# Zebu: A Tool for Specifying Reversible LALR(1) Parsers

Joachim Laubsch (laubsch@hplabs.hp.com)

April 19, 1994

**Abstract**

Zebu[1] is part of a set of tools for the translation of formal languages. Zebu contains a LALR(1) parser generator like Yacc does. Aside from generating a parser, Zebu will also generate the inverse of a parser (unparser). In contrast to Yacc, the semantics is not given in terms of "routines" but declaratively in terms of typed feature structures.

The ability to declaratively define a reversible grammar, together with a rewrite-rule mechanism (Zebu-RR) for transforming abstract syntax trees constitute the basic tools for specifying translators for formal languages.

**Keywords** Formal language, LALR-grammar, parsing, translation, generation, interoperability, LEX, YACC.

## 1 Introduction

Our goal is to develop an environment for the design, analysis and manipulation of formal languages, such as programming languages, markup languages, data interchange formats or knowledge representation languages (such as the translation to and from KIF) [4]. Being able to design, analyze, and manipulate formal languages is crucial for achieving software interoperability [3], automatic code analysis, indexing, and retrieval for potential reuse. Zebu has been applied to writing translators for formal languages [6]. The main idea of this work is that a module $m$ communicates by sending or receiving messages in some language $L(m)$, and that for various reasons different modules use different languages.

[1] "**zebu***n.*, *pl.* **-bus**, **-bu**: 1. an oxlike domestic animal (*Bos indicus*) native to Asia and parts of Africa: it has a large hump over the shoulders, short, curving horns, pendulous ears, and a large dewlap and is resistant to heat and insect-born deseases." [Webster's New World Dictionary.] A zebu should not be confused with a yacc or a gnu although it bears similarity to each of them.

For communication to be successful, translators have to be used. Zebu provides tools to define translators at a high level of abstraction[2].

McCarthy introduced the notion of "abstract" and "concrete" syntax. The concrete syntax describes the surface form of a linguistic expression, while the abstract syntax describes a (composite) object. E.g. "1+a" is the surface string rendered by a particular concrete syntax for an object described by an abstract syntax: an addition operation with two operands, the first being the numeral "1", and the second being the variable named "a".

Manipulation of linguistic expressions is much easier to express in the abstract syntax than in the concrete syntax.

If we were to design an algorithm for simplifying expressions of some language — say "arithmetic" — we would use as the front end the "arithmetic-parser" to translate into abstract syntax, then express the simplification rules in terms of tree transformation rules that operate on the abstract syntax, and finally add as the back-end the "arithmetic-unparser".

More generallly, if we were to design an algorithm for translating from language A to language B, we would define reversible grammars for languages A and B, and sets of rewrite rules to transform the abstract syntax trees from the domain of language A to the domain of language B. The front end would be the "A-parser" and the back-end the "B-unparser"

The work described in this report ows a lot to the pioneering research at Kestrel [9] that resulted in the Refine[3] program transformation system [8]. The basic ideas underlying Zebu are already present in Refine. Zebu is a much more compact sytem than Refine (it runs on a MacIntosh in MacIntosh Common Lisp), and the semantics is expressed in typed feature structures. Zebu also offers the possibility of defining a meta-grammar. Zebu lacks Refine's ability to declaratively specify transformations using a pattern language. [4]

The LALR(1) parsing table generated by Zebu follows algorithms described in [1] or [2]. The current implementation was developed from the Scheme program developed by William Wells and is written in Common Lisp.

The next section will explain how a grammar can be defined, and how semantics can be associated with a grammar rule. Section 3 describes the definition of the semantic domain. With this capability it is possible to state declaratively what the abstract syntax should look like. Section 4 describes a simpler grammar notation that is very close to ordinary BNF. Section 5 summarizes the functional interface of Zebu and explains how a parser can be customized. The last chapter describes, how lexical analysis can be defined using regular expressions and parameterization.

[2] The rewrite-rule mechanism (Zebu-RR) is implemented, and will be described in a future report.

[3] Refine is a trademark of Reasoning Systems, Palo Alto.

[4] Zebu can be obtained via anonymous ftp from hplabsz.hpl.hp.com (15.0.152.161) as a compressed tar file: pub/zebu-???.tar.gz. It contains several example grammar definitions.

