Gump–A Front-End Generator for Oz

Leif Kornstaedt

mozart

Abstract

This manual describes Gump, the front-end generator for Oz. It reads Oz files with embedded scanner and/or parser definitions and produces Oz code as output in which these have been replaced by classes implementing scanners and/or parsers. The semantic actions in the specifications allow the full flexibility and expressivity of Oz to be used.

Credits

Cover illustration by Andreas Schoch
Mozart logo by Christian Lindig

License Agreement

Contents

1

Introduction

The Gump is a fictional 'creature' in the children's novel 'The Marvelous Land of Oz'[1] by L. Frank Baum, sequel to 'The Wonderful Wizard of Oz' [1]. It is a living flying machine assembled by the main characters from individually selected materials. The Tin Woodman fastens a gump's head to one end of the machine, explaining that it 'will show which is the front end of the Thing.'

Gump is also the name of the specification language and tool described in this handbook from the user's point of view. Like the gump's head in the novel, the language is used to specify front-ends in Oz, in this case the lexical and phrase structure of a language. The tool is the 'Powder of Life' that generates executable programs (Oz class definitions) from such specifications.

Overview

This manual consists of two parts. Chapter 2 describes the Gump Scanner Generator; Chapter 3 the Gump Parser Generator. Each chapter is divided in two sections, the first explaining the basic concepts by example and the second providing a detailed reference for the programmer.

Acknowledgements

The specification language and the software tool have been designed and implemented as the subject of my Master's Thesis [3] which has been supervised by Prof. Dr. Gert Smolka and Dipl.-Inform. Christian Schulte at the Programming Systems Lab at the Universität des Saarlandes, as well as by Prof. Dr. Hans-Wilm Wippermann at the University of Kaiserslautern.

The software makes use of the GNU variants of lex and yacc, called flex [4] and bison [2] respectively.

Oz is the name of a programming language developed at the Programming Systems Lab at the Universität des Saarlandes. Mozart is an implementation of Oz.

The cover illustration has been drawn by Andreas Schoch.

---

[1] `http://almond.srv.cs.cmu.edu/afs/cs.cmu.edu/user/rgs/mosaic/ozland-ftitle.html`

The Gump Scanner Generator

This chapter describes the Gump Scanner Generator. Its input consists of an Oz source with embedded scanner specifications; the output implements each scanner by an Oz class.

Definitions   A scanner is a program that performs lexical analysis, which means that it transforms a stream of characters into a stream of tokens. The text is read from left to right. During this process, sequences of characters are grouped into lexemes according to user-defined rules, specified by so-called regular expressions and associated semantic actions. An action computes tokens from a lexeme, each consisting of a token class and an optional token value, which are appended to the token stream. The process is iterated until the end of the character stream is reached.

This chapter first describes the basic principles of the Gump Scanner Generator by means of an example in Section 2.1. A more detailed reference is then given in Section 2.2.

2.1   Example

As a running example we will specify, throughout the manual, a front-end for a compiler or an interpreter for a small functional language Lambda. In this section we will define the scanner for this language, in Section 3.1 we build a parser on top of this scanner.

2.1.1   Writing a Scanner Specification

Figure 2.1 shows the specification of the sample scanner we will consider in this section. In the following we will examine this example line by line.

Class Descriptors   At the first glance the scanner specification closely resembles a class definition with some extra elements, introduced by the keyword `scanner` instead of `class`. This is intentional, since it will ultimately be replaced by a class. This is why all descriptors allowed in a class definition are also allowed at the beginning of a scanner specification. Consider the `from`, `attr` and `meth` constructs used in lines 2 to 10.

Figure 2.1 The `LambdaScanner` scanner specification.

```
declare
scanner LambdaScanner from GumpScanner.'class'
   attr LineNumber
   meth init()
      GumpScanner.'class', init()
      LineNumber <- 1
   end
   meth getLineNumber($)
      @LineNumber
   end

   lex digit = <[0-9]> end
   lex letter = <[A-Za-z]> end
   lex id = <{letter}({letter}|{digit})*> end
   lex int = <{digit}+> end

   lex <define>
      GumpScanner.'class', putToken1('define')
   end
   lex <lambda>
      GumpScanner.'class', putToken1('lambda')
   end
   lex <{id}> A in
      GumpScanner.'class', getAtom(?A)
      GumpScanner.'class', putToken('id' A)
   end
   lex <{int}> S in
      GumpScanner.'class', getString(?S)
      GumpScanner.'class', putToken('int' {String.toInt S})
   end
   lex <"."|"("|")"|"="|";"> A in
      GumpScanner.'class', getAtom(?A)
      GumpScanner.'class', putToken1(A)
   end

   lex <[ \t]> skip end
   lex <"%".*> skip end
   lex <\n>
      LineNumber <- @LineNumber + 1
   end
   lex <.>
      {System.showInfo 'line '#@LineNumber#': unrecognized character'}
      GumpScanner.'class', putToken1('error')
   end

   lex <<EOF>>
      GumpScanner.'class', putToken1('EOF')
   end
end
```

Lexical Abbreviations The scanner-specific declarations begin at line 12. Two kinds of definition can be introduced by the keyword **lex**: either a lexical abbreviation, as seen in lines 12 to 15, or a lexical rule as found from line 17 to the end of the specification. A lexical abbreviation **lex** *I* `=` *<R>* **end** associates an identifier *I* with a given regular expression *R*. Occurrences of {*I*} in other regular expressions are then replaced to `(`*R*`)`.

Note that regular expressions use the same syntax as regular expressions in flex [4], with a few exceptions (detailed in Section 2.2.1). Furthermore, we must either enclose them in angle brackets or give them as Oz strings. (The latter proves useful when the angle-bracketed version confuses Emacs' fontification mode, but is a bit harder to read, since more characters must be escaped.)

The example defines four lexical abbreviations: `digit` stands for a decimal digit, `letter` for an uppercase or lowercase letter; `id` defines the syntax of identifiers to consist of a letter, followed by an arbitrary sequence of letters and digits; and finally, `int` defines the syntax of positive decimal integers as a nonempty sequence of digits.

