle, the only excellent consortium is structure that they offer for the formulation of replacement rules, the pattern matching with uni cation. In addition, the recursive descent with the rule applications usually has to be from Be formulated by hand.

The language TXL CCH95], on the other hand, takes over the recursive descent itself. There the rules specified by the user are applied after the parsing process is completed. det. Several types of rules can be de ned:

Transformation functions that are a replacement at the top level of the term to which they are applied (in the tools mentioned above this is the only type of rule offered),

Search functions that search the tree (leftmost-innermost) and the first possible Carry out a replacement,

> Transformation rules that search the tree and replace at each point
> Carry out where applicable. A substitution can be turned into a match
> for the same rule.

The desired replacement rules correspond to the transformation rules. This
but do not offer all the required properties: The applicability of the rules cannot
be statically resolved (i.e. with a reduction with certain productions
be coupled), so the tree must be searched dynamically, and this into the
most cases even several times {the patterns are not included in a single comparison
tree mixed, but strictly sequential in the manner specified by the user
applied.

The Kimwitu Term Processor vEB93] offers a different approach. This one is very close
on the reduction systems: A reduction system is de ned from which data types
as well as rewrite and output functions for terms for the programming language C.
becomes. These can then be used in programs to produce a term of
Form normal form. The rules correspond to the transformation rules of TXL, but
the applicability of all rules is tested at the same time.

The most promising approach is that from CMA94]. There are the replacement rules
specified as semantic actions in the grammar, i.e. especially interlinked with
applied to the analysis. For this reason, they are also potentially associated with other semantic
actions can only be linked to the structure of an abstract syntax: For node con-
instructions, the user must provide functions that also perform other actions at the same time
can execute. The replacement rules are formulated in the concrete syntax, in-
which a nonterminal is specified with which the right side is to be parsed. There
corresponding semantic actions are generated at compile time that use the same radio
Execute functions as if the simplified form had been entered directly. The disadvantage
of this is that a given lexical structure for the conversion of the concrete
syntax in a token stream is assumed (due to the ASCII limitation
are placeholders {metavariables {only with additional requirements of identifiers of the
Language itself distinguishable).

4.1.3 Design decisions

The solution that was implemented for the tool to be developed is based on the approach
from CMA94]. This was modified to have all the desired properties
Offer. The result looks like this:

Replacement rules may be used wherever semantic ac-
are permitted. They are converted into semantic actions when generating the parser
translated. For this reason, they are also linked to the analysis process
applied.

The production, with the reduction of which the substitution rule is coupled, delivers the same-
early their left side. This connection is made possible by its placement in the grammar
tik made. The procedure is equivalent to a pattern matching process,
but at the level of the concrete syntax. Subsequent ine cient searching
of the abstract syntax tree according to the places where rules are applied
can do that, not applicable. The restrictions imposed by the parsing algorithm
the class of substitution rules that can be used are easy with this approach
detectable.

The right side of the replacement rule is determined by the non-terminal with which it is parsed
should be given, as well as a series of applications. Each of these is called a
Tuple represents: The label indicates the token type or the nonterminal
the features the parameters. So the right side is only from the concrete one
Syntax dependent, but decoupled from lexical conventions (although not
the representation of the tokens). However, legibility is compromised as a result.
The token types must be statically fixed; for the token values remain complex
Calculations possible.

The concrete syntax and the solution are shown in detail together with some examples.
Section 5.4 presented in detail.

4.2 Bond Analysis

The binding analysis is the process that generates the corresponding de-
nition determined. This task is part of every compiler, since the information it provides
both for semantic analysis (checking the correctness of references,
Type checking) as well as for code generation (memory / register access)
is required.

The link analysis is usually taken over by the symbol table management.
men. A lookup function returns the corresponding variable name for a variable name
Entry. This component can uniquely redesign all linked identifiers at the same time
name, as the information required for this is available to you anyway. This

can possibly avoid the implementation of tree transformations
simplify name clashes, such as inlining procedure calls, instan-
ciation of generic modules or expansion of type de nitions. Furthermore could
they generate fresh variables (i.e. not used in the source text) if necessary.

Section 4.2.1 explains the advantages that an automation of the
analysis can bring with it, and the properties required for this are identified
is working. Section 4.2.2 presents some approaches from the literature. Finally, in
Section 4.2.3 the design decisions for the tool to be developed in relation to
met the bond analysis.

4.2.1 Goals

First of all, some programming languages are to be examined representative
how identifier bindings look like with them. These languages are all based on the lexical
skoping on, which means because the de nition to which each identifier refers
in the source text, can be determined at translation time. Use in addition
most languages structured in nested visibility areas (blocks),
nor the most closely nested scheme. The description referenced by a variable is
nition always sought in the surrounding blocks. Do variable endings exist with
the name sought in several surrounding blocks, then the one from the innermost
Block the valid one.

If this simple model is assumed, it is easy to deduce which one
Know a tool that automates the bond analysis about the ones to be translated
Language must have: On the one hand, it must be speci ed which constructs a block in the
represent the above sense, on the other hand, the variable occurrence must be classified,
by distinguishing between binding and referencing occurrences.

A few examples of constructs from existing programming languages will now be given
that go beyond the procedure described above. (Should
technical subtleties are ignored.)

> The programming language Pascal JW75] offers with ... do a construct, da
> introduces implicit declarations in a block: The fields of a record-valued variable
> variables can be accessed in this block without specifying the variable name.
>
> Among other things, in Oberon-2 there are predefined global identifiers (without, because they
> was previously imported from another module). The bond analysis
> so does not start with an empty symbol table.

Symbols can be exported in Oberon-2. Information about this will be
stored in symbol files created for each module. The one when renaming
Any substitution of the global visibility area must be explicitly processed here
will.

For example with the modular concept of Ada MW91] it is possible to have all (visible
ren) to import de nitions of another module without qualification, which means
since they can be accessed under their name without being identified by their module name.
to be drawn. Identifiers are declared implicitly without giving their name
must appear bindingly in the source code of the importing module.

In W-Lisp (Version 3) WKE96], variables become implicit through their use
declared if they were previously unknown. Furthermore, variables are not allowed in
nested blocks are covered.

In Oz, variables that designate private features, attributes or methods can be
in the class to which they belong, can also be used before their de nition.

So a formalism has to be found that is powerful enough to
to specify calculations from the simplest model of lexical mapping.

The following properties are provided by automatic support for the binding analysis
expected:

The unambiguous renaming of the bound variables should be supported.
The definitive names should be determined as quickly as possible, not first
after the parsing process is complete. The bond analysis is thus entangled with
from parsing. This leads to the fact that the component is also at arbitrary moments
fresh variables can generate what the replacement rules from the previous one
Section makes it much more efficient: replacement rules can then be used to define local variable
Introduce bindings without causing name clashes. For example
there was the following replacement rule in Oz in version 1.1.1:

```
thread E end )        local X in
                         if X = 'go' then E fi
                         X = 'go'
                      end
```

X must be a fresh variable so that there are no con icts with the free variables
blanks from the expression E can occur.

The bond analysis must be carried out in a skewed manner with the parsing process.
can, but should not be strictly bound by it. So it should be possible
their functionality also in handwritten (as opposed to generated) radio
functions, for example if part of the abstract syntax tree has
simultaneous unique renaming of all bound variables can be copied
should.

The component that performs the bond analysis should be used to save labor
extended by the user to a fully functional symbol table management
can be tert.

The support of several different types of variables should be provided,
for which different visibility regulations can apply. For languages this is the
Case in which other languages are embedded, such as database queries
in extensions of C or grammar definitions in Oz as directed by the to
developing tool. Such a possibility is also applicable
if names of record fields or labels are globally unique as in C or Java
should be renamed.

Page 65

The tool aims to accurately represent the variable terminals and their
Values remain independent. This has the advantage that any additional information
can be saved by the user in the nodes, e.g. source code position
nen, the original names for diagnostic outputs or print names
generated, but speaking and informative names.

4.2.2 Approaches

This section aims to investigate which approaches found in the literature the
Carry out assignment of referencing and binding occurrences of identifiers.

The language
VisiCola Klu91]. An attempt was made to include all components as possible
identify from which such a speci cation can be built. The result is
a pure speci cation language in which everything has to be described in much more detail,
than is necessary for the task presented here. The author gives a few examples:
The specification of the Pascal visibility rules require 5000 lines, which are used by
Modula-2 (with the exception of the enumeration constants) 1280 lines. So this approach is
unsuitable for the problem at hand.

So-called hygienic macros HDB92] can be de ned in the Scheme language.

In principle, these correspond to the replacement rules coupled with binding analysis
se to avoid name clashes. For the unambiguous identification of the variables, a
Multi-pass timestamp algorithm applied. This operates on the completely con
structured the abstract syntax of a Scheme form. The situation there is simpler than in
in the present case, since a fixed source language can be assumed that
the variable bindings that can be determined very easily (in particular
the two different Common-Lisp namespaces for exist in Scheme
Functions and variables not).

In CMA94] an approach is presented which classifies the occurrence of variables.
coupled with the grammar specification. There are the variables
corresponding terminals are provided with an annotation that they can use as a binder, variable
(Reference) or label; So there are only two predefined visibility rules
lungs usable {the one for normal variables, which follows the block structure, and the one for
Label that is restricted to a single global scope. Block boundaries for visible
Areas of ability cannot be specified explicitly: Each production represents a block
Fresh variables can be added by specifying a variable terminal with the annotation
fresh.

4.2.3 Design decisions

The approach implemented here is a modification of the approach from CMA94]. The solution
roughly sketched looks like this:

The variables appearing are via the terminal annotations binder and reference
ten classified. In addition, the name of a variable type can be specified,
to which the terminal belongs (for example program variable, grammar symbol or
Label). As a result, the two types of variables that the approach from CMA94] and
First used, emulatable, but expandable by any number of others.

The terminal annotation fresh creates a fresh variable.

The grammar is expanded to include a further bracketing construct, namely
scope ... end. A type of variable can be specified. The construct spe-
zi decorates the block boundaries of a visibility area of the variable type. Thereby,
since these are brackets, these blocks are always nested.
This system is less restrictive than that from CMA94].

There is a predefined class called 'BindingAnalysis' that provides the necessary
stores and processes information. It is in-

stamped. It allows the operational implementation of the declarative grammar notations and offers the operations openScope and closeScope for the specification the block boundaries. mkBinder, mkLabel and mkFresh correspond to the Terminal Annostations; With enterSubstitution, implicit declarations can be made will.

By expanding the "BindingAnalysis" class, a full-fledged symbol table can be management can be implemented.

The user must load methods in a class derived from 'BindingAnalysis' which take over the abstraction from the terminal representation. For this count the following operations:

anonymize converts a terminal into one with a (not determinated) variable is used.

nameToken replaces the placeholder in an anonymized token with a concrete name (as an atom).

generate generates a new name (as an atom). This function can consist of a regular expressions specified by the user are generated in the sense ne, since it contains the elements of the language that is de ned by the expression, enumerates.

tokenToAtom returns an atom that contains the (user-given) name of the variable identified. This is used for the internal administration of the substitutions. Are not all characters of the lexeme relevant (e.g. in UCSD Pascal, where only the first 8 characters apply) or there are several representations for the same object (as in Oz, atoms with and without quotes), so can these Dependencies on this function can be made transparent.

makeTokenFromAtom creates an appropriately named variable from an atom riablentoken. This operation is only performed in combination with 'fresh' used.

The advantages of this method over CMA94] are obvious. For one thing is the Binding analysis now orthogonal to the replacement rules. Furthermore, from the Representation of the tokens, so print names in particular can be instead of simply numbering the identifiers sequentially. The sub stitution can be made explicit as a first-class data structure and thus further processed (for example for the output of a symbol file). By separating between declarative annotations of the grammar and the operational part (in the class, binary

dingAnalysis`) the latter can also be used independently of the rest of the system.

The concrete syntax of the annotations and the description of the class, BindingAnalysis` are given in Section 5.5.

4.3 Automatic construction of abstract syntaxes

Often the result of the parsing phase should be an internal, structured representation of the entered program. This chapter examines ways to
to support this transformation with a tool. But first I have to look into some
Before considering it, it is necessary to identify the properties of a suitable internal
Representation are expected.

While the linear token stream is described by a concrete syntax, with
the term abstract syntax denotes a representation that is subject to restrictions or
abstracted from overspecification, which is characterized by the requirement of parseability and / or
Readability. This term can be de ned more precisely: A syntax a applies
as more abstract than a syntax b if and only if there is a total function from a to b,
however, the inverse relation is not a function Bev93]. In practice, this form is the
Abstraction means removing essential details from the representation.