# 2 The Representations of Grammars in Files

## 2.1 Grammar notation

We first describe the null-grammar, which is a powerfull but verbose way to specify a grammar. Only a parser and optionally a domain will be generated but an unparser (printer) will not. If this is desired, you must use the notation of the meta-grammar "zebu-mg" which is described in section 4.

Non-terminals are represented by symbols, terminals (also referred to as keywords) by strings. There are the following open classes of non-terminals[5]:

`identifier` ::= ⟨lisp symbol⟩
`number` ::= ⟨integer⟩
`keyword` ::= ⟨string⟩

where

⟨integer⟩ ::= ⟨digit⟩*
⟨string⟩ ::= " ⟨character⟩* "

A ⟨lisp symbol⟩ may be qualified by a package name, e.g. `zb:cons-1-3` is a valid identifier. In case packages should be disallowed during lexical analysis, the variable `*disallow-packages*` should be bound to *true*. (It defaults to *false*). The alphabetic case of a keyword is not significant if the variable `zb:*case-sensitive*` is *false* (the default) when the grammar is loaded.

If alphabetic case of identifiers is to be preserved, `zb:*preserve-case*` should be set to *true*. Other categories can be defined as regular expressions (see 6.2).

### 2.1.1 Grammar Rules

**Grammar Rule Syntax** A grammar file consists of a header (the "options list", see section 3.1) followed by one or more domain definitions or grammar rules. The non-terminal defined by the first grammar rule is also the *start-symbol* of the grammar. A parser will accept exactly the strings that rewrite to the *start-symbol*.

This example shows how a BNF-like rule can be encoded as a Zebu grammar rule (using the null-grammar):

A ::= B | C ⟨number⟩ | "foo" A | "c" | ⟨the-empty-string⟩

```
(defrule A
         := B                          ; (1)
         := (C NUMBER)                 ; (2)
         := ("foo" A)                  ; (3)
         := "c"                        ; (4)
```

[5] The Kleene * indicates 0 or more occurences of the preceding constituent

```
         := ()                              ; (5)
         )
```

This rule describes 5 productions, all deriving the non-terminal `A`. Each of the productions has the left-hand side `A`. The right-hand side of (1) consists of just one constituent, the non-terminal `B`. (2) has a right-hand of length 2, and its second constituent is the non-terminal `NUMBER` (which rewrites to any integer, real or rational). (3) is a recursive production. (4) contains just the terminal (or keyword) `"c"`. (5) derives the empty string.

None of these productions has a semantic action attached. By default, the semantic action is the `identity` function if the right-hand side of the rule consists of a single constituent and the `identity*` function otherwise. (`identity*` is defined as the function that returns all its arguments as a list.)

**Grammar Rule Semantic Actions** If we want to attach other than these default semantic actions, we have to use a `:build` clause after a production.

The build clause has the syntax:

⟨build clause⟩ ::= `:build` (⟨lisp function⟩ ⟨argument list⟩)
⟨build clause⟩ ::= `:build` ⟨atomic lisp form⟩
⟨build clause⟩ ::= `:build` (`:form` ⟨lisp form⟩)
⟨build clause⟩ ::= `:build` (`:type` ⟨struct-type⟩
`:map` ((⟨non-terminal⟩ . ⟨Slot⟩) ...))

The first case

`:build` (⟨lisp function⟩ ⟨argument list⟩)

is like a function call. It may contain free variable occurrences. These will be bound to the non-terminal constituents of the same name occuring in the right-hand side of the production at the time of applying the semantic action.

In the second case

`:build` ⟨atomic lisp form⟩

the ⟨atomic lisp form⟩ must be a function. It will be applied to the constituents of the right-hand side. This function should have the same number of arguments as the right-hand side of the corresponding production has constituents.

Since it happens often, that only some of the constituents of the right-hand side are selected, or combined, a few useful semantic actions have been predefined in Zebu.[6]

An example for such a predefined action is the function `cons-2-3` which takes 3 arguments and returns a *cons* of its second and third argument.

The third form of the `:build` clause is just a long way to write the first form, i.e.