Lexical Rules Lexical rules of the form **lex** *<R>* *S* **end** are more interesting, since the set of these is the actual scanner specification. Upon a match of a prefix of the input character stream with the regular expression `R`, the statement `S` is executed as a method body (i.e., **self** may be accessed and modified). Two methods are provided by the mixin class `GumpScanner.'class'` (inherited from in line 2) to append tokens to the token stream: `putToken1`, which appends a token of a given class without a value (**unit** being used instead), and `putToken`, which allows a specific token value to be provided. Token classes may be represented by arbitrary Oz values, but the parser generator in Chapter 3 expects them to be atoms. In lines 18 and 21 you can see how constants are used as token classes. In line 33 the token class is computed from the lexeme.

Accessing the Lexeme The lexeme itself may be accessed in several ways. The method `getAtom` returns the lexeme as an atom, which is the representation for identifier token values chosen in line 25. The method `getString` returns the lexeme as a string, such as in line 28, where it is subsequently converted to an integer.

The remaining lexical rules are easily explained. Lines 36 and 37 respectively describe how whitespace and comments are to be ignored. This is done by neither calling `putToken1` nor `putToken`. (Note that an action can also invoke them several times to append multiple tokens to the token stream, just as it may chose not to invoke them at all to simply ignore the lexeme or only produce side effects.) The rule in line 38 ignores any matched newlines, but updates the line counter attribute `LineNumber` as it does so. The rule in line 41 reports any remaining unmatched characters in the input as lexical errors and returns the token `'error'` which the parser can recognize as an erroneous token.

End-of-File Rules The final rule, in line 46, has the special syntax **<<EOF>>** (it might also have been written as `"<<EOF>>"`) and only matches the end of the character stream. It returns the token `'EOF'` which can be recognized by the parser as the end of input. Note that the action might just as well open another file to read from.

More information about acceptable sets of regular expressions in scanner specifications, conflict resolution and grouping into lexical modes is given in Section 2.2.1.

2.1.2 Invoking Gump

Now that we have finished writing our specification, we want to translate it into an Oz class definition that implements our scanner. For this, we issue the compiler directive

```
\switch +gump
```

whereupon the compiler will accept Gump specifications.

Running Gump   Save the above specification in a file `LambdaScanner.ozg`. The extension `.ozg` indicates that this file contains Oz code with additional Gump definitions, so that Emacs will fontify Gump definitions correctly. Feeding

```
\insert LambdaScanner.ozg
```

will process this file. Switch to the Compiler buffer (via `C-c C-c`) to watch Gump's status messages and any errors occurring during the translation.

Output Files   When the translation is finished, you will notice several new files in the current working directory. These will be named after your **scanner** specification. Suppose your scanner was called `S`, then you will find files `S.l`, `S.C`, `S.o` and `S.so`. The first three are intermediate results (respectively the input file for flex, the flex-generated C++ file and the object code produced by the C++ compiler) and the last one is the resulting dynamic library used by the generated scanner.

Implementation Limitation   Note that due to limitations of dynamic linking, a scanner may only be loaded once into the system. When interactively developing a scanner, this means that you will not see changes you make to the set and order of the regular expressions consistently. You should thus halt and restart Mozart each time you make changes to the regular expressions.

See also Section 2.2.2 for a workaround around this limitation.

2.1.3 Using the Generated Scanner

Figure 2.2 shows a sample program running our generated scanner.

The generated `LambdaScanner` class is instantiated as `MyScanner`. We have to call the method `init()` first to initialize the internal structures of the `GumpScanner.'class'`.

Requesting Tokens   The procedure `GetTokens` repeatedly invokes the `GumpScanner.'class'` method

```
getToken(?X ?Y)
```

which returns the next token's token class in `X` and token value in `Y` and removes it from the token stream. `GetTokens` exits when the end of the token stream is reached, which is recognized by the token class `'EOF'`.

Figure 2.2 A program making use of the generated scanner.

```
\switch +gump
\insert gump/examples/LambdaScanner.ozg

local
   MyScanner = {New LambdaScanner init()}
   proc {GetTokens} T V in
      {MyScanner getToken(?T ?V)}
      case T of 'EOF' then
         {System.showInfo 'End of file reached.'}
      else
         {System.show T#V}
         {GetTokens}
      end
   end
in
   {MyScanner scanFile('Lambda.in')}
   {GetTokens}
   {MyScanner close()}
end
```

Providing Inputs   To actually start scanning we have to provide an input character stream. This is done via one of the methods

```
scanFile(+FileName)
```

or

```
scanVirtualString(+V)
```

Each of these pushes the currently used buffer (if any) upon an internal stack of buffers and builds a new buffer from the given source. Each time the end of a buffer is reached, the `<<EOF>>` rule is matched. This may pop a buffer and continue scanning the next-outer buffer where it left off, using the `closeBuffer` method described in Section 2.2.3.

Closing Scanners   When a scanner is not used anymore, it should be sent the message

```
close()
```

so that it can close any open files and release any allocated buffers. (This is even necessary when scanning virtual strings due to the underlying implementation in C++.)

The following is a sample input for the scanner. The above example expects this to be placed in the file `Lambda.in` in the current directory:

```
% some input to test the class LambdaScanner
define f = lambda y.lambda z.(add y z);
define c = 17;
f c 7;
((f) c) 7
```

2.2   Reference

This section is intended to serve as a reference for the user of the Gump Scanner Generator. It details the syntax of the embedded scanner specification language in Section 2.2.1, which options are supported and how they are specified in Section 2.2.2 and finally the runtime part of the Scanner Generator, the mixin class `GumpScanner.'class'`, in Section 2.2.3.

2.2.1   Syntax of the Scanner Specification Language

The notation used here for specifying the syntax of the specification language is a variant of BNF and is defined in Appendix A.

A scanner specification is allowed anywhere as an Oz statement:

⟨statement⟩   +=   ⟨scanner specification⟩

It is similar to a class definition, except that it is introduced by the keyword **scanner**, must be named by a variable (and not an arbitrary term), since this is used for assigning file names, and allows for additional descriptors after the usual class descriptors.

⟨scanner specification⟩   ::=   **scanner** ⟨variable⟩
                                 { ⟨class descriptor⟩ }
                                 { ⟨method⟩ }
                                 { ⟨scanner descriptor⟩ }+
                                 end

A lexical abbreviation associates an identifier with a regular expression, which can then be referenced in subsequent lexical abbreviations or any lexical rules by enclosing the identifier in curly brackets. The regular expression is additionally parenthesized when it is expanded.

⟨lexical abbreviation⟩   ::=   **lex** ⟨atom⟩ '=' ⟨regex⟩ **end**
                          | **lex** ⟨variable⟩ '=' ⟨regex⟩ **end**

The definition of a lexical rule is similar to the definition of a method. However, its head consists of a regular expression; when this is matched, the body of the lexical rule is executed (as a method).