It is obvious that any number of abstract syntaxes can be used to form a concrete syntax
can give. However, these can be used differently for processing in the following
appropriate to the phases, analogous to the effects described in section 3.2
different grammar formulations for concrete syntax. For the abstract
Syntax even the following requirements are made:

> Between the tree nodes of the abstract syntax and the semantic concepts
> the language should have a one-to-one relationship. This can be in the concrete
> Syntax usually cannot be established.
>
> In the structure of a syntax, different numbers of context conditions can be used
> Language (also called static semantics). An abstract syntax

> should make use of this as intensively as possible because of structural restrictions
> are often easier to understand than those that (in the semantic analysis) through
> arbitrary predicates can be expressed.

The goodness of an abstract syntax can be demonstrated by an analysis of the compiler phases that refer to you operate, be evaluated. For this purpose, it must be identified in each case which information

tions required and which ones are produced. With good abstract syntax, these are required context information of each node in a simple recursive descent easily accessible.

For these reasons, an abstract syntax is better than a concrete one for them Development and semantic definition of programming languages suitable. It often becomes that It is advisable to start with an abstract syntax, for which only then a concrete syntax is de ned when all the concepts of the language are well understood. In practice However, the opposite route often has to be followed: Many language reports only give the concrete syntax. For a compiler builder who already has implementation experience with the paradigms of the language for which a translator is to be developed is designing an abstract syntax is usually not difficult. So the wish comes quickly after tool support.

Section 4.3.1 identifies the aspects that are necessary for tool support are relevant. The requirements for such a system are also formulated. A An overview of existing approaches is given in Section 4.3.2. Finally, in ab- Section 4.3.3 justified why there is no support in the tool developed here for the automatic construction of abstract syntaxes.

4.3.1 Goals

Three aspects can be derived from the de nition of the term abstract syntax Automation must also be included: 1) the speci cation of the concrete Syntax k through a grammar, 2) the derivation of an abstract syntax a, the example wisely given by a type de nition, and 3) the mapping: k ?! a (a total Function).

The development of a tool that carries out this transformation on its own, equates to the definition of a limited class of abstract syntaxes that can be generated. In other words: Let K be the set of all concrete syntaxes. Then the functional wise of the tool through the set of abstract syntaxes A it can generate, as well as by a function: K ?! A de ned. (With the above designations, gen a 2 A, k 2 K and 2.)

With this formulation of the task, it is clear in which aspects the solutions can distinguish:

An important aspect is the shape of A, that is, with which data structures

the abstract syntax is built up in the target language. There are roughly three possible be differentiated:

Tuple. Tuples are tuples like in Oz or product types like understood in Gofer, i.e. the Cartesian product of n types, provided with a named tag (called a label or constructor). This is neither explicitly stored nor managed by typing the target language. Because of their similarity, tuples represent both BNF productions and trees with named nodes are the simplest and most common form. syntax. Important aspects are for which semantic concepts Constructors are de ned and which partial values are saved in each case have to.

Records. Tuples are referred to as records here, the elements of which are defined by field names men are addressed. There is also the need to provide names for this Fields to find. Assuming there is a relationship between certain th productions and the constructors; then there are two ways of doing this Naming: Either the user can change the field names by annotating the Assign productions or the tool generates the names from the Grammar symbols used in production. The uniqueness of the Names guaranteed by an appended index that occurs in the Production marks. The advantage of a record representation is that it is in less error-prone to use and more resistant to modifications the abstract syntax is.

Objects. In the case of an object-oriented abstract syntax, further basic Various structuring options are added: Between the individual Node types can now exist inheritance relationships (a node type is a specialization of another) and property classes (also: Mixin classes) can be de ned. As a result, because with the node types directly Operations (methods) are connected, the implementation management of the compiler phases considerably: Instead of a case distinction The dynamic is used in relation to the constructor of a tuple or record Binding. Another implication is that the reuse of egg property classes for common concepts or paradigms of programming languages is possible. For these reasons the creation is an abstract one Object syntax is a far less trivial process than using Tuples or records, if these possibilities are to be eliminated.

The other important aspect is how abstracting the functions are are that Tool used. Is fixed as constant during the development of the tool. that is, abstract syntaxes are exclusively derived from the concrete syntax. no context dependencies (which are not included in the context-free

Grammar can be coded) be included. This case is also called a
ne abstract syntax of the first order denotes Bev93]. Otherwise the user has to
You can provide additional information that controls the generated function
allow.

Apart from these basic questions, the following aspects are for a
Tool support important:

Local grammar changes should only be local changes to the abstract
Effect syntax. This requirement is important for those in development
Languages or for the development of a compiler through gradual refinement.

The tool should generate the de nition of the abstract syntax a as well as
also apply the underlying to the input. The result is called the
abstract syntax tree called.

The automatic structure of an abstract syntax is to be based on parts of the grammar
can be adjusted. In the case of interactive systems, for example, this prerequisite
can be limited to sub-trees: Each processing unit of a text
based user interface (such as a line or a command) can be automatic
be converted into an internal representation, but this must then be processed immediately.
be switched. This requirement is more important when side effects are exerted deep in the tree
must be, for example lexical tie-ins or modifications of the symbol table.

The advantages of tool support are not just a more compact one
and more readable specification of the transformation into an internal representation or ar-
labor savings. A much more important aspect is because it is a formally described abstract syntax
can be used for the automation of other compiler phases: For many tools
ge, the pattern-based tree transformations, unambiguous renaming of the bound
Apply identifiers or automatic attribution, this is a basic precondition
suspension. The replacement rules and binding analysis presented above are more likely to be used here
the exception.

4.3.2 Approaches

The generation of an abstract syntax of the first order in tuple or record form from a
concrete syntax is a very simple process. There are simply the following implementations
of the components of an (E) BNF:

Constant terminals are eliminated. Constant means here that the terminal
carries no interesting value, as is usually the case with literals.

Variable terminals are assigned a unique type name directly to them
mapped with a single constructor. Variable here means that the term
minal carries an intrinsic value, such as the numerical value of an integer token.

Non-terminals , like variable terminals, are mapped to type names. This
can, however, contain multiple constructors (see below).

Sequences are first broken down into their components, for each the abstract
Representation is determined. At this moment a case distinction must be made
be made: If a single element remains, it is a so-called
named chain rule and its constructors are converted to the currently created type
accepted. Otherwise the results are processed with a new constructor (for
a product type with corresponding elements). The unambiguous naming
This constructor can be used either by a user specification or by the
Concatenation of the non-terminal to which the considered production belongs with the
Production number.

Alternatives consist of several sequences, their abstract representations each
be determined. The constructors identified in the process are used in the currently
put type taken over.

Parentheses are outsourced to new rules. The name of the new non-terminal
can be done either by user specification or by generating a customized
dear unique grammar symbol.

Options are canceled because a default value is used when the op-
tion does not appear in the input. This can either be specified by the user
or can be specified by the generator.

Repetitions are translated into a primitive list type of the target language.

The resulting type definition can also be described by a context-free grammar.
be practiced; it only needs elements of the languages it defines
stored as a tree with named nodes {but neither unique nor the
Requirements to conform to any parsing technique.

Examples of tools that can be used in this way (or even more simply {without EBNF or Elimination of constant terminals and / or chain rules) are TXL CCH95]
or the approach from Wad90]. With the Maptool KW95] tool from the Eli system
this also corresponds to the given relationship between the concrete and the
straight syntax. However, the user can control this process by means of special directives
and thus make the abstract syntax actually more abstract than the concrete: it can
Identi ed nodes of different non-terminals with each other (whereby, for example, chain
rules can be resolved), grammar symbols eliminated or for the same non-
terminal {depending on its context {different node types are used.

At the points where the concrete and the abstract syntax (with the above rules)
only one of the two needs to be specified, with the other
is then automatically completed.

The tool PCCTS Par95] offers an alternative procedure. Be here
only limited tree structures are supported (from which, however, complicated
can be built), which correspond to the conses from Lisp: Every node owns
Slots for an rst-child and a next-sibling pointer. Productions can either
have an automatic node construction carried out or in a special semantic
Specify the exact shape of the subtree to be instantiated. Will the
If automatic construction is carried out, further operators are available in the EBNF
Disposal: The inclusion of a subtree in the sibling list can be prevented (Eli-
mination of constant terminals) or a subtree can become the new root of the current
the constructed results of the production are explained (elimination of chain rules),
as a result, it is not included in the sibling list, but instead via this
go on.

If an object-oriented representation is to be found, the above can be used
Standard scheme can be modified as follows:

Variable terminals and non-terminals are mapped to classes.

Sequences are broken down into their elements, which are stored in attributes of the class
will. Classes are not created, rather than constructors.

Alternatives don't just collect different constructors for the same type,
but the classes generated from the sequences. These are then passed over
Exercise on alternative options for the currently created class explained. (Is the
The grammar is clear, so only cycle-free class hierarchies are generated.)

It should be noted that in this mechanism there is no inheritance of property classes
is used for the first time. Analogous to how the production schemes in section 5.3.3
Generalizations of the EBNF operators were, would still be conceivable, user-de ned
Allow EBNF operators by instantiating generic classes.

For example, according to this method, the grammar

Type: NamedType j ArrayType j RecordType:
NamedType: Identi er:
ArrayType: ARRAY Integer OF Type:
RecordType: RECORD Fields END:

translated into a class hierarchy as shown in Figure 4.1. The

rounded rectangles represent classes, the names of which appear above the horizontal right line is indicated and its attributes with type appear below. Arrows

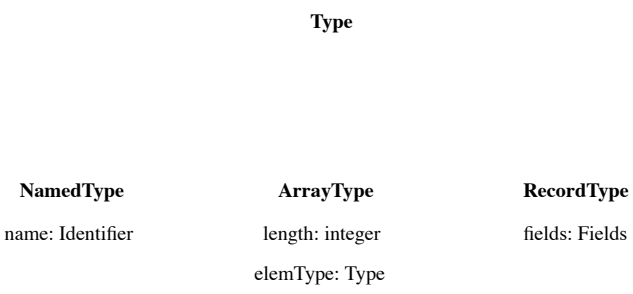


Figure 4.1: Example of an object-oriented abstract syntax

illustrate the „specialized \ relationship. In the illustration are for the (however generated) attribute names have been chosen.

A corresponding method is used, for example, by Mango Age94]. There the This simplifies the situation, however, since structured grammars (cf. 3.1.3) applied: Every structured production corresponds exactly to the application of one of the above rules.

Another tool that works in this direction is Ast Gro93] from the Cocktail-Toolbox. This allows abstract syntaxes to be specified for which the tool then automatically type definitions as well as tree scrolling and input / output functions generated (the structure of the abstract syntax tree, however, the user must in the se- perform mantic actions by hand). It is not a question of ob- jector-oriented abstract syntaxes, but around records with inheritance relationships can be de ned between nodes and equivalents to mixin classes.

4.3.3 Design decisions

In summary it can be said that it is in the area of abstract syntaxes in tuple form Possibilities do exist, differences between the concrete and the abstract Specify syntax {even if only a few tools support this. The goal, a Generating really more abstract representation could consequently be fulfilled, but will often bought at the cost of another (non-trivial) notation. Considering how easy to read and compactly write down a record construction in Oz (for which there is also no type definition is necessary), it becomes clear that tool support is not necessary here: Used instead of a dedicated tool, you simply have semantic actions in Oz

a very powerful notation {without having to de ne another one of your own.

There are far fewer approaches for building object-oriented abstract syntaxes in the literature. The greatest potential in comparison to the tuples is here in the re- the use of property classes, as in the OCC system Kna96] for the impe- rative and object-oriented paradigms are practiced on a large scale. Which own class, however, is required for other paradigms has not been investigated.

For these reasons, tool support was used in the present work apart in the area of abstract syntax. This limits the possibilities of others planned part tools are not included, as in contrast to most other tools the substitution rules de ned here and the linkage analysis do not depend on the existence of a abstract syntax.

4.4 Automatic unparsing

Unparsing describes the process of a character string from an abstract syntax tree in concrete syntax to generate the relevant syntax tree when parsing again supplies. This is relevant when implementing a translator of the programs in one language into a program in the same language. Examples for this are Optimizations or simplifications at the source code level. Is the language about reduction de ned into a core language (for which the substitution rules presented above are used can be), the output of this phase is an aid for the programmer, if he wants to better understand the semantics of language.

Typically, the solution to this problem consists only of a recursive passage run the abstract syntax tree, in which a character string is generated for each node becomes. This is made from the information of the node itself as well as from the information for the successor node generated strings. To write such a program by hand is a cumbersome and error-prone task, as it must be guaranteed that the The output string is a legal symbol sequence of the concrete syntax (which is also still represents a program equivalent to the syntax tree). Because of this, one is Tool support is desirable.

Requirements for this process are de ned in Section 4.4.1. Thereupon be Existing approaches are presented in Section 4.4.2. Section 4.4.3 explains why a Automation was not implemented in the present tool.