[6] They are described in the file "zebu-actions.l".

`:build` (⟨lisp function⟩ ⟨argument list⟩)

is the same as

`:build` (`:form` (⟨lisp function⟩ ⟨argument list⟩))

Similarly,

`:build` (progn ⟨atomic lisp form⟩)

is the same as

`:build` (`:form` ⟨atomic lisp form⟩)

The last `:build` clause is more interesting:

```
:build (:type ⟨struct-type⟩
        :map  ((⟨Nonterminal⟩ . ⟨Slot⟩) ...))
```

where ⟨struct-type⟩ is a symbol that must be the name of a structure type. Instead of having to write the semantic action as a constructing form, we just have to specify the type and the mapping of non-terminals to slots, as in the following example[7]:

```
(defrule Boolean-Expr
         := (Formula.1 "and" Formula.2)
         :build (:type Boolean-And
                 :map ((Formula.1 .  :-rand1)
                       (Formula.2 .  :-rand2)))

         := (Formula.1 "or" Formula.2)
         :build (:type Boolean-Or
                 :map ((Formula.1 .  :-rand1)
                       (Formula.2 .  :-rand2)))
         )
```

The map indicates that the slot `-rand1` is to be filled by the value of the non-terminal `Formula.1`, etc.

This example also makes use of the `".n"` notation: If on the right-hand side of a production a nonterminal occurs repeatedly, we distinguish it by appending `"."` and a digit.

The function `print-actions` applied to the name of a grammar may be used to find out what the generated code for the semantic actions looks like, e.g. after compiling the sample grammar `‘‘pc1.zb’’`:

[7](taken from the grammar named “pc1”; see the file “pc1.zb” in the test directory)

```
(print-actions "pc1")

...
Rule: BOOLEAN-EXPR
(LAMBDA (FORMULA.1 DUMMY FORMULA.2)
        (DECLARE (IGNORE DUMMY))
        (MAKE-BOOLEAN-AND :-RAND1 FORMULA.1 :-RAND2 FORMULA.2))
(LAMBDA (FORMULA.1 DUMMY FORMULA.2)
        (DECLARE (IGNORE DUMMY))
        (MAKE-BOOLEAN-OR :-RAND1 FORMULA.1 :-RAND2 FORMULA.2))
...
```

These semantic actions have been generated from the `:build` clauses of the above rule for `Boolean-Expr`.

# 3 The Options List of the Grammar File

## 3.1 Keyword Arguments to Grammar Construction

Some global information to control grammar compilation, lexical analysis, and the generation of semantic actions will be declared in the beginning of a grammar file[8]. We chose to let the first list of a grammar file take the form of a keyword argument list to be given to the function `zb:make-grammar`. The following keywords are valid:

| | |
|---|---|
| `:name` | a string, the name of the grammar |
| `:package` | a string, the name of the package where the non-terminal symbols and the function symbols used in semantic actions reside |
| `:identifier-start-chars` | a string. During lexical analysis any character in this string can start an `identifier` non-terminal.<br>The default is `*identifier-start-chars*`. |
| `:identifier-continue-chars` | a string. During lexical analysis any character in this string can continue an `identifier` (i.e. characters not in this string terminate `identifier`). The default is `*identifier-continue-chars*`. |
| `:string-delimiter` | a character, the character that delimits a |

[8]A grammar file has the default type ".zb".

| | |
|---|---|
| | string to be represented as a Common Lisp string. (default `#\"`) |
| `:symbol-delimiter` | a character, the character that delimits a string to be represented as a Common Lisp symbol. (default `#\'`) |
| `:domain` | a list, representing the hierarchy of the domain. See section 3.2 below. |
| `:domain-file` | a string naming the file where the generated CL program that implements the domain will be stored. Definitions of functions for semantic actions and regular expression categories are kept here as well. This string defaults to the concatenation of the grammar's :name and "-domain". |
| `:grammar` | a string, by default: `"null-grammar"`, naming the grammar to be used to parse the grammar defined in this file. If the grammar `"zebu-mg"` is used, an unparser will also be generated. |
| `:lex-cats` | an association list of terminal category names and regular expressions (see section 6.2) |
| `:white-space` | a list of characters each of which will be ignored before a token. (default `(#\Space #\Newline #\Tab)`) |
| `:case-sensitive` | true if the case of keywords is significant, false otherwise (default *false).* |

## 3.2 Defining a Domain

The `:domain` keyword is used to specify a type hierarchy. This specification will expand into `defstruct` forms that implement this hierarchy. It is also possible to write such structure definitions directly into the grammar file. The value of the `:domain` keyword argument must be a list of the following form:

(⟨Root Struct⟩
`:subtype` ⟨Struct Desc⟩
`:subtype` ⟨Struct Desc⟩
...)