⟨lexical rule⟩   ::=   **lex** ⟨regex⟩
                       ⟨in statement⟩
                       end

Regular expressions may be annotated with lexical modes. Each lexical mode constitutes an independent sub-scanner: At any time a certain mode is active; in this mode only the regular expressions annotated with it will be matched. All lexical rules defined within the scope of a lexical mode are annotated with this lexical mode. A lexical mode may inherit from other lexical modes; all regular expressions in these modes are then annotated with the inheriting lexical mode as well. Lexical modes implicitly inherit from all lexical modes they are nested in. Lexical rules written at top-level are annotated with the implicitly declared mode INITIAL.

$$\langle\text{lexical mode}\rangle \quad ::= \quad \textbf{mode} \ \langle\text{variable}\rangle \ [\ \textbf{from} \ \{ \ \langle\text{variable}\rangle \ \}+ \ ]$$
$$\{ \ \langle\text{mode descriptor}\rangle \ \}$$
$$\textbf{end}$$

$$\langle\text{mode descriptor}\rangle \quad ::= \quad \langle\text{lexical rule}\rangle$$
$$| \quad \langle\text{lexical mode}\rangle$$

2.2.1.1 Syntax of Regular Expressions

Regular expressions $\langle\text{regex}\rangle$ correspond to the regular expressions used in flex [4] with a few exceptions:

- Gump regular expressions are either enclosed in angle brackets or given as Oz strings.

- The angle-bracket annotation with lexical modes is not supported by Gump; use scopes of lexical modes instead. Note that several distinct lexical mode definitions may occur for the same lexical mode name as long as no inheritance cycles are created.

Due to the underlying use of flex, the names of lexical abbreviations are restricted to the syntax allowed in flex name definitions.

2.2.1.2 Ambiguities and Errors in the Rule Set

Tokenization is performed by a left-to-right scan of the input character stream. If several rules match a prefix of the input, then the rule matching the longest prefix is preferred. If several rules match the same (longest) prefix of the input, then two rules may be applied to disambiguate the match (see Section 2.2.2 on how to select the rule):

First-fit. The rule notated first in the scanner specification is preferred. In this case, every conflict can be uniquely resolved. Two errors in the rule set are possible: holes and completely covered rules (see below).

Best-fit. Suppose two conflicting rules are rule r_1 and rule r_2, which are annotated by sets of lexical modes S_1 and S_2 respectively. Then r_1 is preferred over r_2 if and only if the following condition holds:

$$S_1 \subseteq S_2 \ \wedge \ L(r_1) \subseteq L(r_2)$$

where $L(r)$ is the language generated by a regular expression r, that is, the set of strings that match r. Intuitively, this rule means that r_1 is 'more specialized than' r_2. Additionally to the errors possible in the rule set in the first-fit case, here the situation may arise that the rule set is not well-ordered wrt. the 'more specialized than' relation.

The following errors in the rule set may occur:

Holes in the rule set.   For some input (in some mode), no true prefix is matched by any rule. Due to the underlying implementation using flex, this will result in the warning message

```
"-s option given but default rule can be matched"
```

If at run-time some such input is encountered, this will result in an error exception

```
"flex scanner jammed"
```

Completely covered rules.   A rule r is never matched because for every prefix in $L(r)$ exists another rule s which is preferred over r.

Non well-orderedness.   Two rules r_1 and r_2 are in conflict in the best-fit case, but neither is r_1 more specialized than r_2 nor the other way round, and no rule or set of rules exists that covers $L(r_1) \cap L(r_2)$.

2.2.2 Parameters to Scanner Generation

The Gump Scanner Generator supports several configuration parameters, which may be set on a per-scanner basis via the use of macro directives.

Macro Directives   Due to the implementation of scanners in C++, a unique prefix is required for each scanner to avoid symbol conflicts when several scanners reside at the same time in the Mozart system. The following macro directive allows this prefix to be changed (the default `zy` is all right if only a single scanner is used at any time):

> `\gumpscannerprefix` ⟨atom⟩

Switches   Figure 2.3 summarizes some compiler switches that control the Gump Scanner Generator.

Figure 2.3 Compiler switches for the Gump Scanner Generator.

Switch	Effect
gumpscannerbestfit	Use best-fit instead of first-fit disambiguating
gumpscannercaseless	Generate a case-insensitive scanner
gumpscannernowarn	Suppress warnings from flex

2.2.3 The Mixin Class `GumpScanner.'class'`

The module `GumpScanner` defines the runtime support needed by Gump-generated scanners. All operations and data are encapsulated in the mixin class `GumpScanner.'class'` that scanners have to inherit from in order to be executable.

Abstract Members The mixin class expects the following features and methods to be defined by derivate classes. (It is a good idea not to define any class members whose name begins with **lex**... since these may be used for internals of the Scanner Generator.)

feat lexer

This feature must contain the scanner-specific loaded foreign functions, which includes the generated scanner tables.

meth lexExecuteAction($+$I)

This method is called each time a regular expression is matched. Regular expressions are assigned unique integers; I indicates which rule's associated action is to be run.

Provided Members The `GumpScanner.'class'` class defines some user functionality that is to be used either by users of the generated scanner or by the semantic actions in the scanner itself.

meth init()

This initializes the internal structures of the `GumpScanner.'class'`. This must be called before any other method of this class.

meth setMode($+$I)

The operation mode of the scanner is set to the lexical mode I. Lexical modes are represented internally as integers. Since modes are identified by variables, the class generation phase wraps a **local** ... **end** around the class equating the mode variables to the assigned unique integers.

meth currentMode($?$I)

This returns the integer I identifying the lexical mode the scanner currently operates in.

meth getAtom($?$A)

This method is used to access the lexeme last matched. It is returned as an atom in the variable A. Note that if the lexeme contains a NUL character (ISO 0) then only the text up to the first NUL but excluding it is returned.

meth getString($?$S)

This method returns the lexeme as a string in the variable S. The restrictions concerning `getAtom` do not apply for `getString`.

meth getLength($?$I)

This method returns the length of the lexeme (number of characters matched).

meth putToken(+X Y**)**

>   This method may be used to append a token with token class X and value Y to the
>   token stream. (Actually, the token class may be an arbitrary Oz value, but atoms and
>   the integers between 0 and 255 are the only representations understood by Gump-
>   generated parsers.)