4.4.1 Goals

As the output produced by this phase is in the context in which in this work a
Tool should be developed, mainly for the user of the generated compiler
is intended, the following requirements are made:

The output should be legible. In particular, this means that they are the ones in the language
normal formatting conventions are respected. These are with fully compositional
Languages more complicated than other languages: Because they are very complex nested
Expressions can occur, it is not enough, pro „Instruction \ to provide one line.
There must also be permitted positions for line breaks within expressions as well
associated indentations can be de ned.

Is the language de ned into a core language through reduction and is this through
Substitution rules implemented, so only a small part of the productions come
the specific syntax for the output in question. Production should be accordingly
the unparsing function can be restricted to the node types of the core language
can.

The unparsing specifications should be as independent of the abstract syntax as
be possible so that changes to this only affect the additional
sing declarations.

4.4.2 Approaches

Since the tree traversal plays the controlling role in unparsing, it is obvious that the
abstract syntax must be known to the tool. The in the illustration of the concrete
the function underlying the abstract syntax (mentioned above) must to a certain extent
en are reversed (which changes the permitted abstract syntaxes to those of the first order
restricts).

There are tools that do not just automatically turn the concrete syntax into an abstract one
Generate syntax, but also an unparsing function. These are briefly described below
to be introduced.

The tool Kimwitu vEB93] offers a special notation for specifying
of unparsing. Rules are selected through pattern matching (the user
must therefore undertake the recursive descent yourself), whereupon the right side of the
denotes the string of characters by a series of unparsing items. These will be one at a time
unparst (for example an attribute or a constant string) and concatenated.
Actions written in the C language are also possible; in this can turn

further unparsing items can be embedded.

The speci cation of unparsing runs similarly in SSL, the speci cation language of the Cornell Synthesizer Generator RT88]. However, positions can still be specified here those on which line breaks are permitted but do not necessarily have to be carried out. The The actual formatting is then found out at runtime and can vary from sizes such as for example the window width depends.

The ParsesraP Bea90] tool is based on another idea: There the input input grammar used at the same time as output grammar. Can be used for any production in addition to the semantic action for reading in, there is also an action for outputting be inflicted. This tool does not need to know the abstract syntax {the Unparsing rules are specified in the semantic output actions by naming their non-terminals called. Since not between several productions of a non-terminal can be differentiated, the first one is selected by default. Another disadvantage is because formatting conventions have to be incorporated directly into the grammar {every place where spaces are allowed has to be formulated in the grammar.

In TXL CCH95] the input grammar also becomes the output grammar at the same time. However, since the structure of the abstract syntax is known here and since various product tions for the same non-terminal are to be distinguished, it is sufficient here to use the grammar to add some formatting hints. The declarations SP] for are available for this a space, NL] for a line break, IN] for an increase in the indentation depth and EX] for its reduction. The tool also offers a ready-made Set of formatting rules that, according to the authors, apply to most Pascal or C-like languages provide meaningful results. However, if necessary, you can also switched.

4.4.3 Design decisions

Since no automatic abstract syntax was developed in the tool developed here. de, the prerequisite for supporting unparsing is not given. Indeed it is very convenient to build the output through the virtual strings of Oz { in most other languages this is more cumbersome.

To encode formatting information in the string, the following is exible, but simple system conceivable. One would have controls for the output formatting as tuples of the form format (atom) in the virtual string, which in a post-processing step have been replaced by indentations and line breaks. The string run through and internally kept a list that saves the last indentation depths. The following control atoms were desirable:

break creates a line break and jumps according to the one at the top of the stack given indentation depth.

glue indicates a place where a line break is allowed, but not compulsory must take place. Depending on how much space is left on the current line, this control information is ignored or treated as a break.

indent increases the indentation depth given at the top of the stack by a constant n (for example 3).

exdent lowers the indentation depth given at the top of the stack by one contact stante n.

push puts the current column on the stack.

pop removes the top element of the stack.

These formatting directives are an extension of those offered by TXL. Through optional nal line breaks like in SSL and the indentation stack, they become fully compositional However, languages were more fair. A function that supports this formatting should be included in the standard scope of later versions of the tool.

4.5 Modularization of specifications

The input for a tool that is to generate a front end can optionally be get very big. It therefore makes sense to want to design specifications in a modular manner. Section 4.5.1 examines which parts of the specification should be modularized and how. Existing approaches are then presented and assessed in Section 4.5.2.

4.5.1 Goals

This section does not discuss the general advantages of modularizing special cations examined; only the goals are explained, which result from the shape of the tool developed in this work.

The class-based approach of the tool makes it easy to do the lexical analysis to be separated from the remaining phases, as also in Section 1.4 as a requirement was formulated. However, since the replacement rules and the linkage analysis are entangled with the parsing performed, it seems better not to have these partial speci cations to separate. It is therefore to be expected that the majority of a front-end speci cation

is claimed by the syntactic rules with their annotations. Of special
Interests are therefore modularizations of context-free grammars.

A question that always arises in modularization is that of the separate translation
availability. Depending on the parsing algorithm, this is in principle conceivable, provided the degree of coupling
the sub-parser is very small and the same token stream is used. For example,
wise a (closed) grammar for expressions to be written and by one
other grammar can be used, provided that these are only complete expressions and none
Partial expressions required. Parsing techniques for which a corresponding incremental parser
generation is possible, are those in which each nonterminal has its own function
The translation is {for example in PCCTS PDC91] for LL or the approach in BP95] for LR. Also
this is not a problem with Combinator Parsing. The aspect of modularization
has hardly been found in the literature at this level. Since then only
Possibilities exist that depend heavily on the parsing technique, separate transfer is made
settability not required.

4.5.2 Approaches

PCCTS PDC91] makes it possible to distribute the productions over several files
simply concatenated in the translation. A real logical independence
there is no between the partial speci cations; there are no controllable cutting
and the coupling can be arbitrarily high.

This is an example of how the clarity can suffer from modularization
Tool Eli Com96a], Com96b]. There must be many directives in separate files

be written, even if they are logically related. By such a
The solution is foreseeable.

A more interesting approach is that of CMA94], in which grammars are incremental
can be de ned by removing an existing one by adding nonterminal
len, adding alternatives to existing nonterminal and replacing
Productions is modi ed. The authors would like to use this to
Antennas, sublanguages and extended languages support an application of this system
for the purpose of modularizing a single grammar is also conceivable.

4.6 Summary

In this chapter several possibilities have been examined to expand the tool.

tern, which was developed in Chapters 2 and 3. The automatic implementation of simplifications, bond analysis, construction of abstracts Syntax and unparsing. Furthermore, the possibility of modularization was considered.

In the end, only the first two extensions were taken over into the tool. men. Replacement rules and a specification for bond analysis have been developed, which differ from the approaches from the literature in order to meet the intended area of the tool {the translation of multiparadigmatic and fully compositional Languages {to meet.

Chapter 5

De nition of the tool

This chapter provides a detailed description of what has happened over the previous one Chapter designed front-end generator.

Section 5.1 gives an overview of how the sub-tools can be used to create a complete be joined together. An example of a complete speci cation is used to pressure from it conveys what the result looks like. After that, each component is detailed explains: Sections 5.2 to 5.5 describe the scanner generator, the parser generator, the execution of the replacements and the binding analysis. The meaning the language constructs are explained on the basis of their concrete syntax and which describe the prede ned classes which operationalize the speci cation take over. Wherever it seemed necessary for understanding, it is also used Algorithms received. Often examples from the Oz syntax are given.

Finally, in section 5.6 some notes on implementation are given and it explains how the tool was implemented using itself. It is interesting to see how the first tepid system was created.

5.1 Overview of the overall system

This section describes how the part tools interact and how their inputs into a single speci cation language, which is also embedded in Oz.

Section 5.1.1 gives an overview of which phases a front end supports and how the corresponding sub-tools build on each other. Thereupon Section 5.1.2 gives an example of a complete speci cation that allows this interaction nen clarified. Section 5.1.3 shows how the Grammar framework that accommodates front-end speci cations, incorporated into the Oz language. is in bed.

Scanner definition

Generator frame Substitution rules

Figure 5.1: Dependencies between the sub-tools

5.1.1 Dependencies between the partial tools

Figure 5.1 shows how the individual components of the tool depend on one another.
An arrow shows a used relation.

The generator frame explained in section 5.1.3 is embedded in Oz and is
ne Oz class end translated. Scanners and / or parsers can be de ned in it;
however, these de nitions are independent of one another. The substitution rules and varia-
However, naming for the binding analysis is based on a parser. The one in Oz
The predefined class, BindingAnalysis`, is written by the binding annotations
referenced, but can also be used without it.

5.1.2 Complete example

The following example gives a complete front-end speci cation for a small test
language. In addition to the lexical and syntactic analysis, the example contains
added to the visibility rules of the language, so that the bound identifiers
automatically be renamed.

The language is lexically copied. An identifier is bound via decl, which can be found in
its visibility area can be referenced via use. Identifiers are always
visible in the whole block between the keywords block and end in which you
be de ned. For labels that can be used after the keyword label,
there is only one visibility area that encompasses the entire program.

Identifier tokens are internally represented by atoms that represent the name of the identifier
ner correspond. Therefore, the operations in Identi erBindings for the binding
analysis can be de ned, very simple.

```
\ switch + synverbose
declare
class IdentifierBindings from BindingAnalysis
   attr count: 0
   meth anonymize (X? Y)              Y = _    end
```

```
    meth nameToken (XA)                 X = A     end
    meth tokenToAtom (X? A)             A = X     end
    meth makeTokenFromAtom (A? X)       X = A     end
    meth generate (X? A)
       count <- @count + 1
       A = {String.toAtom & X | {Int.toString @count}}
    end
end
grammar Example from UrObject LexBaseClass SynBaseClass
    attr default labels
    meth init
       LexBaseClass, init
       SynBaseClass, init
       default <- {New IdentifierBindings init}
       labels <- {New IdentifierBindings init}
    end

    lex <block> << lexYield ( Â´ block Â´ ) >> end
    lex <end>     << lexYield ( ´ end ´ ) >>end
    lex <decl> << lexYield ( ´ decl ´ ) >> end
    lex <use>     << lexYield ( ´ use ´ ) >>end
    lex <label> << lexYield ( Â´ label Â´ ) >> end

    lex <[A-Za-z] [A-Za-z0-9 _-] *>
       << lexYield (ident {String.toAtom @lexeme}) >>
    end

    lex <[\ t \ n] +> skip end
    lex <.>
       << synError ("illegal character in input") >>
    end

    token ´ block ´ ´ end ´ ´ decl ´ ´ use ´ ´ label ´ ident

    syn program (? B)
       scope (labels) B = block ($) end
    end
    syn block ($)
       ´ block ´
          scope Ss = (Statement ($)) + end
       ' End '
       => block (Ss)
    end
    syn statement ($)
          Block ($)
```

```
            [] ´ decl ´ ident (I): binder              => decl (I)
            [] ´ use ´ ident (I): reference            => use (I)
            [] ´ label ´ ident (I): binder (labels) => label (I)
        end

        meth start (FileName? Program? Status)
            Buffer = {New LexBuffer fromFile (FileName)}
        in
            LexBaseClass, lexSwitchToBuffer (Buffer)
            SynBaseClass, synParse (program (? Program)? Status)
            {Buffer close}
        end
    end
```

In the grammar, attributes are first de ned that contain the information for the binding analysis of identifiers (default) and labels (labels), as well as the measurement method init which initializes it. This is followed by the de nitions of the lexical structure. First the token types are declared for the parsing, then the syntax rules are given. The information necessary for the bond analysis is stored directly in the Syntax rules specified. The last method starts analyzing a file and returns back the abstract syntax tree.

5.1.3 Concrete syntax of grammar definitions

The notation used in this section for context-free grammars is described in more detail in Appendix A.

The front-end generator is started by specifying an Oz file, which is also gram contains matic de nitions. The output of the generator consists of a newly created one File that has the same content as the input file with the exception that all gram matic de nitions have been replaced by class end nitions.

A grammar definition is allowed wherever a named class end - nition may stand:

h expression i + = h grammar i

A grammar de nition is similar to a class end nition, but is defined by the key word grammar introduced instead of class and must always be named by a variable will. Their name is used during the translation for the naming of generated files used.

h grammar i :: = grammar h variable ifh grammar descriptor i g end

All class descriptors (from, attr, feat and meth) are used as grammar descriptors authorized. Later on, further alternatives will be added to this rule in which the lexical structure and context-free grammar are specified.

h grammar descriptor i :: = h class descriptor i

5.2 The scanner generator

The scanner generator is a tool part of the front-end generator. When a grief lexical de nitions, it is called automatically to analyze them.
and generate appropriate descriptors for the generated class.