⟨Root Struct⟩ ::= ⟨Symbol⟩

```
⟨Struct Desc⟩ ::= ⟨Symbol⟩ |
                  ( ⟨Symbol⟩ :slots (⟨Slot⟩ ...) ) |
                  ( ⟨Symbol⟩ :slots (⟨Slot⟩ ...)
                             :subtype ⟨Struct Desc⟩
                             :subtype ⟨Struct Desc⟩
                  ... )

⟨Slot⟩ ::= ⟨Symbol⟩ | ( ⟨Slot Name⟩ ⟨Filler Type⟩ )
⟨Filler Type⟩ ::= ⟨Symbol⟩
```

This describes a type hierarchy with root node ⟨Root Struct⟩. A node of the hierarchy tree can have children, denoted by `:subtype` followed by the structure description of the child node. Each node can have slots, described as a list following `:slots`. A child node inherits the slots of its parent node. The value of a slot can be type-restricted to ⟨Filler Type⟩.

⟨Root Struct⟩ will be implemented as a structure type directly below the structure type `kb-domain`, i.e. (`kb-domain-p` x) is *true* for any instance x of a subtype of ⟨Root Struct⟩.

**Example domain definition** The grammar defined in "pc2.zb" accepts a simple propositional calculus language with sentences such as

```
walks(agent: John),
```

which yields the following abstract syntax (printed out using the Common Lisp structure printer):

```
#S(ATOMIC-WFF -PREDICATE WALKS
              -ROLE-ARGUMENT-PAIRS #S(ROLE-ARGUMENT-PAIR
                                      -ROLE AGENT
                                      -ARGUMENT JOHN) )
```

The types — such as `ATOMIC-WFF` and `ROLE-ARGUMENT-PAIR` — are defined by the following domain declaration:

```
:domain (PC ;; PC is the root of the hierarchy
         :subtype (Formula
                   :subtype (Propositional-variable :slots (-name))
                   :subtype (Boolean-Expr
                             :slots ((-rand1 Formula)
                                     (-rand2 Formula))
                             :subtype Boolean-Or
                             :subtype Boolean-And))
         :subtype (Boolean-Op :slots (-name))
         :subtype (Atomic-Wff
```

```
            :slots (-predicate
                    (-Role-Argument-Pairs Sequence)))
  :subtype (Role-Argument-Pair :slots (-Role -Argument))
  )
```

# 4 The Zebu Meta Grammar

Using the grammar "zebu-mg" as the `:grammar` argument in the options list indicates that the following grammar is to be preprocessed with the grammar "zebu-mg" before compilation.

The advantages of the meta-grammar (versus the default null-grammar) are a more concise representation of rules, automatic generation of the functions that implement the semantic actions and reversibility of the grammar (generation of printing functions).

The disadvantage is that the semantics is limited to constructing typed feature structures. But these have great expressive power, and furthermore could subsequently be transformed into some other program. Typed feature structures are ideally suited to present abstract syntax. The fact that unification, specialization and generalization are well defined operations on feature structures, makes them appropriate for further transformations by (Zebu-RR). For an introduction into feature structures see [5].

Since there is a restricted way of expressing the semantics of a rule – namely as a typed feature structure – the grammar compiler will be able to generate code for the domain hierarchy and print-functions associated with each type of that domain.

"zebu-mg" is defined in terms of the null-grammar described above[9].