meth putToken1(+X**)**

>   This method may be used to append a token with token class X and value `unit` to the
>   token stream.

meth getToken(?X Y**)**

>   The next token is removed from the token stream and returned. The token class is
>   returned in X and its value in Y.

meth input(?C**)**

>   The next (unmatched) character is removed from the character stream and returned
>   in C.

meth scanFile(+V**)**

>   This method causes the currently scanned buffer (if any) to be pushed on a stack of
>   active buffers. A new buffer is created from the file with name V and scanned. If the
>   file does not exist, the error exception `gump(fileNotFound V)` with the filename in V
>   is raised; the default treatment is the invocation of a custom error printer.

meth scanVirtualString(+V**)**

>   Like `scanFile`, but scans a virtual string V. If V contains NUL characters (ISO 0) then
>   the virtual string is only scanned up to and excluding the first NUL character.

meth setInteractive(+B**)**

>   Each buffer may be either interactive or non-interactive. An interactive buffer only
>   reads as many characters as are needed to be considered to decide about a match; a
>   non-interactive buffer may read ahead. This method allows the topmost buffer on the
>   stack to be set to interactive (if B is `true`) or non-interactive (if B is `false`). New
>   buffers are always created as non-interactive buffers.

meth getInteractive(?B**)**

>   Whether the topmost buffer on the buffer stack is interactive is returned.

meth setBOL(+B**)**

>   The beginning-of-line (BOL) flag indicates whether the beginning-of-line regular ex-
>   pression ˆ will currently match the input. This flag is true at the beginning of a buffer or
>   after a newline has been scanned. The flag's value may be set at will with this method.

meth getBOL(?B**)**

>   Returns the current state of the beginning-of-line flag. See the `setBOL` method.

meth closeBuffer()

>   Closes the topmost buffer on the buffer stack and resumes scanning from the buffer
>   on the new stack top (if any). If the buffer stack is or becomes empty through this
>   operation, only tokens with class `'EOF'` and value `unit` are returned subsequently
>   (until a new buffer is created).

meth close()

Closes all buffers on the buffer stack. Before calling any other methods, you should call `init()` again.

The Gump Parser Generator

This chapter describes the Gump Parser Generator. As for the Gump Scanner Generator described in the last chapter, its input consists of an Oz source with embedded parser specifications and the output are Oz class definitions.

Definitions   A parser is a program that performs syntax analysis. This means that a stream of tokens is analyzed and a (unique) tree structure on the tokens in this stream is computed. The token classes are called terminal symbols; additionally, new nonterminal symbols are introduced in the specification. For each nonterminal, a set of rules is given which indicates sequences of symbols that may be replaced by this nonterminal. The token sequence is read from left to right and subsequences of symbols are replaced by nonterminal symbols according to the rules (which is called a reduction). Either the result is a special nonterminal, a start symbol, or the input is erroneous and rejected. A result is constructed during the parse by executing user-specified semantic actions each reduction.

This chapter first describes the basic concepts of the Gump Parser Generator by example in Section 3.1. Section 3.2 presents the more advanced concepts and a detailed definition of the specification language.

3.1   Example

This section presents the parser for the functional language Lambda for which a scanner was specified in the last chapter.

3.1.1   Writing a Parser Specification

Figure 3.1 shows the parser specification which will serve as an example to demonstrate the basic concepts of the Gump Parser Generator. This example will be examined in detail in the following.

Class Descriptors   Again, a Gump specification resembles a class definition introduced by a special keyword, `parser`, and augmented by additional declarations. The usual class descriptors `from` and `meth` are also used in this specification in lines 2 to 8. The switches `gumpparseroutputsimplified` and `gumpparserverbose` simply

Figure 3.1 The `LambdaParser` parser specification.

```
\switch +gumpparseroutputsimplified +gumpparserverbose

declare
parser LambdaParser from GumpParser.'class'
   meth error(VS) Scanner in
      GumpParser.'class', getScanner(?Scanner)
      {System.showInfo 'line '#{Scanner getLineNumber($)}#': '#VS}
   end

   token
      'define' ';' '=' ')'
      '.': leftAssoc(1)
      'APPLY': leftAssoc(2)
      'lambda': leftAssoc(2)
      '(': leftAssoc(2)
      'id': leftAssoc(2)
      'int': leftAssoc(2)

   syn program(?Definitions ?Terms)
      !Definitions={ Definition($) }*
      !Terms={ Term($) // ';' }+
   end
   syn Definition($)
      'define' 'id'(I) '=' Term(T) ';' => definition(I T)
   end
   syn Term($)
      'lambda' 'id'(I) '.' Term(T)     => lambda(I T)
   [] Term(T1) Term(T2) prec('APPLY')  => apply(T1 T2)
   [] '(' Term(T) ')'                  => T
   [] 'id'(I) Line(L)                  => id(I L)
   [] 'int'(I)                         => int(I)
   end
   syn Line($)
      skip => {GumpParser.'class', getScanner($) getLineNumber($)}
   end
end
```

cause additional information to be output at parser generation time; we will see this in the next section.

The `error` method will be called upon detection of parse errors. Its parameter is a virtual string describing the error. We redefine this method (which has a default implementation in the super class `GumpParser.'class'`) since we want to provide the user with the line number information we maintain in the scanner.

Token Declarations   In line 10 begin the token declarations. All token classes (which must be atoms) that the scanner can produce are listed after the **token** keyword. Additionally, some tokens are assigned an associativity (here: `leftAssoc`) and a precedence value (a nonzero positive integer) after a colon. These are used to resolve ambiguities in the syntax rules. The reason for the assignments in our example are explained below. (You may notice that one of the listed tokens cannot be produced by the scanner, the `'APPLY'` token. This is called a pseudo-token and is solely defined for its associativity and precedence information.)

Syntax Rules   Line 19 marks the start of the syntax rules themselves. For each nonterminal, a syntax rule (introduced by the keyword **syn**) must be given. Nonterminals may be named by atoms or variables.

Start Symbols   An atom means that this nonterminal is a start symbol. Several start symbols may be defined – the one to reduce to is selected when a concrete parse is initiated.

Formal Parameter Lists   Following the nonterminal is its parameter list, consisting of zero or more variables in parentheses. The start symbol `program` has two parameters: a list of definitions and a list of terms. These are both output parameters, as is indicated by the commentary `?`.

EBNF Phrases   The body of each syntax rule is an EBNF phrase (EBNF is an abbreviation of Extended Backus-Naur-Formalism). As in Oz, we distinguish between statements and expressions: Some EBNF phrases carry values and may thus only stand at expression position, others don't and must be used at statement position.