The description of the scanner generator is divided into three parts. The first (in Section 5.2.1) describes on the basis of the specific syntax used for this, such as the dem The machine on which the scanner is based is specified. The runtime functionality of the scan ners and its interfaces from the user's point of view are defined by the prede ned Classes are delivered, which are explained in Section 5.2.2. The third part concerns the control generation of the scanner generator itself, which is carried out using the compi learning directives is implemented.

5.2.1 Scanner specification

Scanners are specified in additional grammar descriptors. These are xical de nitions and fashion clauses expanded.

h grammar descriptor i + = h lexical abbreviation i
jh lexical rule i
jh mode clause i

Regular expressions can be named. You can then by specifying their Abbreviated names can be used to construct new regular expressions.

h lexical abbreviation i :: = lex h label i = h regex i end

Examples:

The following abbreviations are useful to specify Oz tokens:

```
lex lower = <az]> end
lex upper = <AZ]> end
lex digit = <0-9]> end
lex alphaNum = <{lower} | {upper} | {digit} | _> end
lex atom = <{lower} {alphaNum} *> end
```

A lexical rule is similar to a method ending. However, she will
Introduced by the keyword lex instead of meth and your head consists of one
regular expression. The method body is executed if the regular
Expression is selected.

h lexical rule i :: = lex h regex i h expression i in] h expression i end

Examples:

The following rule recognizes (with the above definitions) an Oz atom and returns a token with the token type 'ATOM'. The value of the token is the lexeme as Atom:

```
lex <{atom}>
    << lexYield ('ATOM' {String.toAtom @lexeme}) >>
end
```

It is possible to match all one-character tokens by a single rule.
There are two methods of doing this. The first simply converts the matched character into an atom, which is then returned as a token type:

```
lex <{} () \] | #: =.^@$! ~ _,]>
    << lexYield ({String.toAtom @lexeme}) >>
end
```

The second possibility makes use of the fact that the whole numbers 1 {255 in
Parsers are predefined. If the scanner is sent with a generated parser
used together, this speci cation has the same semantics as the previous
rige, but is more e cient:

```
lex <{} () \] | #: =.^@$! ~ _,]>
    << lexYield (@ lexeme.1) >>
end
```

A fashion clause introduces a new lexical mode. The included lexical
General rules apply in the mode as well as in all modes derived from it. It can
De nitions of other modes are inherited by listing them in a from clause
will. Nested modes implicitly inherit the lexical rules of everyone around them-
the modes.

h mode clause i :: = mode h variable ifh mode descriptor i g end
h mode descriptor i :: = h inherited modes i

jh mode clause. i
jh lexical rule 1
h inherited modes i :: = from fh variable i g

Page 85

Example:

In this example, nested comments are used over lexical modes recognized. To do this, an attribute is used that shows the current nesting deep stores. The rules of the lexical mode 'COMMENT try out E ciency reasons, always matching as much text as possible:

```
attr CommentDepth
lex <"/ *">
    CommentDepth <- 1
    << lexSetMode (COMMENT) >>
end
mode COMMENT from INITIAL
    lex <"/ *">
        CommentDepth <- @CommentDepth + 1
    end
    lex <"* /">
        CommentDepth <- @CommentDepth - 1
        case @CommentDepth == 0 then
            << lexSetMode (INITIAL) >>
        else skip
        end
    end
    lex <^ * /] +> skip end
    lex <"*" + ^ * /]> skip end
    lex <"/" + ^ * /]> skip end
    lex <<EOF>>
        << reportError ("underminated comment") >>
    end
end
```

The syntax of the regular expressions is not explained in detail here. It corresponds to die ex Pax95] is used, with the following exceptions:

The modes in which an expression is valid are not given in angle brackets in the regul aren expression presented, but speci ed via the mode clauses.

The syntax of the names of lexical abbreviations, noted in curly brackets mern, is extended to the syntax of Oz atoms.

Regular expressions are used to avoid confusion with the comparison operators of Oz to avoid {recognized only if they were preceded by the keyword lex.

Page 86

5.2.2 Preceding classes

The scanner generator ex is used to generate the machine. These facts
However, the user does not need to be familiar with this because the basic functionality of the Scanner is given by some prede ned classes that provide the interface to the ex-Encapsulate scanner.

The class 'LexBaseClass'

The Mixin class, LexBaseClass` implements the methods for interacting with the Scanner and the table scanner itself. Should a
Scanners are brought to execution, so must be inherited from this class.

'LexBaseClass' requires some features and methods on the part of the inheriting class. The same notation is used for the modes of method parameters as for the Description of the Oz Standard Modules HMSW96]).

lexer Feature]

This feature contains the generated tables for the scanner.

lexExecuteAction (+ I) Method]

This method is called when a match has been selected. The parameter ter is the number of the regular expression; the method must be the appropriate perform semantic action. The method is created by the generator; also the The regular expressions are numbered automatically.

, LexBaseClass` de nes the following functionality:

lexeme Attribute]

Contains the last matched lexeme as a string.

length Attribute]

Contains the length of the last matched lexeme as an integer.

lexYield (+ AX (Unit) Method]
Appends a token of token class A with the value X to the token stream.

lexSwitchToBu er (+ Bu er) Method]
Redirects the input stream to the transferred book. This must be an instance of the class explained below, LexBu er`.

lexCurrentBu er (? Bu er) Method]
Returns the currently active input bu er (an instance of the class "LexBu er").

lexSetMode (+ I) Method]
Switches to the machine that corresponds to lexical mode I.

Page 87

lexCurrentMode (? I) Method]
Returns the integer that stands for the active lexical mode.

lexInput (? C) Method]
Requests the next character in the input stream and removes it from the input. Here, handwritten mini scanners can be embedded in the scanner.

lexUnput (+ C) Method]
Appends the transferred character to the front of the input stream. This means that the following behavior of the scanner can be influenced.

lexAppendMatch () Method]
Causes the next match not to have its lexeme in "lexeme" and "length" saved, but the current one is added in front of it. (This method speaks the ex-function yymore.)

lexShortenMatch (+ I) Method]
Abbreviates the current lexeme at the end by I characters that are returned to the input stream to return.

lexRejectMatch () Method]
Causes the selected match to be rejected. Then the next best selected.

lexGetToken (? A? X) Method]
This method is intended for calling from outside. She delivers the token (with type A and value X), which is at the beginning of the token stream.

lexPreAction () Method]
Doesn `t do anything; the method can be overloaded by the user. It will be with everyone
Match is called before the associated semantic action is carried out. In the-
This action can, for example, update the source text coordinates or
Debug information is output.

close () Method]
Should always be called after the completion of a scan.

The class 'LexLineNo'

The Mixin class, LexLineNo`, provides a way to get information about the current
Update position in source code:

lexLine Attribute]
Contains the current line number as an integer 1.

Page 88

lexColumn Attribute]
Contains the current column number as an integer 0.

lexPreAction () Method]
Overloads the method of the same name, LexBaseClass` and updates the above
both attributes in every match. That means because the coordinates are always on
interpret the sign according to the current lexeme.

The class 'LexBu er'

The class "LexBu er" represents an input buffer for a scanner. Your instan-
zen can be passed to the method, lexSwitchToBu er` of the Mixin class, LexBaseClass`
will. It offers the following functionality:

init () Method]
Must always be called (is automatically called by 'fromFile' and 'fromVirtualString')
table done).

fromFile (+ V) Method]
Initializes the input book with the contents of the file named V. The book
is considered interactive by default.

fromVirtualString (+ V) Method]
Initializes the input bu er with the value of the virtual string V. The bu er applies
not as interactive.

setInteractive (+ B) Method]
Sets the input mode of the current book to interactive or not, depending on the value
the Boolean variable B. Interactive means that the characters are requested individually
and not en bloc, i.e. a minimum number of characters must be present. Relates
If the input file is on the keyboard, for example, then user input
can be edited immediately. Non-interactive books can be edited faster
will.

getInteractive (? B) Method]
Returns a flag as to whether the bu er is currently considered to be interactive or not.

setBOL (+ B) Method]
Sets the value of a flag that indicates whether the input pointer is currently at a line
lenbeginn (B eginning O f L ine) stands or not, in other words: whether the ^ operator is matched
may or may not be.

getBOL (? B) Method]
Returns the value of the flag that indicates whether the input pointer is at the beginning of a line
stands.

Page 89

close () Method]
Closes the file being read from. Mu always be called when a
If it is no longer needed.

5.2.3 Compiler directives

Table 5.1 gives an overview of the compiler directives that are supported and that the scanner
generation in uences. Syntactically they correspond to the Oz compiler directives, will
but processed by the front-end generator and not appearing in the generated oz file
more on. Most of the directives are modeled on options from ex version 2.5.2,
where a detailed description is given. In particular, the options that the table
control generation (lexC...), are not described in detail here and only the complete
Listed for the sake of quality.

Switch name	Default	meaning
lexignorecase	O	No distinction between upper and lower case

lexbest t	on	Use the best-fit instead of the first-fit rule
lexusereject	O	Support of "lexReject" in generating tables
lexCa	O	Corresponds to "ex -Ca"
lexCe	on	Corresponds to "ex -Ce"
lexCf	O	Corresponds to "ex -Cf"
lexCF	O	Corresponds to "ex -CF"
lexCm	on	Corresponds to "ex -Cm"
lexbackup	O	Output of the states that require backing-up
lexperreport	O	Output of a performance report of the scanner
lexstatistics	O	Output of statistics about the scanner
lexnowarn	O	Suppression of warnings

Table 5.1: Compiler directives for controlling the scanner generation

5.3 The parser generator

This section introduces the parser generator. Its concepts are based on
its specification language is explained and the predefined functionality is presented.
Just like the scanner generator, it is automatically called when a grammar
Syntax rules de ned.

Token types must be declared; the necessary speci cations are
Section 5.3.1 presented. The syntax and semantics of
Syntax rules de ned. This already fully functional parser generator is then used in
Section 5.3.3 is much more expressive due to the introduction of production schemes

made. Finally, in Section 5.3.4, the predefined classes and production
schemes presented.

The concrete syntax in this section is simplified for the sake of clarity
been. The syntax used in the tool has additional notational abbreviations
that do not change the power of the tool.

5.3.1 Token declarations

The atoms of length 1 are prede ned as terminals, the other token types must
however, they must first be declared before they are used as grammar symbols in syntax rules
can be. This is done via a further grammar descriptor.

h grammar descriptor i + = h token clause i
h token clause i :: = token h token declaration ifh token declaration i g

Token types are represented by atoms, just like in the scanner. Additionally you can
a maximum of one associativity and one precedence can be assigned to each token type.
This is done by using a term following a colon:

h token declaration i :: = h atom i : h term i]

The term must be a tuple with one of the labels leftAssoc, rightAssoc or nonAssoc
which indicate the associativity of the token. The only feature of the tuple is a
ne precedence allowed as an integer> 0. The absolute values of the precedents are
any, only the order of which counts: the greater the value, the stronger the
rator. This information is used to resolve ambiguities in the grammar
to be resolved as described in Section 3.2.1.

Example:

As an example, the information required for parsing Oz is given here by
Precedents and associativities listed:

```
token
    'declare': rightAssoc (1)
    '=': rightAssoc (2) 'FDCOMPARE': rightAssoc (2)
    '<-': rightAssoc (2)
    'orelse': rightAssoc (3)
    'andthen': rightAssoc (4)
    'COMPARE': leftAssoc (5)
    'FDIN': leftAssoc (6)
    '|': rightAssoc (7)
```



```
    '#': rightAssoc (8)
    'ADD': leftAssoc (9)
    'FDMUL': leftAssoc (10) 'OTHERMUL': leftAssoc (10)
    ',': rightAssoc (11)
    '~': leftAssoc (12)
    '.': leftAssoc (13) '^': leftAssoc (13)
    '@': leftAssoc (14)
```

5.3.2 Syntax rules

This section describes how to de ne syntax rules. After a
Some basic principles have been explained on the basis of the abstract syntax, is applied to the
Derivation of the attribute types of the parameters of non-terminals entered. Afterwards
remarks on the nontrivial implementation in an LR parser are given
and compared this with LL parsers.

Concrete syntax

Syntax rules are also specified as grammar descriptors.

h grammar descriptor i + = h syn clause i

A syntax rule {similar to an Oz method {consists of a head and a
Hull. The head indicates the non-terminal, as an atom or variable, as well as its parameters.
Atoms are used as starting symbols. Only one syntax rule may be de ned per nonterminal.
The trunk is given by an EBNF alternation.

h syn clause i :: = syn h syn head i h syn alt i end
h syn head i :: = h atom label i h syn formals i
jh variable label i h syn formals i
h syn formals i :: = (fh syn formal i g)

EBNF expressions can provide semantic values; otherwise they have at most
Side effects on the variables in their environment. The following is highlighted in each case
which constructs deliver values and which do not, as well as where values are expected and where
none are allowed to stand.