**BNF description of "zebu-mg":**

| | |
|---|---|
| ⟨Domain-Defn⟩ | ::= ⟨Type-name⟩ `":="` ⟨Feat-Term⟩ |
| | [ `"<<"` "print-function:" Identifier `">>"` ] `";"` |
| ⟨zb-rule⟩ | ::= ⟨Non-terminal⟩ `"-->"` ⟨Rhs⟩ `";"` |
| ⟨Rhs⟩ | ::= ⟨Rhs1⟩ ⟨More-Rhs⟩ \| ⟨Kleene-Rhs⟩ |
| ⟨Rhs1⟩ | ::= ⟨Constituent⟩* [ `"{"` ⟨Semantics⟩ `"}"` ] |
| ⟨Constituent⟩ | ::= ⟨Identifier⟩ \| ⟨String⟩ |
| ⟨More-Rhs⟩ | ::= \| ⟨Rhs1⟩ ⟨More-Rhs⟩ |
| ⟨Semantics⟩ | ::= ⟨Feat-Term⟩ |

A ⟨Feat-Term⟩ is a typed attribute value matrix.

[9] You may study the definition of the meta grammar in terms of the null-grammar in the file "zebu-mg.zb".

| | | |
|---|---|---|
| ⟨Feat-Term⟩ | ::= | [⟨Type-name⟩ ":"] ⟨Conj⟩ |
| ⟨Conj⟩ | ::= | "[" ⟨Label-value-pair⟩ * "]" |
| ⟨Label-value-pair⟩ | ::= | "(" ⟨Identifier⟩ ⟨Feat-Term⟩ ")" |
| ⟨Type-name⟩ | ::= | ⟨Identifier⟩ |

This BNF-notation makes use of

1. star (*) for 0 or more repetitions of the preceding constituent
2. bar (|) for alternation
3. brackets ([]) for marking the enclosed constituents as optional
4. a quotation symbol (") for delimiting keywords

The above definition is somewhat oversimplified, since it does not deal with the ".n" notation for ⟨Constituent⟩: if on the right-hand side of a production a non-terminal occurs repeatedly, we can distinguish it by appending "." and a digit to the identifier. The semantics can then unambiguously refer to this identifier.

The semantics is described as a typed feature structure. Names of variables occuring in feature term position corespond to constituent names in the right-hand side of the rule. The effect of applying a rule is to instantiate a feature structure of the type described in the rule semantics, substituting variables with their values.

If the relation between semantics and syntax is one-to-one, the inverse of a parser, a printer, can be generated.

## 4.1 Domain Definition

Although it is possible to specify the hierarchy of domain types using the `:domain` keyword as in section 3.2, a more convenoient syntax is offered by the meta grammar rule ⟨Domain-Defn⟩.

The type definition

*atype* := *super*: [($s_1$) ... ($s_n$)];

will define the type *atype* inheriting from *super*, and having slots $s_1$ through $s_n$.

*atype* := [($s_1$) ... ($s_n$)];

is as above but defines the type *atype* as a subtype of the top type named `KB-domain`.

A slot may be type restricted as in:

*atype* := *super*: [($s_1$ `KB-sequence`)];

which restricts $s_1$ to be of type `KB-sequence`. An optional *print-function* may be specified, as in *atype* := *super*: $[(s_1)]$ `<< print-function:` *print-atype* `>>`;

Here we supply for *atype* its own printer called *print-atype* and no printer will be generated for *atype*.

## 4.2 Example Grammars

**Example Grammar for Arithmetic Expressions**

```
(:name "arith-exp" :grammar "zebu-mg")

;; Domain definition

Arith-exp := Kb-domain: [];
Factor    := Arith-exp: [(-value)] <<print-function: Print-factor>>;
Mult-op   := Arith-exp: [(-arg1) (-arg2)];
Plus-op   := Arith-exp: [(-arg1) (-arg2)];

;; Productions

EE -->  EE "+" TT { +-op: [(-arg1 EE) (-arg2 TT)] }
        |  TT ;

TT --> TT "*" F   { *-op: [(-arg1 TT) (-arg2 F)] }
       | F ;

F -->  "(" EE ")"        { factor: [(-value EE)] }
       | IDENTIFIER      { factor: [(-value IDENTIFIER)] }
       | NUMBER          { factor: [(-value NUMBER)] } ;
```

The semantics of the first rule says that an object of type `+-op` should be created with slot `-arg1` filled with the value of `EE` and `-arg2` filled with the value of `TT`.