The basic building blocks of EBNF expressions are grammar symbol applications, denoted by the name of a terminal or nonterminal followed by the actual parameter list in parentheses. An example of this is the `Definition($)` in line 20, which is an application of the nonterminal `Definition` with a single actual parameter. Since this is the nesting marker, the application is an expression (as opposed to a statement) with the value of the corresponding actual parameter as its value. This application is written inside the repetition symbols { ... }*****, which means that the application is to be repeated 0 to *n* times. The repetition construct builds a list of its argument's values at each iteration, since it is used in expression position. This list is assigned to the formal parameter `Definitions`.

The next line, line 21, is similar: Here, a nonempty list (note the **+**) of `Term`s is expected, seperated by semicolons. The values computed by each `Term` are collected in a list, which is assigned to the formal parameter `Terms`.

Local Variables   The next syntax rule introduces a new feature: local variables. All variables in pattern position in syntax rules are implicitly declared local. EBNF pattern positions are the left side of an assignment (such as in line 20) and the actual parameters of grammar symbol applications. If in any of these places a single non-escaped variable (i.e., written without `!`) is used, it is implicitly declared local to the EBNF construct it is used in. Such is the case for the variables `I` and `T` in line 24. The formal parameter variables assigned to in lines 20 and 21 had to be escaped to avoid their implicit (re-)declaration.

The syntax rule for `Definition` in line 23 has a single parameter. Since this is the nesting marker, an EBNF expression is expected as body of this rule. The value of a sequence of EBNF expressions is the value of the last expression (as in Oz, where the value of a sequential composition is the value of the composition's second argument).

Semantic Actions   The last EBNF expression in line 23 is the semantic action, introduced by the arrow `=>`. This action constructs an abstract syntax tree node (represented as a tuple).

Alternatives   Lines 26 to 32 show the rule for `Term`. This rule has several alternatives, separated by the choice operator `[]`. These alternatives also imply the need for the given token precedences and associativities mentioned above: Not all inputs have a unique parse tree. If, for example, we wrote

    lambda x.y z

this could be parsed as either

    (**lambda** x.y) z

or

    lambda x.(y z)

We want to enforce the second meaning (that is, the application has a higher precedence than the abstraction); furthermore, the application should be left-associative (i.e., `x y z` means `(x y) z`).

Resolving Conflicts   This is why the pseudo-token `'APPLY'` was introduced. Each alternative may also have, like the tokens, a precedence and an associativity. If the alternative contains a terminal, than the values of the last terminal are used. Alternatively, a special precedence token may be specified via `prec(`*terminal*`)`; then the values of this are used instead. Thus, the application `Term Term` is left-associative. Higher precedence values mean tighter binding of operators. Thus, the application (token `'APPLY'` of precedence 2) has precedence over the abstraction (token `'.'` of precedence 1).

However, one anomaly remains because the application has no (visible) operator – to resolve conflicts, the precedence/associativity values of the lookahead token are compared to the values of the (potentially) applicable rules. So if the lookahead is one of the tokens with which a `Term` can begin, it is in fact an application we have to parse. This is why all these tokens are assigned the same precedence as the application. (For a more detailed description of how operator precedence information is used to resolve conflicts, consult the bison manual [2].)

Epsilon Productions   The last nonterminal, `Line` in line 33, is actually only introduced for the semantic value it computes. The empty sequence of grammar symbols is denoted by **skip**.

3.1.2   Invoking Gump

Parser specifications are processed in the same way scanner specifications are. First we prepare the Gump Parser Generator by feeding:

```
\switch +gump
```

Then the file to translate is simply fed into the compiler. Suppose you saved the example specification in the file `LambdaParser.ozg`; feed:

```
\insert LambdaParser.ozg
```

The extension `.ozg` indicating, as before, an Oz file with embedded Gump specifications.

Output Files   Two files are generated from the **parser** definition: `LambdaParser.simplified` contains a simplified version of the syntax rules where the EBNF constructs have been expanded to equivalent BNF forms (because the `gumpparseroutputsimplified` switch was set), whereas the file `LambdaParser.output` contains the output from the bison parse table generator (because the `gumpparserverbose` switch was set). These names are generated from the parser specification's name.

3.1.3   Using the Generated Parser

Figure 3.2 shows an example Oz program that uses both the generated scanner from the last chapter and the generated parser from above.

Initialization   First, the scanner and parser classes are loaded. After instantiating and initializing the scanner, a parser object is created. This needs as initializer a single parameter, a scanner. This is, technically speaking, a unary procedure that understands the messages `putToken` and `getToken` described in Section 3.2.3.

Initiating a Parse   The most interesting message sent to the parser is the `parse` message. The first argument has to be a tuple. The label specifies the start symbol to use, the features correspond to the actual parameters of the start symbol. In this case, the actual parameter variables `Definitions` and `Terms` are bound to lists of definitions and terms, respectively. The second argument to the `parse` message is the result status. This is either unified with **true** if parsing was successful or with **false** otherwise.

Figure 3.2 A program making use of the generated parser.

```
\switch +gump
\insert gump/examples/LambdaScanner.ozg
\insert gump/examples/LambdaParser.ozg

local
   MyScanner = {New LambdaScanner init()}
   MyParser = {New LambdaParser init(MyScanner)}
   Definitions Terms Status
in
   {MyScanner scanFile('Lambda.in')}
   {MyParser parse(program(?Definitions ?Terms) ?Status)}
   {MyScanner close()}
   if Status then
      {Browse Definitions}
      {Browse Terms}
      {System.showInfo 'accepted'}
   else
      {System.showInfo 'rejected'}
   end
end
```

3.2   Reference

This section is the reference manual for the Gump Parser Generator. It is divided into three parts: First, the syntax of the Gump parser specification language is given in Section 3.2.1. Then, the options to parser generation supported by the Gump Parser Generator are detailed in Section 3.2.2. Finally, the runtime support for generated parsers, the mixin class `GumpParser.'class'`, is presented in Section 3.2.3.

3.2.1   Syntax of the Parser Specification Language

The meta-notation used for describing the syntax of the specification language is explained in Appendix A. (Note: This is *not* the language used in Gump to specify parsers. This is intentional.)