Parameters are identified by variables. A maximum of one parameter is allowed
a nesting marker ($) can be used instead of a variable. In this case
The body of the syntax rule delivers a value that is used in applications of the Nonterminal (see
below) is unified with the current parameter.

h syn formal i :: = h variable i
j $

An EBNF alternation indicates several EBNF sequences, separated by the choice operator]. Either every sequence has to supply a value or none; correspond- this then applies to the alternation.

h syn alt i :: = h syn seq i f] h syn seq i g

Local variables can be declared at the beginning of a sequence. These are only visible within the sequence. The sequence itself consists of a sequence of EBNF- Factors, optionally followed by a semantic action; this will be in the following Executions treated like an EBNF factor. For an empty sequence, true can also be written.

If an Oz term is specified as the semantic action, this returns its value, otherwise none. No EBNF factor in a sequence, with the exception of the last, may be a Deliver value; the sequence returns the value of its last factor or none.

h syn seq i :: = fh variable i g in] fh syn factor i gh syn action i]
j true h syn action i]
h syn action i :: = => h expression i in] h expression i
j => h expression i in] h expression i] h term i

An EBNF factor is either an application or an assignment. An application tion is noted by a terminal or non-terminal, followed by any current len parameters in brackets. Terminals must either have no parameters or be precise have a variable parameter; a variable is unified with the token value. At one In a nonterminal, the number of current parameters must match the number of formal Match the parameter in its de nition. Variables that are saved as Parameters are specified, are implicitly used in the surrounding sequence as local variables declared riable (variables are present as current parameters {in Oz terminology { Pattern position).

At most one current parameter may be a nesting marker. If so, so the application supplies the value of the parameter, otherwise none.

h syn factor i :: = h syn application i
jh syn assignment i
h syn application i :: = h atom label i h syn actuals i
h syn actuals i :: = (fh term i g)

At this point two predefined grammar symbols should be mentioned, one Experience special treatment:

prec By inserting an application of the prede ned grammar symbol prec,
A precedence and associativity are assigned to a production. One and only
current parameter must be a terminal symbol whose precedence and associative
vit at to be taken. For productions without the application of prec, the
Data taken from the last terminal used in production. This information
mations are used to resolve ambiguities in the grammar,
as described in section 3.2.1.

error An application of the predefined terminal error creates a restart point
de ned for error recovery. They are processed as with Bison DS95].
The terminal has no token value.

An assignment equates a variable with the value of an EBNF factor. This must provide a value. The variable must have been declared within the syntax rule. If it is noted without an escape symbol (!), It is declared implicitly. An assignment does not provide any value.

h syn assignment i :: = h escaped variable i = h syn factor i

Examples:

At this point the example from 5.1.2 should be read again. There
many of these constructs are used.

The following syntax rules implement the syntax rules for type de nitions that use in Section 4.3.2. The parameters provide the object-oriented
syntax tree:

```
syn Type ($)
        NamedType ($)
    ] ArrayType ($)
    ] RecordType ($)
end
syn NamedType ($)
    Identifier (I) => {New NamedType init (I)}
end
syn ArrayType ($)
    'ARRAY' Integer (I) 'OF' Type (T) => {New ArrayType init (IT)}
end
syn RecordType ($)
    'RECORD' Fields (Fs) 'END' => {New RecordType init (Fs)}
end
```

Derivation of the attribute types

In the case of parameters of grammar symbols, two types of attributes are differentiated.
den, namely synthesized and inherited attributes. In an application one can inherit
Te consider attributes as input parameters, synthesized as output parameters. in the
In contrast to Oz, where there is no distinction between input and output parameters thanks to the uni cation
the two types of attributes must be required due to restrictions on the
are conditioned by the parsing algorithm, are treated very differently. In order to
To take over this task from users, a system was developed that selects the types of attributes
the use of the parameter variables. This is covered in this section
presented.

However, a term must first be introduced.

De nition: Let a sequence with EBNF factors 0; :::; n be given. Let i be the index of the
first factor (application, assignment or semantic action) in which a local variable
riable V of the sequence is used. Then V holds in all factors with index j, j i,
as initialized , in all others as uninitialized . 2

Now we can introduce the rules on which the derivation of the attribute types is based
lie:

The optional parameter of a terminal is always a synthetic
tized attribute.

If in an application of a grammar symbol B a previously unitized local
If a variable or a nesting marker is specified as the i-th current parameter, the i-
te formal parameters of B a synthesized attribute. The uninitialized variable
must not appear in the other current parameters of the application.

If in an application a grammar symbol B becomes an already initialized local
If a variable or a complex term is specified as the i-th current parameter, then the
i-th formal parameter of B an inherited attribute. In the current parameter
there must be no uninitialized variable.

If in the syntax rule of a non-terminal A one of its formal parameter values is
riablen specified in an application of a grammar symbol B as a parameter, see above
the corresponding formal parameters of A and B are attributes of the same
Kind, i.e. either both synthesized or both inherited.

It should be noted that the use of a formal parameter variable in a semantic
table action nothing can be inferred about its attribute type: In Oz, between
There is no difference between access to and assignment to variables, since both are via uni cation
happens.

If contradicting attribute types are used for the same formal parameter of a non-terminals concluded that there is an error; if no species can be determined, a synthesized attribute assumed.

Example:

This example implements the declarations from Section 3.3, where an inherited attribute is required to pass the type through. Since T in the pro- used twice in the same sequence is that Argument of 'DeclaredVariables' an inherited attribute. Correspondingly follows this also for 'DeclaredVariable':

```
syn declaration
    Type (? T) DeclaredVariables (T) ';'
end
syn DeclaredVariables (T)
          DeclaredVariable (T)
      ] DeclaredVariables ',' DeclaredVariable (T)
end
syn DeclaredVariable (T)
    Variable (V) => << enterDeclaration (VT) >>
end
```

Realization of attributes and local variables in LR parsers

Both the de nition of local variables in EBNF syntax rules and inherited attributes are difficult to implement in LR parsers. This is also the reason why many Compiler builders prefer LL parsers. This section explains how these Implement features in the developed system with few restrictions.

Since LL parsers have a one-to-one relationship between non-terminals and functional can be created in the generated program, local variables can be added to the Entry into such a function must be declared. This is not the case with LR parsers, since it is not clear to which non-terminal the parsed input corresponds. The The only option is therefore to save local variables on the parsestack.

Thanks to the distinction between initialized and uninitialized local variables it is possible to find a fixed position in every production for every local variable to which it can be allocated. The time of their initialization thus corresponds to theirs Allocation on the parsestack. This is relative to the EBNF factors listed below Position always constant and known at translation time; so it can be efficient on the variables can be accessed. Since this variable was not declared locally in a function, but is in the parsestack, is transparent to the user because it is logical

Variables.

The problem also occurs with the inheritance of attributes, since non-terminals do not unambiguous functions can be assigned to which one can use the values as arguments which could pass inherited attributes. Access to inherited attributes is therefore Realized similarly to the one on local variables by taking their values directly from the parsestack to be fetched. However, there are two problems:

The relative position in the parsestack of the local passed as an inherited attribute Variables need not be unique. This is shown in the following example:

```
syn N ()
          a (V) M (V)
      ] a (V) b () M (V)
end
syn M (W) ... end
```

Let a be a terminal symbol; its application synthesizes the value of the variable V (strictly speaking, V stands for the token value from this point in time, which is in Parsestack is saved). This means that M becomes an inherited for the parameter W Attribute derived (in both sequences of the syntax rule of N). Its relative However, the position in the parsestack is one in the two sequences (as seen from M) other.

Instead of an inherited attribute, the current parameter is a complex one Term, its value has absolutely no position in the parsestack because the term has not yet been evaluated; for example:

```
syn R ()
     a (V) S ({FV})
end
syn S (W) ... end
```

The solution is to find all non-terminals that match one of these F be turned. Directly in front of the applications of each of these non-terminals, an additional licher slot created on the parsestack with the values of their inherited attributes. In order to complex terms are evaluated as well as constant relative positions for all inherited variables guaranteed. In the examples above it would look like this:

```
syn N ()
```

```
        a (V) Gen1 (V V1) M (V1)
    ] a (V) b () Gen2 (V V1) M (V1)
end
syn Gen1 (XY)
    => Y = X
```



```
end
syn Gen2 (XY)
    => Y = X
end
syn R ()
    a (V) Gen3 (V V1) S (V1)
end
syn Gen3 (XY)
    => Y = {FX}
end
```

The non-terminals „Gen ... \ are different from all other grammar symbols
the; they are called marker non-terminals. Your parameter X is inherited and Y is an inherited one
synthesized attribute. It is guaranteed that these non-terminals are only used once in the
Grammar are applied, the relative position of all inherited attributes in the par-
sestack constant after this transformation. The limitation, however, is because here
possibly parsing con icts can be introduced into the grammar. In this case mu
the user rephrase his grammar.

With the expansion of the production schemes presented below, the case can also be
occur, as alternatives are nested in sequences; also can thereby
semantic actions are at any point in a sequence (so-called mid-rule
as opposed to end-rule actions). LR parsing tables can only be generated from strict BNF
nested alternatives and mid-rule actions must be in their own rules
be outsourced. The difficulty is because this is based on local variables of the surrounding
Can access sequences. The de nitions of the initialized local variables and the
EBNF factors no longer apply here. The solution is simple: when outsourcing,
which the referenced local variables are added to the parameter list. The derivation of the
Attribute types and the inherited attribute mechanism then do the rest.

Comparison with LL parsers

The parsing algorithm restricted by the transformation no longer has the
full power of LR if these features are used, but is still min-
at least as powerful as LL. This is because LR is just as powerful as LL, if
A marker non-terminal is inserted at the beginning of every BNF production. One and only
The difference in semantics is that in case of doubt the values of inherited attributes
bute are always evaluated if a non-terminal could follow that needs them.
Therefore, when calculating the values for inherited attributes, neither side e ects
produces functions that are not yet monotonous, if the speci cation is un-
depending on whether an LL or an LR parser is used, have the same semantics
should.

5.3.3 Production schemes

This section describes the syntax rules de ned above for production schemes
expanded. These offer the user the opportunity to de-
kidneys.

In a first step, it is explained here how production schemes are speci ed
the. Their expansion is then discussed and some examples are given.

De nition

Production schemes can be de ned in two places: on the global level
a file and within a grammar. The global de nitions are in the list
of the prede ned production schemes included in each following grammar
are valid. Local production schemes only apply within the grammar in which they
be de ned.

h top level expression i + = h prod clause i
h grammar descriptor i + = h prod clause i

Like most other grammar descriptors, there is a production scheme
from a head and a torso. The body specifies the EBNF expression that is used in
Application of a production scheme is to be used. This can be optional local
Reference syntax rules.

h prod clause i :: = prod h prod head i h local rules i in] h syn alt i end
h local rules i :: = h syn clause ifh syn clause i g

The head of a production scheme also provides the names of its arguments
a unique identification of the schema. This consists of the following components
together:

1. the fact whether the scheme if it is used in the place of an EBNF factor
   returns a value (noted by „V =. . . \, where V denotes the variable whose
   Value is delivered),

2. the name that the scheme may have, noted in front of a colon,

3. the pair of brackets used (round, square or curly brackets),

4. the number of arguments separated by // and

5. the Post xoperator used, if one was specified.

For example, X] is a common EBNF notation for an option and {X // Y} +
could stand for a separated list with at least one element. This construct
could result in a value, for example an Oz list of the values returned by the X, what by
Z = {X // Y} + is noted.

h prod head i :: = h template de nition i
jh variable i = h template de nition i
h template de nition i :: = h prod formal list i
jh atom i : h prod formal list i
h prod formal list i :: = (h prod formals i) h prod post x i]
jh prod formals i] h prod post x i]
j { h prod formals i } h prod post x i]
h prod formals i :: = h variable i f // h variable i g]
h prod post x i :: = +
j *

expansion

The instantiation of a production scheme is possible wherever an EBNF factor
may stand:

h syn factor i + = h template instantiation i

The application of a production scheme resembles its de nition, only because instead

of the formal parameter variables, current EBNF expressions are permitted.

h template instantiation i :: = h prod actual list i
jh atom i : h prod actual list i
h prod actual list i :: = (h prod actuals i) h prod post x i]
jh prod actuals i] h prod post x i]
j { h prod actuals i } h prod post x i]
h prod actuals i :: = h syn old i f // h syn old i g]

In the expansion of production schemes, the capturing of variables must be avoided , i.e. the emergence of a con ict between identical local variables of the Production schemes and free variables of the arguments. The expansion will be in the following Sub-steps carried out:

The local variables of the production scheme are clearly renamed, both in the EBNF expression as well as in the local rules. Between free variables of the schema and other variables, variable capturing cannot occur if It is assumed that all variables are unique in the entire abstract syntax tree have been renamed.

The local rules are clearly renamed to avoid confusion with existing avoid the grammar symbols.