Gump specifications are allowed anywhere as a statement.

$$\langle\text{statement}\rangle \quad +\!= \quad \langle\text{parser specification}\rangle$$

A parser specification is introduced by the keyword **parser**, followed by the usual components of an Oz class. After these come additional parser-specific descriptors. Parser specifications must be named by variables, since the names of these variables will be used to generate auxiliary file names during parser generation.

⟨parser specification⟩   ::=   **parser** ⟨variable⟩
                                 { ⟨class descriptor⟩ }
                                 { ⟨method⟩ }
                                 [⟨token clause⟩]
                                 { ⟨parser descriptor⟩ }+
                                 end

3.2.1.1 Token Declarations

The first extra parser descriptor is the **token** clause. This defines the names of the terminals used in the specification as well as (optionally) their associativity and precedence. Several tokens are predefined: Atoms of length 1 are always considered to be tokens. Furthermore, token `'error'` stands for an erroneous token (sequence) and is used for error recovery, and token `'EOF'` signalizes the end of input and is always expected before reduction to the start symbol can take place.

⟨token clause⟩   ::=   **token** { ⟨token declaration⟩ }+

⟨token declaration⟩   ::=   ⟨atom⟩ [':' ⟨expression⟩]

The optional expression following the colon in a token declaration must be a tuple with arity 1 and one of the labels `leftAssoc`, `rightAssoc` or `nonAssoc`, depending on the desired associativity. The feature must always be a nonzero positive integer. Only the relative values matter; they are used to derive an ordering on the tokens. Larger values imply a greater binding strength of the operator. For the algorithm used to resolve conflicts using operator precedence information, refer to the bison manual [2].

3.2.1.2 Syntax Rules

Syntax rules are parser descriptors. They are composed of a head and a body. The head specifies the name of the defined nonterminal, where atoms are considered start symbols, as well as the formal parameters of the nonterminal. Only one syntax rule per nonterminal name is allowed.

⟨parser descriptor⟩   ::=   ⟨syn clause⟩

⟨syn clause⟩   ::=   **syn** ⟨syn head⟩ ⟨syn alt⟩ **end**

⟨syn head⟩   ::=   ⟨atom⟩
                 | ⟨atom label⟩ ⟨syn formals⟩
                 | ⟨variable⟩
                 | ⟨variable label⟩ ⟨syn formals⟩

⟨syn formals⟩   ::=   '(' { ⟨syn formal⟩ } ')'

The body of a syntax rule is an EBNF phrase. It is distinguished between EBNF statements and EBNF expressions: EBNF expressions carry an additional value. In the following, it is always specified where EBNF statements or expressions are expected and which constructs yield a value.

Formal parameters are denoted by variables. At most one parameter may be the nesting marker; in this case the body of the syntax rule must be an EBNF expression. Its value is unified with the corresponding actual parameter upon application of the nonterminal.

$$\langle\text{syn formal}\rangle \quad ::= \quad \langle\text{variable}\rangle$$
$$| \quad \text{'\_'}$$
$$| \quad \text{'\$'}$$

An alternation specifies several sequences (called alternatives), separated by the choice operator `[]`. Either all sequences must be EBNF expressions or all sequences must be EBNF statements. If all alternatives are expressions, the alternation is an expression and yields at runtime the value of the selected sequence at runtime.

$$\langle\text{syn alt}\rangle \quad ::= \quad \langle\text{syn seq}\rangle \; \{ \; \text{'[]'} \; \langle\text{syn seq}\rangle \; \}$$

At the beginning of an sequence, local variables may be declared. These are visible only inside the sequence. The sequence itself is composed of $n \geq 0$ EBNF factors, optionally followed by a semantic action. If an EBNF expression is expected at the place the sequence stands, then a semantic action must either be an expression or be omitted. In the latter case, the last EBNF phrase must be an EBNF expression, the value of the sequence then is the value of this EBNF expression. All other EBNF factors must be statements. If $n = 0$, then the sequence may be written as `skip`.

$$\langle\text{syn seq}\rangle \quad ::= \quad [\; \{ \; \langle\text{variable}\rangle \; \}+ \; \textbf{in} \;] \; \{ \; \langle\text{syn factor}\rangle \; \} \; [\; \langle\text{syn action}\rangle \;]$$
$$| \quad \textbf{skip} \; [\; \langle\text{syn action}\rangle \;]$$

$$\langle\text{syn action}\rangle \quad ::= \quad \text{'=>'} \; (\; \langle\text{in statement}\rangle \; | \; \langle\text{in expression}\rangle \;)$$

An EBNF factor is either an application or an assignment. An application is denoted by the name of either a terminal or a nonterminal, optionally followed by the actual parameters in parentheses. Terminals may either have a single (variable) parameter or no parameter at all; if a parameter is specified then it is unified with the actual token value at runtime. In the application of a nonterminal, the number of actual parameters must correspond to the number of formal parameters in the nonterminal's definition. Non-escaped variables as actual parameters are implicitly declared local to the innermost sequence that contains the application. At most one actual parameter may be the nesting marker. In this case, the application is an expression yielding the value of the corresponding actual parameter; else it is a statement.

$$\langle\text{syn factor}\rangle \quad ::= \quad \langle\text{syn application}\rangle$$
$$| \quad \langle\text{syn assignment}\rangle$$

$$\langle\text{syn application}\rangle \quad ::= \quad \langle\text{atom}\rangle$$
$$| \quad \langle\text{atom label}\rangle \; \langle\text{syn actuals}\rangle$$
$$| \quad \langle\text{variable}\rangle$$
$$| \quad \langle\text{variable label}\rangle \; \langle\text{syn actuals}\rangle$$

$$\langle\text{syn actuals}\rangle \quad ::= \quad \text{`(' } \{ \langle\text{expression}\rangle \} \text{ `)'}$$

Two grammar symbols are predefined which receive a special treatment:

'prec'(A)

By inserting an application of `prec` into a sequence, the latter is assigned an associativity and a precedence. These are taken from the token A. Sequences that contain no application of `prec` inherit the values of the last token used in the sequence if there is one, and have no associated associativity and precedence otherwise.

'error'

The application of the predefined terminal `'error'` defines a restart point for error recovery. Consult the bison manual [2] for additional information.

An assignment equates a variable with the value of an EBNF expression. Unless the variable is escaped, it is implicitly declared local to the sequence the assignment appears in, else it must have been declared local within the current syntax rule (or be a formal parameter). An assignment is always a statement.

$$\langle\text{syn assignment}\rangle \quad ::= \quad [\text{ `!' } ] \langle\text{variable}\rangle \text{ `=' } \langle\text{syn factor}\rangle$$

3.2.1.3 Definition of Production Templates

This section and the next augment the syntax rules defined above by the concept of production templates. These provide for, e.g., the repetition constructs used in the example in Section 3.1.

The definition of a production template is another parser descriptor. Production templates are local to the parser specification they are defined in, and may be used only textually after their definition. (This is to avoid cyclic production template expansions.) Production templates may be redefined.

$$\langle\text{parser descriptor}\rangle \quad += \quad \langle\text{prod clause}\rangle$$

A production template definition consists of a head and a body. The body specifies the EBNF phrase the production template is to be replaced with when instantiated. The body may introduce optional local syntax rules which are always newly created when instantiated. These must be denoted by variables.

$$\langle\text{prod clause}\rangle \quad ::= \quad \textbf{prod} \ \langle\text{prod head}\rangle$$
$$[\ \langle\text{local rules}\rangle \ \textbf{in} \ ] \ \langle\text{syn alt}\rangle$$
$$\textbf{end}$$

$$\langle\text{local rules}\rangle \quad ::= \quad \{ \ \langle\text{syn clause}\rangle \ \}+$$

The head of a production template provides – aside from the list of its formal parameters – the unique identification of the production template. This is composed of the following parts:

1. whether the production template is an expression or a statement when it is instantiated (expressions being denoted by `v=...` or `$=...;` in the head);

2. the optional identification name of the template, written before a colon;

3. the used parentheses, brackets or braces, if any;

4. the number of arguments, all being separated by `//`; and

5. the used postfix operator, if any.

For example, you could define the commonly used notation `[ X ]` as an EBNF option, or use `{ X // Y }+` for a separated list with at least one element. This construct could yield a value, such as a list of the Oz values produced by the expression *X*, which would be denoted by the production template *Z*=`{ X // Y }+`. (Compare this to the template's instantiation in Figure 3.1 in line 21.)

⟨prod head⟩ ::= ⟨template definition⟩
    | ⟨variable⟩ '`=`' ⟨template definition⟩
    | '`$`' '`=`' ⟨template definition⟩

⟨template definition⟩ ::= ⟨prod formal list⟩
    | ⟨atom⟩ '`:`' ⟨prod formal list⟩

⟨prod formal list⟩ ::= '`(`' ⟨prod formals⟩ '`)`' [⟨prod postfix⟩]
    | '`[`' ⟨prod formals⟩ '`]`' [⟨prod postfix⟩]
    | '`{`' ⟨prod formals⟩ '`}`' [⟨prod postfix⟩]
    | ⟨variable⟩ ⟨prod postfix⟩

⟨prod formals⟩ ::= ⟨variable⟩ { '`//`' ⟨variable⟩ }

⟨prod postfix⟩ ::= '`+`' | '`-`' | '`*`' | '`/`'

3.2.1.4 Expansion of Production Templates

Production templates may be instantiated as EBNF factors.

⟨syn factor⟩ += ⟨template instantiation⟩

The instantiation of a production template is very similar to its definition, since it must specify the same unique identification. The difference is that instead of the formal parameter variables actual EBNF phrases are allowed.

⟨template instantiation⟩ ::= ⟨prod actual list⟩
    | ⟨atom⟩ '`:`' ⟨prod actual list⟩

⟨prod actual list⟩ ::= '`(`' ⟨prod actuals⟩ '`)`' [⟨prod postfix⟩]
    | '`[`' ⟨prod actuals⟩ '`]`' [⟨prod postfix⟩]
    | '`{`' ⟨prod actuals⟩ '`}`' [⟨prod postfix⟩]
    | ⟨syn application⟩ ⟨prod postfix⟩

$$\langle\text{prod actuals}\rangle \quad ::= \quad \langle\text{syn alt}\rangle \ \{ \ \text{`//'} \ \langle\text{syn alt}\rangle \ \}$$

When a production template is expanded, name clashes must be avoided. This is why the expansion proceeds in several steps:

- The local variables of the template are uniquely renamed, both in the body's EBNF phrase as well as in the local rules.

- The local rules are uniquely renamed to avoid confusion with other rules in the parser specification.

- The actual EBNF phrases are substituted for the parameter variables of the production template. The formal parameter variables may only occur as applications of grammar symbols and may either be applied with a single actual parameter or none at all. If the parameter is given, then the actual EBNF phrase must be an expression whose value is unified with the application's actual parameter.

- The local rules are quantified over the local variables used in actual EBNF phrases of the instantiation by adding these as parameters.

- The local rules are aded to the table of grammar symbols.

- The template instantiation is replaced by the body's EBNF phrase from the production template's definition.

3.2.1.5 Predefined Production Templates

Figure 3.3 shows the predefined production templates. For many operators several equivalent notations exist. All operators also have a form that yields a value: The grouping construct yields the value of its argument, as do options (or `nil` if they are not chosen at runtime); the repetition constructs yield Oz lists of their first argument.

Figure 3.3 Predefined production templates.

Grouping	(A)
Option	[A]
Mandatory Repetition	A+ (A)+ { A }+
Optional Repetition	A* (A)* { A }*
Mandatory Separated Repetition	(A // B)+ (A // B) { A // B }+ { A // B }
Optional Separated Repetition	(A // B)* { A // B }*

3.2.1.6 Assignment of Attribute Types

Due to the underlying LR(1) algorithm used, two different attribute types must be distinguished concerning parameters to nonterminals, namely synthesized and inherited attributes. This is in contrary to Oz, where input and output arguments need not be distinguished due to the concept of logical variables and unification. However, things are simplified by an algorithm determining the attribute types automatically.

Before this algorithm is explained in the following, we need to introduce a definition.

Definition   Let S be an expanded sequence (i.e., template instantiations and assignments have been expanded) with EBNF factors $0, \ldots, n$. Let i be the index of the first EBNF factor (application or semantic action) in which a local Variable V (which is not a formal parameter) of the sequence occurs. Then we say that V is initialized in all EBNF factors with indices j, $j \geq i$, and uninitialized in all others.

The following rules describe how attribute types are derived from their uses in applications of grammar symbols:

- The (optional) parameter of a terminal always is a synthesized attribute (since the scanner always produces the token value).

- Let the ith actual parameter of an application of a grammar symbol B be either an uninitialized local variable V or a nesting marker. Then the ith formal parameter of B is a synthesized attribute. Furthermore, V may not occur in any other actual parameter of the application.