The current EBNF expressions are used for the parameter variables of the production schemes used. The formal parameter variables are allowed as grammar symbols are used in applications and have either no or one argument.
If they have an argument, the current EBNF expression must return a value, which is unified with the argument.

The local rules are determined by the local rules used in the current parameters. cal variables of the calling syntax rule are quanti ed by giving them parameters to be added. Their attribute types are determined using the method de ned above. automatically determined.

The local rules are entered in the table of grammar symbols.

The scheme call is made by the EBNF printout of the production scheme puts.

In order to ensure the termination of the expansion of production schemes, the in the de nition of a schema, EBNF expressions used only on previous pro

make use of production schemes. After all production schemes have been expanded
some grammar simplifications are made; be for example
Alternatives and sequences from eight and consecutive semantic actions
in summary. After that, all constructs are outsourced to their own rules, provided that this is the case
is necessary to maintain a strict BNF.

The local rules allow access to the current printouts to local varia-
the call location become non-local access. You can see how easy this is through the
Mechanism of automatic attribute type derivation is handled.

Examples:

The simplest production scheme is the bracketing construct:

```
prod (X)
    X ()
end
```

For example, the sequence would be

```
T in ('INT' => T = intTp] 'REAL' => T = realTp)
```

expands to T in Gen (T), where Gen is a new nonterminal with the following
The rule is:

Page 101


```
syn gene (T)
          'INT' => T = intTp
       ] 'REAL' => T = realTp
end
```

The bracketing construct can also return a value:

```
prod V = (X)
    X (V)
end
```

The following construct implements the EBNF operator „Option\:

```
prod X]
```

```
    X] true
end
```

The following example is intended to illustrate how production schemes are used to
It can be possible to specify e cient formulations for a construct, which
look different depending on the parsing algorithm. Through production schemes, this
However, this implementation detail is hidden from the user.

In the example, the optional repetition is implemented, which is a
Returns the list of values returned by the repeated element. How-
repetitions must be made in LL grammars in BNF by right-recursive rules
be formulated:

```
prod V = {X}
    syn N ($)
            true => nil
        ] X (A) N (As) => A | As
    end
in
    N (V)
end
```

Each element is placed in front of the result list after it has been recognized.
hangs. This implementation could also be used for an LR parser
the; there, however, left-recursive rules are preferred, since with these the storage
The memory consumption of the parsestack is constant and not linear in the number of
Elements of repetition. To make the list construction efficient anyway
hold, di erence lists are used.


```
prod V = {X}
    syn N (Hd Tl)
            true => Hd = Tl
        ] N (Hd Tl0) X (Elem) => Tl0 = Elem | Tl
    end
in
    N (Hd Tl) => Tl = nil V = Hd
end
```

5.3.4 Predefined classes and production schemes

The front-end generator generates some features and a class from a grammar
Methods for these, in which the parsing tables and the behavior in the event of reductions (i.e. the semantic actions) are saved. The prede ned class, SynBaseClass`
operates on this data.

The 'SynBaseClass' class

The mixin class, SynBaseClass` implements the table parser and provides methods available through which its behavior can be controlled. If there is inheritance from her, the syntax rules of a grammar can be used to analyze an input.

The class SynBaseClass requires the following methods of derived classes:

lexYield (+ AX (Unit) Method]

This method is to send a token of token class A with the value X to the token stream It is used to add an internal terminal symbol to the
inherited attributes of the start symbol.

lexGetToken (? A? X) Method]

The parser uses this method to request a new terminal symbol. She mu min-
At least one token type A can be returned and this can also be given a token value in X assign.

synExecuteAction (I Xs? Ys? Z) Method]

This method is called by the parser when a reduction is performed.
It receives the number I of the BNF rule and the current parsestack Xs. You must the parsestack Ys after the reduction and the generated semantic value Z
hand back. Since the representation of the attributes and local variables is implicit here on the parsestack, the user does not need to use this method
k ummern: It is generated automatically by the parser generator.

There are also features for the parsing tables that are automatically generated by the parser generator be generated. They all start with, syn. . . `.

'SynBaseClass' has the following prede ned functionality:

synLookaheadSymbol Attribute]

The token type of the currently considered lookahead token is entered in this attribute saved.

synLookaheadValue Attribute]
The token value of the currently considered lookahead tokens saved.

synNoLookahead Feature]
This value is saved in "synLookaheadSymbol" if there is currently no lookahead head token is present. This is the case when the token has been consumed, but still no new one was requested. In error situations, the lookahead token can use this value can be deleted.

synParse (T? B) Method]
This method starts the parse process. The label of the tuple T gives the start symbol, its features the parameters of the corresponding non-terminal. The WerTe of the inherited attributes of the start symbol are taken from this tuple and the synthesized attributes after parsing with the rest
Partial trees of the tuple uni ed.

synAccept () Method]
With this method, the parsing process is canceled and the status is positive reported. It can then be used if the rest of the input is ignored shall be.

synAbort () Method]
With this method the parsing process is aborted and the status is negative reported. It can then be used when an error situation has been detected, in which no error recovery should be carried out.

synRaiseError () Method]
This method puts the parser in the same state as if a syntax errors have been detected. The parser continues in error recovery mode.

synErrorOK () Method]
This is used to inform the parser's error recovery mode, since the error recovery action is complete and normal parsing can continue.

synClearLookahead () Method]
This method clears the current lookahead token. This comes in handy when that

Lookahead token in the event of a restart-point specified by the nonterminal error

is not valid.

synError (+ V) Method]

The parser calls this method when an error message V is output the should. It can be overloaded to format this output or further processing.

Predefined production schemes

Table 5.2 shows the predefined production schemes. It's all EBNF constructs which were described in section 3.1.3. For each operator there are several rere synonymous spellings. The operators all also exist as forms that return a value: The brackets return the value of their argument; the option too or nil if it is not chosen; the repeat constructs provide Oz-Lists of their first argument.

construct	Scheme	Synonyms
grouping	(A)	|
option	A]	|
Compulsory repetition (A) +		{A} +
Optional repetition	(A) *	{A}, {A} *
Compulsory separated Repetition	(A // B) + (A // B), {A // B} +, {A // B}	
Optional separated Repetition	(A // B) * {A // B} *	

Table 5.2: Predefined production schemes

5.3.5 Compiler directives

Table 5.3 gives an overview of the compiler directives that provide additional output Get information about the parser generation. The output is in files that contain by the name of the grammar with the appended .simplified or .output are formed.

Switch name	Default meaning	
synoutputsimpli ed	O	Causes an output of the BNF form (.simplified)
synverbose	O	Causes the machine to issue (.output)

Table 5.3: Compiler directives for controlling the parser generation

5.4 Substitution rules

In this section, the extension that was examined in section 4.1 is implemented. Section 5.4.1 shows how replacement rules fit into the existing syntax classify. Section 5.4.2 explains their implementation. With a few examples, the Description of the replacement rules in Section 5.4.3 completed.

5.4.1 Concrete syntax

A replacement rule is a special semantic action.

h syn action i + = => h rewrite action i

Your left side is the EBNF sequence, at the end of which it stands. The right side is through given an application and a sequence of terms. The application says with which Nonterminal the term sequence is to be parsed. Each term is either a tuple {and then corresponds to an application {or a variable. This must be a local variable of the Be production that was implicitly declared as the only parameter of an application; they then stands for this application.

h rewrite action i :: = rewrite h syn application ifh term i g end

5.4.2 Realization

Every replacement rule is transformed into a semantic action. This is done in a additional phase at translation time done: With the information from the replacement After analyzing the grammar and transforming it in BNF, the rules will be temporary Generates parsing tables that are only used to resolve the right-hand pages. Thereafter the parser generation can be continued. In this section, the additional phase described informally.

First of all it has to be determined which non-terminals for parsing the right sides used in grammar. These are then the start symbols of the temporary Parsing tables. Furthermore, all non-terminals N i are determined which are on a right Side to be applied. A fresh terminal symbol a i is used for each non-terminal N i which replaces all occurrences of N i in the right-hand sides of the replacement rules will. Furthermore, the grammar around the productions N i ! a i expanded. If the Grammar did not contain any con icts, then this also applies to the extended grammar, for which now a parser is generated.

With this parser every right side can be parsed: Let S be the empty semantic one Action. With every reduction with a production i, the corresponding semantic Action S i not executed, but appended to S. Thereby attribute references

adapted accordingly so that it does not refer to the parsestack but to the specified one Relate symbol sequence. After parsing every right side, S contains the semantic Action that replaces the rewrite action.

It is obvious that replacement rules in this algorithm are not recursively de- can be ned. This case must be caught and reported as an error.

5.4.3 Examples

In this section, some replacement rules from the Oz notation are presented as examples. posed. The semantic actions that build an abstract syntax tree were omitted for reasons of clarity.

The first example is the syntax of the method application, in which the object specification can be omitted if it is self:

```
syn methapply ($)
        '<<' Term (X1) Term (X2) '>>' => ...
    ] '<<' Term (X) '>>'
        => rewrite methapply ($)
                '<<' 'self' X '>>'
            end
end
```

The next example shows how the shorthand notation for the declaration of local variables blen in procedure umpfen etc. can be expanded.

```
syn Expression ($)
        'local' Expression (E1) 'in' Expression (E2) 'end' => ...
    ] ...
end
syn InExpression ($)
        Expression ($)
    ] Expression (E1) 'in' Expression (E2)
        => rewrite Expression ($)
                'local' E1 'in' E2 'end'
            end
end
```

This replacement rule implements the example from Section 4.1. It's a little here more complicated, since instead of a simple then branch there is also a sequence of ... elseof ... is allowed.

```
syn CaseExpression ($)
    'case' Expression (E) ElseOfList (EOs) ElseCases (Cs) 'end' => ...
end
syn ElseCases ($)
        'elsecase' Expression (E) ElseOfList (EOs) ElseCases (Cs)
        => rewrite ElseCases ($)
                'else' 'case' E EOs Cs 'end'
            end
      ] 'else' InExpression ($)
      ] true => ...
end
```

5.5 Bond Analysis

This section introduces the implementation of the bond analysis, which is discussed in Section 4.2. was searched. Section 5.5.1 explains the declarative part, namely the grave matic annotations (based on their specific syntax). The operational part in the form of the The predefined class "BindingAnalysis" is described in Section 5.5.2.

The speci cation may accompany the explanations as a mild example Section 5.1.2 must be read.

5.5.1 Concrete syntax

Section 4.2 explained how visibility areas are divided into blocks. A EBNF factor can represent such a block. The type of variable after the keyword scope, for which this block applies, and a Term after end that is uni ed with a list of pairs that denote the renaming of the block contains.

⟨ syn factor ⟩ + = ⟨ scope start ⟩ ⟨ syn alt ⟩ ⟨ scope end ⟩
⟨ scope start ⟩ + = scope
j scope (⟨ variable sort ⟩)
⟨ scope end ⟩ + = end
j end (⟨ term ⟩)
⟨ variable sort ⟩ :: = ⟨ atom ⟩

The semantics of these annotations are de ned by the following rules. Abbreviated

Notations are completed by default values and then on the class, binary dingAnalysis`, which is presented in the next section.

scope) scope (default)
end) end (unit)
scope (S) ...) << openScope (S) >> ...
end (T) << closeScope (ST) >>

Annotations of the occurrence of variables are made after a colon on an application. This must be a terminal application with a variable as a parameter. The Variable takes the token value modified by the binding analysis.

⟨ syn application ⟩ + = ⟨ atom label ⟩ ⟨ syn actuals ⟩ : ⟨ occurrence kind ⟩

A distinction is made between binding and referencing occurrences. The annotation tion fresh means that there is a fresh identifier of the specified token type at this point should be generated. The term passed in addition to the variable type is transferred to the function passed on, which the lexeme generates; he can there for example in the construction a print name can be used.

⟨ occurrence kind ⟩ :: = binder
j binder (⟨ variable sort ⟩)
j reference
j reference (⟨ variable sort ⟩)
j fresh
j fresh (⟨ term ⟩)
j fresh (⟨ variable sort ⟩ ⟨ term ⟩)

Analogous to the blocks, the semantics of the occurrence of variables are also de ned by since semantic actions are introduced, the methods of the class "BindingAnalysis" call:

id (V): binder) id (V): binder (default)
id (V): reference) id (V): reference (default)
id (V): fresh) id (V): fresh (default unit)
id (V): fresh (T)) id (V): fresh (default T)
id (V): binder (S)) V = (id (X) => {@S mkBinder (id X $)})
id (V): reference (S)) V = (id (X) => {@S mkReference (id X $)})
id (V): fresh (ST)) V = (=> {@S mkFresh (id T $)})

Examples of this can be found in Section 5.1.2.

5.5.2 The "BindingAnalysis" class

The "BindingAnalysis" class represents the operational part of the binding analysis, the was already motivated in Section 4.2. Your goals can be found there.

In order to be able to rely on the representation of the variable
To remain independent of tokens are user-de ned methods in derived classes required to implement the necessary operations on the variable token:

anonymize (+ X? Y) Method]
This method is used to copy a token X and the attribute for the name replaced by an indeterminate variable; the other intrinsics are retained. The copy is returned in Y.

nameToken (+ X + A) Method]
An already anonymized token X is to be given the name A here.

generate (+ X? A) Method]
This method is called when a new variable identifier A is required. The method must return a different result each time it is called. The entered Term X comes from the grammar annotations and can be used to assign a print name can be used.

tokenToAtom (+ X? A) Method]
In order to be able to distinguish between variable tokens internally, they are represented by atoms represented. This method takes care of the conversion.

makeTokenFromAtom (+ A? X) Method]
If a fresh variable name Aber was generated as an atom, it will be Converted to a variable token X using this method. The other intrinsics than the name should be set to default values.

The "BindingAnalysis" class de nes the following methods for the user:

init () Methods]
initializes the internal structures that save the renaming.

openScope (+ A) Method]
o net a Scopus of the variable type A.

closeScope (+ A? X) Method]
closes a scope of variable type A. In X a list of pairs is returned.

given that map atoms to atoms. These are those performed in the block Renaming.

mkBinder (+ A + X? Y) Method]

declares a token with type A and value X as a binding occurrence of a variable, to be renamed. The renamed variable token is returned in Y ben.

mkReference (+ A + X? Y) Method]

declares a token with type A and value X as a referencing occurrence of a variable ble. The renamed variable token is returned in Y. Maybe it still is anonymous and will only be named later.

mkFresh (+ A + X? Y) Method]

requests the generation of a variable token with token type A. In X, any ge user data from the grammar annotation are passed, such as a Provide Pr ax for the print name. The generated variable token is returned in Y give.

enterSubstitution (+ A1 + A2) Method]

enters a renaming from atom A1 to A2 in the renaming function. This can be used to identify implicitly declared identifiers (e.g. by a Import directive) in the binding analysis. The substitution must be there come from a list supplied by 'closeScope'.

5.6 implementation

As shown, the specification language of the tool itself has a relatively extensive input language. The expansion of the production schemes continues to lose itself the unique renaming of the identifier. So it is obvious to use the system below Use his own to implement. Since in a first step it is not at all exists, a so-called bootstrapping process is necessary. This is done with as little A moderate prototype of the tool is created according to the effort, although not necessarily the offers full functionality, but with which the final version can be translated.

In the present case, this was realized because the front end of the existing DFKI-Oz-Compilers was modified. This was done in C ++ using ex and Bison and created an abstract syntax tree in tuple form as a tele- Values. Tel is the programming language in which the remaining compiler phases are implemented are. However, a version that emerged from it already existed, which instead was Oz-

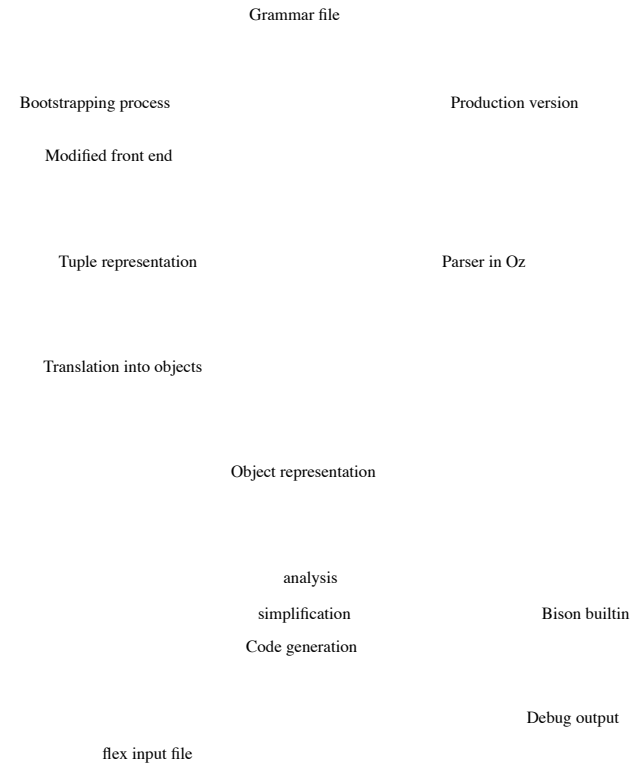
Tuple builds up.

The latter was extended to include the syntax rules for the grammar specification expanded. However, everything was left out that had to do with the bond analysis or the Replacement rules attached. The modified front end creates its abstract syntax also in tuple form; was a requirement when implementing the tool but to use an object-oriented abstract syntax. Therefore the tuple form must follow Completion of the parsing process, whereby the grammar-specific cations can be converted into objects. The final tool can now be placed here and perform the necessary transformations to create an executable Oz program receive.

Figure 5.2 shows the architecture of the tool. The phases are through dashed rectangles, which are used in the bootstrapping process and in the duction version of the tool run differently.

The figure also shows that for the generation of the scanner back to ex is accessed by generating an input file for it. The one created from it

Page 111



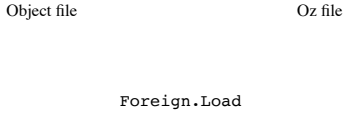


Figure 5.2: The architecture of the tool

Scanner is sent to the user agent via Oz's foreign-function interface.
tied up. Bison is used in a similar way to generate the parsing tables. Instead of
To communicate via files, however, Bison was modified and used as an Oz built-in
Provided. The grammar transformed in BNF is transferred to the builtin in tuple representation
position passed, which in turn returns the parsing tables generated from them as tuples.
supplies. Bison's table parser has been translated into Oz so that the foreign-function-
Interface is not necessary for this at runtime.

Chapter 6

Summary and Outlook

This chapter summarizes the results of the work. First of all,
Section 6.1 rated the tool developed. A check is carried out to determine whether the requirements in Section 1.4
requirements are met. Based on the experience gained, we will continue to
judges which aspects in the system have been implemented well or not so well.

Section 6.2 gives an outlook on the areas in which work should still be done,
to improve the tool.

6.1 Evaluation of the system

To validate the system, a front-end for its own specification language was created
created. Subset of this language is the entire syntax of Oz; So that's it
a nontrivial test example. Based on the experience gained in doing so,
In this section the quality of the developed front-end generator is evaluated.

In the following, the system is validated against the global requirements that are
Section 1.4 were formulated.

Supported phases. The front-end generator supports all phases that are planned
were, with the exception of the automatic construction of an abstract syntax. For this
Further research is necessary (see Outlook).

Independent usability. Thanks to the class-based approach, the
Usage of scanner and parser (with its extensions) possible. Also
the free combination of scanners and parsers without having to store them multiple times
had to be more secure, is allowed by inheritance.

High level of integration. The user has to be concerned with the interfaces between the scanner and
For most applications, parsers do not take care of the coupling from the tool

is adopted {regardless of whether the lexical and the syntactic ana-
lyse were de ned in the same or in different grammars and by
Inheritance can be coupled. Furthermore, the processes that are interlinked with parsing are
The following phases are specified directly in the grammar in their own concrete syntax.
Thus, the requirement for high integration is met.

Embedding in Oz. Wherever semantic calculations are necessary, Oz syntax is used
is used, so the full power of this language can always be used.
Furthermore, grammar definitions are divided into classes according to their transformation
a good knowledge of the language. This means that the embedding in Oz can be rated as successful
will.

Information hiding. Since the user has his own method endings in the grammar
bring in a class, let any class inherit it, or specialize it

can, for user-de ned operations all possibilities of information hi
thing available from Oz. On the part of the parsing tables, however, are in this relationship
Improvements still possible: To change the predefined parser class from the generated
To be able to decouple the class, the features for storing the parse-
barking as well as the generated methods are public and are therefore from the outside
accessible. If system variables were used this could be avoided.

Interactive systems. Care was taken with every part tool, as it
does not restrict the manageability of interactive systems. The parser is deterministic
and the replacement rules are interlinked with the parsing as soon as
they are applicable. The binding analysis does not need semantic actions either
to delay, since with undetermined variables as placeholders for the names of
Identifiers will continue.

Now the advances made over existing tools will be
summarized.

Production schemes have been developed that are not in this form in the literature
were found. An important feature is theirs (not with Combinator Parsing
given) static expansion in BNF. This has several advantages. For one, can
the actually recognized language can thus be formally described. On the other hand
this means that the use of production schemes does not cause any loss of e ciency
causes, but on the contrary for simple EBNF operators complex and
e cient implementations can be provided.

The automatic derivation of the attribute types, i.e. whether a parameter of a product
tion represents a synthesized or an inherited attribute, allows very powerful
Extensions to the LR parsing procedure: The de nition and use became
local variables in rules possible and a step has been taken in the direction of
First made use of L- instead of only S-attributed grammars. That was

Page 114

the independence from the parsing technique has been significantly improved. Something equivalent
was not found in any existing tool; thus this represents a real one
Innovation.

A new system for specifying replacement rules has been developed. This
Fulfilled all the required properties, but unfortunately has not become as legible as
increases.

An existing system for specifying identifier bindings has been expanded.

tert and factored into a declarative and an operational part, whereby it became significantly more eighty.

Despite the power of the system, the e ciency is very satisfactory: The generic Generating a front end from a description is particularly fast because for the table generation on the tools ex and bison was used. There too If an ex scanner is used at runtime, the lexical analysis is not important slower than in the C programming language, since the execution of the semantic actions make up only a small part of her. The slowest at runtime is the syntactic cal analysis, since the table parser has been translated from Bison to Oz. Also the Memory consumption is only linearly greater than in C, since Oz is also used for methods in which the gotos had to be translated from the C version, tail calls supported.

6.2 Outlook

It is clear that the tool must now prove itself in practice. But already that Experience from the development and the test examples leave the desire for some Future investigations and expansions may arise.

In order to be able to abstract even further from the parsing technique used and easily offer different parsing techniques to choose from in a later version to be able to do this, further automatic transformations of the grammar have to be implemented. be mentored. These should be speci ed independently of each parsing technique Prepare grammar for the desired procedure.

The promising concept of dynamic operator tables discussed in Section 3.2.1 should be implemented.

In favor of better maintainability and greater similarity to Oz methods it should also be possible for the parameter lists of non-terminals instead of just Tuple syntax to also allow the record notation.

It would make sense to use one of the class as standard „BindingAnalysis \ terminated to offer a directed class that takes on full-fledged symbol table management.

The binding analysis does not yet have a specific syntax with which to declare could determine for which token types it should be carried out. The necessities Up to now, the user still has to explicitly explain and initialize class attributes. This should be automated.

In order to avoid renaming variable names during import and export,
certain variables occur during the binding analysis (for example those from the
global visibility area) can be marked with the property because they
should not be renamed. However, this would mean that the final
Names could only be determined after the entire parsing process had been completed.
It is conceivable to have an alternative class too „BindingAnalysis \ to offer this
supports.

Investigations still have to be made, such as an automatic set-up
object-oriented abstract syntax could not be realized under the
Disadvantages suffers, which were explained in Section 4.3.3.

A function should be included in the standard scope of the front-end generator
which outputs a virtual string obtained from (manual) unparsing.
The control tuples for the formatting as described in Section 4.4.3 should be
are correctly implemented.

Further studies should be carried out into the possibility of modularizing special
cations are made. It should be able to be inherited from a grammar
and these can be changed through extensions, deletions and modifications
can. However, there is a division of the grammar in fully compositional languages
only possible to a very limited extent in less coupled or even hierarchical subtasks
this goal has a low priority.

Appendix A.

The grammar notation used

This appendix explains the notation for context-free grammars used in Chapter 5 de nes the concrete syntax of the specification language.

It is an extended Backus-Naur form, which consists of the following components share composes:

Terminals and non-terminals are enclosed in angle brackets h ::: i ;

the left side is separated from the right by :: = or + =, where by + = Productions are added to an existing non-terminal;

the vertical bar separates alternatives;

the curly brackets note a 0 to n-fold repetition of your argument ments;

the square brackets mark a 0 to 1 repetition;

Literal strings are set in the Typewriter character set.

The special thing about this notation is that grammars are de ned incrementally can: An existing grammar can be expanded because its non-term New productions can be added via + =.

Elements from the Oz syntax

Since the tool's speci cation language is embedded in Oz, the same lexical conventions such as for Oz. An additional terminal is h regex i , the one for one regular expression. This is noted in ex syntax and enclosed in angle brackets (<and>) included.

The following terminals from the Oz syntax are used:

⟨ atom ⟩ stands for an Oz atom (no distinction is made between quoted or not).

⟨ variable ⟩ is an Oz variable.

⟨ atom label ⟩ is an Oz atom followed directly by opening round brackets becomes.