- Let the ith actual parameter of an application of a grammar symbol B be either an initialized local variable V or a complex Oz expression (i.e., neither a variable nor a nesting marker). Then the ith formal parameter of B is an inherited attribute. Furthermore, no uninitialized variable may occur in said actual parameter.

- If a formal parameter of the syntax rule for a nonterminal A is used as actual parameter of an application of a nonterminal B, then the corresponding formal parameters of A and B are attributes of the same type, i.e., either both synthesized or both inherited.

Note that nothing can be concluded from the use of a formal parameter variable in a semantic action, since Oz does not distinguish between access of and assignment to a variable: both are realized by unification.

If contradicting attribute types are derived for any formal parameter variable of a nonterminal, then this is an error. If no attribute type can be derived for a formal parameter variable, then it is realized as a synthesized attribute.

3.2.2 Parameters to Parser Generation

Macro Directives   The following macro directive tells the bison parse table generator to expect a certain number of shift/reduce conflicts:

> `\gumpparserexpect` $\langle$int$\rangle$

Switches   Figure 3.4 summarizes the options that the Gump Parser Generator understands. They may be given as compiler switches before a parser specification.

Figure 3.4 Compiler switches for the Gump Parser Generator.

Switch	Effect
gumpparseroutputsimplified	create the `.simplified` file with the BNF version of the grammar
gumpparserverbose	create the `.output` file with the Bison verbose output

3.2.3 The Mixin Class `GumpParser.'class'`

The mixin class `GumpParser.'class'`, defined in the module `GumpParser`, is required to make Gump parser specifications executable. It requires some features to be present in derived classes; these are automatically inserted by the Gump Parser Generator and contain the generated parse tables. They all begin with `syn...`; thus it is a good idea not to define any such named class components in order to avoid conflicts with Gump internals. Likewise, you should not define any variables beginning with `Syn...`, since such variable names are generated by the tool.

Abstract Members Furthermore, the following method must be defined:

meth synExecuteAction(+I)

This method is invoked each time a reduction takes place. The parameter `I` is the number of the rule reduced.

Provided Members `GumpParser.'class'` defines several attributes and methods that may be called by users of the generated parser or from inside semantic actions:

attr lookaheadSymbol

This contains the token class of the current lookahead symbol.

attr lookaheadValue

This contains the token value of the current lookahead symbol.

feat noLookahead

This is the value `lookaheadSymbol` should be set to if you want to skip a token from inside a semantic action.

meth init(+P)

This initializes the internal structures of the `GumpParser.'class'` and connects it to a scanner `P`. `P` must at least understand the messages `putToken` and `getToken` as described in Section 2.2.3.

meth parse(+T ?B)

This methods initates a parse. The label of tuple `T` denotes the start symbol to use (which must be a declared nonterminal named by an atom); its features correspond to the parameters of the corresponding syntax rule. Values of inherited attributes are extracted from this tuple, values of synthesized attributes are unified with the corresponding features after the parse is finished (successfully). The parameter `B` is unified with **true** if the parse was successful and with **false** otherwise.

meth accept()

>   By calling this method the parse is interrupted and success reported. (Note that the values of synthesized attributes of the start symbol given to `parse` are not influenced by this.)

meth abort()

>   By calling this method the parse is interrupted and failure reported. (Note that the `error` method is not called.)

meth raiseError()

>   This method places the parser in the same state as if a syntax error had been found in the input. Normal error recovery is attempted. The method `error` is not called.

meth errorOK()

>   When a production with a restart point (token `error`) is reduced, this method may be called to tell the parser that the error recovery process is finished and normal parsing may be resumed.

meth clearLookahead()

>   When a production with a restart point (token `error`) is reduced, this method may be called to clear the lookahead token (if, for example, it was used to synchronize to the restart point and is not legal thereafter).

meth error($+$V)

>   This method is always invoked when (during normal parsing) an error in the input is recognized. It is handed a diagnostic message in V. This method may be overridden in derived classes.

meth getScanner($?$P)

>   Returns the scanner object or procedure P currently used as the token source.

The Used Notation

This appendix describes the notation used to specify the syntax of the Gump specification language as an extension of the Oz language. It is an extended Backus-Naur-Formalism built from the following parts:

- terminals and nonterminals are enclosed in angle brackets $<\ldots>$;

- the left side is separated from the right side by either ::= or +=, where += adds productions to an existing nonterminal;

- the vertical bar separates alternatives;

- square brackets denote optional phrases;

- curly braces enclose phrases that may be repeated 0 to n times;

- curly braces with a suffixed + enclose phrases that may be repeated 1 to n times;

- literal strings are typeset in `this way`.

A.1 Elements from the Oz Syntax

Lexical Conventions   Since the tool's specification language is embedded into Oz, the same lexical conventions apply as for Oz. The only additional terminal type is ⟨regex⟩, described in Section 2.2.1.

Terminals   The following named terminals from the Oz syntax are used:

⟨atom⟩   stands for an Oz atom (quoted or not).

⟨variable⟩   denotes an Oz variable (backquoted or not).

⟨atom label⟩   is an Oz atom that is immediately followed by a left parenthesis.

⟨variable label⟩ is an Oz variable that is immediately followed by a left parenthesis.

See *"The Oz Notation"* for the exact definitions.

Nonterminals   Furthermore, the following nonterminals from the Oz syntax are used:

⟨statement⟩   is an Oz statement.

⟨expression⟩   is an Oz expression.

⟨in statement⟩ is an Oz statement with optional preceding variable declaration.

⟨in expression⟩ is an Oz expression with optional preceding variable declaration.

⟨class descriptor⟩ stands for a class descriptor, i.e., one of **from**, **prop**, **attr** or **feat**.

⟨method⟩   stands for a **meth** ... **end** definition.

See *"The Oz Notation"* for the exact definitions.

Bibliography

[1] Lyman Frank Baum. *The Wonderful Wizard of Oz*. G. M. Hill, 1900.

[2] Charles Donelly and Richard Stallman. *Bison: The YACC-Compatible Parser Generator (Reference Manual)*. Free Software Foundation, Version 1.25 edition, November 1995. On-Line Info File.

[3] Leif Kornstaedt. Definition und Implementierung eines Front-End-Generators für Oz. Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern und Programming Systems Lab, Universität des Saarlandes, September 1996.

[4] Vern Paxson. *flex – Fast Lexical Analyzer Generator*, Version 2.5.2 edition, April 1995. Unix On-Line Manual Page.

Index