⟨ variable label ⟩ is an Oz variable which is directly enclosed in opening round brackets will follow.

Furthermore, some non-terminals from the Oz syntax are referenced:

⟨ label ⟩ stands for an Oz variable or an Oz atom.

⟨ escaped variable ⟩ stands for an Oz variable, which can optionally be preceded by an exclamation mark pr a - xed, which prevents an implicit declaration of the variables.

⟨ top level expression ⟩ is an expression at the top level of the program text.

⟨ expression ⟩ is an Oz expression.

⟨ term ⟩ stands for an Oz term.

⟨ class descriptor ⟩ stands for a permitted class descriptor , i.e. for one of the from, feat, attr, or meth clauses.

bibliography

Aas92] Annika Aasa. User De ned Syntax. PhD thesis, Department of Computer Sciences, Chalmers University of Technology and University of Gothenburg, 1992.

Ada83] Ada Joint Program O ce, US Government. Reference Manual for the Ada Language, ANSI / MIL-STD 1815a, 1983.

Age94] Ole Agesen. Mango: A parser generator for Self. SMLITR 94 {27, Computer Science Department, Stanford University / Sun Microsystems Laboratories, Inc. June 1994.

AMT92] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A Lexial Analyzer Generator for Standard ML. Princeton University, Version 1.4, October 1992.

And95] Andersson is different. SAGA User Manual. Programming Systems Group, Swedish Institute of Computer Science, June 1995.

ASU86] Alfred V. Aho, Ravi Sethi, and Je rey D. Ullman. Compilers {Principles, Techniques and Tools. Addison-Wesley, 1986.

Aug93] Mikhail Auguston. Rigal Programming System Language Description. Department of Mathematics and Computer Science, University of Latvia, July 1993.

Ave95] J urgen Avenhaus. Reduction systems. Springer textbook, Berlin, Heidelberg, 1995.

BB92] PT Breuer and JP Bowen. A prettier compiler-compiler: Generating higher order parsers in C. PRG-TR 20 {92, Programming Research Group, Oxford University Computing Laboratory, 1992.

Bea90] Steven J. Beaty. ParsesraP: Using one grammar to specify both input and output. ACM SIGPLAN Notices, 30 (2), February 1990.

Bev93] Stephen J. Bevan. Abstract syntax: Is there such a thing as the right one ?, January 1993.

Page 119

BP95] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little e place). TR 95 {09, Department of Computer Science of the University of Arizona, September 1995.

CCH95] James R. Cordy, Ian H. Carmichael, and Russell Halliday. The TXL Program ming language. Software Technology Laboratory, Department of Computing and Information Science, Queen's University, Kingston, Canada, April 1995. Commercial Distribution by Legasys Corp.

CMA94] Luca Cardelli, Florian Matthes, and Mart n Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, February 1994.

Co e93] Alain Co etmeur. Bison ++. R&D Department, Informatique-CDC, France, March 1993. Based on DS95, version 1.21].

Com96a] Compiler Tools Group, Department of Electrical and Computer Engineering, University of Colorado. Eli {Lexical Analysis, Revision 2.8, 1996.

Com96b] Compiler Tools Group, Department of Electrical and Computer Engineering, University of Colorado. Eli {Syntactic Analysis, Revision 1.10, 1996.

Dor96] Chris Dornan. lx {A Lex for Haskell Programmers. ftp://ftp.cs.bris.ac.uk/ users / dornan / lx.tar.gz, July 1996.

DS95] Charles Donelly and Richard Stallman. Bison {The YACC-compatible Par- this generator (Reference Manual). Free Software Foundation, version 1.25, November 1995. On-Line Info File.

DSH95] Charles Donelly, Richard Stallman, and Wilfred J. Hansen. Bison {the YACC-compatible parser generator (Reference Manual). Andrew Consortium, Carnegie Mellon University, Version A2.6, June 1995. Modes ed from DS95, Version 1.24].

EKVW94] Reinhard Eppler, Peter Knauber, Stefan Vorwieger, and Hans-Wilm Wipper- man. The Refus programming language. Internal report 253/94, AG Pro- programming languages and compilers, Department of Computer Science, University of Kaisers- Lautern, September 1994.

Fai87] J. Fairbairn. Making form follow function: An exercise in functional program ming style. Software {Practice and Experience, 17 (6): 379 {386, 1987.

GM96] Andy Gill and Simon Marlow. Happy, the Parser Generator System for Haskell, Version 0.9, February 1996. ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/happy /.

Gro87] Josef Grosch. Efficient generation of table-driven scanners. Compiler Generation Report No. 2, GMD research center at the University of Karlsruhe, May 1987.

Gro88a] Josef Grosch. Lalr {a generator for e cient parsers. Compiler Generation Report No. 8, GMD research center at the University of Karlsruhe, October 1988.

Gro88b] Josef Grosch. Selected examples of scanner specifications. Compiler genes ration Report No. 7, GMD research center at the University of Karlsruhe, March 1988.

Gro89] Josef Grosch. E cient and comfortable error recovery in recursive descent parsers. Compiler Generation Report No. 19, GMD research center at University of Karlsruhe, December 1989.

Gro91a] Josef Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 27, GMD research center at the university sit at Karlsruhe, August 1991.

Gro91b] Josef Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 26, GMD research center at the university sit at Karlsruhe, November 1991.

Gro92] Josef Grosch. Rex {a scanner generator. Compiler Generation Report No. 5, GMD research center at the University of Karlsruhe, July 1992.

Gro93] Josef Grosch. Ast {a generator for abstract syntax trees. Compiler genes ration Report No. 15, GMD research center at the University of Karlsruhe, August 1993.

GS88] Christian Genillard and A. Strohmeier. GRAMOL: A grammar description language for lexical and syntactic parsers. ACM SIGPLAN Notices, 23 (10): 103 { 122, 1988.

GV92] Josef Grosch and Bertram Vielsack. The parser generators lalr and ell. Compi- Generation Report No. 8, GMD research center at the University of Karls- rest, July 1992.

GW85] G. Goos and WM Waite. Compiler Construction. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1985.

HDB92] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Syntactic abstraction in Scheme. Technical Report # 355, Computer Science Department, Indiana University, June 1992.

Hen95] Martin Henz. The Oz notation. Oz Documentation Series, Programming Systems Lab, DFKI, May 1995.

Page 121

Hil94] Steve Hill. Continuation passing combinators for parsing precedence gram-Mars. Technical Report 24 {94, University of Kent, Computing Laboratory, Canterbury, UK, November 1994.

HMSW96] Martin Henz, Martin Muller, Christian Schulte, and J org Wurtz. The Oz Standard modules. Oz Documentation Series, Programming Systems Lab, DFKI, May 1996.

HPJW92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict purely functional programming languge (version 1.2). ACM SIGPLAN Notices, 27 (5), May 1992.

Hud96] Scott E. Hudson. CUP User's Manual. Graphics visualization and usability Center, Georgia Institute of Technology, version 0.9e, March 1996.

Jan94] Sverker Janson. AKL {A Multiparadigm Programming Language. PhD thesis, Swedish Institute of Computer Science, 1994.

Joh75] SC Johnson. Yacc {yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

JPB94] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. A multipara digm approach to compiler construction. ACM SIGPLAN Notices, 29 (9): 29 { 37, September 1994.

JW75] K. Jensen and N. Wirth. Pascal User Manual and Report. Springer-Verlag, New York, 1975.

Kle56] SC Kleene. Representation of events in nerve nets. In C. Shannon and J. McCarthy, editors, Automata Studies, pages 3 {40. Princeton University Press, 1956.

Klu91] Michael Klug. VisiCola, a model and a language for visibility control in programming languages. ACM SIGPLAN Notices, 26 (2): 51 {63, February 1991.

Kna96] Peter Knauber. The OCC system. Working group programming languages and compilers, Department of Computer Science, University of Kaiserslautern, 1996.

Knu91] Donald Ervin Knuth. The TEXbook, volume A of Computers & Typesetting. Addison-Wesley, May 1991.

KR78] Brian Wilson Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice-Hall Software Series, New Jersey, 1978.

Cow94] Guido K uhn. De nition and implementation of a scanner generator. Per-Project work, Department of Computer Science, University of Kaiserslautern, 1994.

Page 122

KVE94] Peter Knauber, Stefan Vorwieger, and Reinhard Eppler. Terminology of Translator construction. Internal report 255/94, AG Programming Languages and Compiler, Department of Computer Science, University of Kaiserslautern, October 1994.

KW95] Basim M. Kadhim and William M. Waite. Maptool {mapping between concrete and abstract syntaxes. CS-CU 765-95, Department of Computer Science, University of Colorado, February 1995.

Les75] ME Lesk. LEX {a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, 1975.

MAS96] MASTER Information Systems Corporation, 3596 Pimlico Drive, Pleasanton, CA 94588. The MUSKOX Software Engineering Tool, Version 2.0, July 1996. Product Description and Release Notes, http://misc-sun.mastersys.com.

May81] Brian H. Mayoh. Attribute grammars and mathematical semantics. SIAM Journal on Computing, 10 (3), 1981.

MW91] H. M ossenbock and N. Wirth. The programming language Oberon-2. Report 160, Institute for Computer Systems, ETH Zurich, May 1991.

N + 63] P. Naur et al. Revised report on the algorithmic language Algol 60th Communications of the ACM, 6 (1): 1 {17, 1963.

Par77] David Lorge Parnas. Use of abstract interfaces in the development of software for embedded computer systems. NRL Report 8047, Naval Research Laboratory, Washington DC, 1977.

Par95] Terence John Parr. Language translation using PCCTS and C ++ (a reference guide). ftp://ftp.parr-research.com/pub/pccts/Book/reference.ps, June 1995.

Pax95] Vern Paxson. ex {Fast Lexical Analyzer Generator, Version 2.5.2, April 1995. Unix On-Line Manual Page.

PDC91] TJ Parr, HG Dietz, and WE Cohen. PCCTS Reference Manual. School of Electrical Engineering, Purdue University, Version 1.0, August 1991.

PJ88] Simon L. Peyton-Jones. The Implementation of Functional Programming Languages. Prentice Hall, New York, 1988.

PQ95] Terence J. Parr and Russell W. Quong. LL and LR translators need k> 1 lookahead. ACM SIGPLAN Notices, 31 (2): 27 {34, July 1995.

PVGK93] Kjell Post, Allen Van Gelder, and James Kerr. Deterministic parsing of languages with dynamic operators. In Dale Miller, editor, Logic Programming Proceedings of the 1993 International Symposium, pages 456 {472, Vancouver, Canada, 1993. The MIT Press.

PW80] FCN Pereira and David HD Warren. De nite clause grammars for language analysis. Artificial Intelligence, 13: 231 {278, 1980.

RT88] T. Reps and T. Teitelbaum. The Synthesizer Generator Reference Manual. Springer-Verlag, Third edition, 1988. First edition, Cornell University, August 1985; Second edition, Cornell University, June 1987.

Sch] FW Schr oer. The Gentle compiler construction system. Http: // www. rst.gmd.de/gentle/.

Sch89] FW Schr oer. Gentle. In WM Waite, J. Grosch, and FW Schr oer, editors, Three Compiler Specifications. GMD {German National Research Center for Information Technology, 1989. GMD Studies No. 166.

Smo94] Gert Smolka. The de nition of Kernel Oz. Oz Documentation Series, Programming Systems Lab, DFKI, November 1994.

Str87] Bjarne Stroustup. The C ++ Programming Language. Addison-Wesley, Reading, Mass., 1987.

TA91] David R. Tarditi and Andrew W. Appel. ML-Yacc Users's Manual. School of Computer Science, Carnegie Mellon University and Department of Computer Science, Princeton University, Version 2.1, March 1991.

vEB93] Peter van Eijk and Axel Belinfante. The Term Processor Kimwitu {Manual and Cookbook. University of Twente, Eschente, The Netherlands, Version 7 for Kimwitu version 3.8, March 1993.

Wad90] Vance E. Waddle. Production trees: A compact representation of parsed programs. ACM Transactions on Programming Languages and Systems, 12 (1): 61 { 83, January 1990.

War77] David HD Warren. Implementing Prolog {compiling predicate logic programs. Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

War80] David HD Warren. Logic programming and compiler writing. Software { Practice and Experience, 10:97 {125, 1980.

WEKV93] Hans-Wilm Wippermann, Reinhard Eppler, Peter Knauber, and Stefan Vor cradle. W-Lisp language description. Internal report 237/93, department Computer science, University of Kaiserslautern, December 1993.

Wil79] R. Wilhelm. Attributed grammars. Computer science spectrum, 2 (3): 123 {130, 1979.

WKE96] Hans-Wilm Wippermann, Peter Knauber, and Reinhard Eppler. W Lisp Language report version 3. Version 2 is described in WEKV93], 1996.