**Example Grammar for Propositional Calculus** This grammar defines the same domain as above (3.2). Compiling it generates a parser and a generator.

```
(:name "pc2"
 :package "CL-USER"
 :grammar "zebu-mg")

;; Domain definition
```

```
Formula := kb-domain: [];

 Propositional-variable := Formula: [(-name) ];
 P-Formula              := Formula: [(-content) ];
 Boolean-Expr           := Formula: [(-rand1 Formula) (-rand2 Formula)];
    Boolean-Or          := Boolean-Expr: [];
    Boolean-And         := Boolean-Expr: [];
 Atomic-Wff             := Formula: [(-predicate)
                                     (-Role-Argument-Pairs RAP-list)];

Role-Argument-Pair := kb-domain: [(-Role) (-Argument)];

RAP-list := kb-domain: [(-first) (-rest)];
  RAP-empty-list := RAP-list:[];

;; Productions

Formula --> Propositional-variable
            | Boolean-Expr
            | "(" Formula ")" {P-Formula:[(-content Formula)]}
            | Atomic-Wff;

Propositional-Variable
  --> Identifier {Propositional-variable: [(-name Identifier)]};

Boolean-Expr --> Formula.1 "and" Formula.2
                 {Boolean-And: [(-rand1 Formula.1)
                                (-rand2 Formula.2)]}

                | Formula.1 "or" Formula.2
                  {Boolean-Or: [(-rand1 Formula.1)
                                (-rand2 Formula.2)]};

Atomic-Wff --> Identifier "(" Role-Argument-Pairs ")"
               { Atomic-Wff:
                 [(-predicate Identifier)
                  (-Role-Argument-Pairs Role-Argument-Pairs)]};

Role-Argument-Pairs -->
      { RAP-empty-list: [] }
      | Role-Argument-Pair Role-Argument-Pairs
        { RAP-list: [(-first Role-Argument-Pair)
                     (-rest  Role-Argument-Pairs)]};
```

```
Role-Argument-Pair -->
      Identifier ":" Term
      {Role-Argument-Pair: [(-Role Identifier)
                            (-Argument Term)]};

Term -->  Identifier | Number ;
```

### 4.3 The Kleene * Notation

The meta-grammar "zebu-mg" provides an abbreviated notation for repeated occurrences of a non-terminal, separated by a keyword. The syntax for this is:

$$\langle\text{Kleene-Rhs}\rangle \quad ::= \langle\text{Identifier}\rangle \; \texttt{*} \; \langle\text{String}\rangle \tag{1}$$
$$\langle\text{Kleene-Rhs}\rangle \quad ::= \langle\text{Identifier}\rangle \; \texttt{+} \; \langle\text{String}\rangle \tag{2}$$

The meaning of (1) is that 0 or more occurrences of the constituent named by ⟨Identifier⟩ and separated by ⟨String⟩ will be accepted by this rule, and that the sequence of the results of these constituents will be returned as the semantics of ⟨Kleene-Rhs⟩. The meaning of (2) is the same, except that at least one occurrence of the constituent has to be found.

The semantics of a ⟨Kleene-Rhs⟩ production is an implicit sequence construction. The Kleene-constituent (⟨Identifier⟩ concatetenated with `*` or `+`) is bound in the semantics of the production, e.g.

```
Disjunction --> Conjunction+ "|"
                {Disj: [(-terms Conjunction+)]};
```

builds a structure of type `Disj` with the `-terms` slot filled by the value of the Kleene-constituent `Conjunction+`.

**Example grammar using Kleene * Notation**

```
(:name "mini-la" :grammar "zebu-mg" )

;; Domain definition

Program := [(-stmts kb-sequence)];
Application := [(-function) (-args kb-sequence)];

;; rules
```

```
Program --> "begin" Stmt+ ";" "end"
             { Program: [(-stmts Stmt+)] } ;

Stmt    --> Identifier | Appl | Program ;

Appl    --> Identifier "(" Arg* " " ")"
             {Application: [(-function Identifier) (-args Arg*)]};

Arg     --> Identifier | Number | Appl ;
```

Compiling this grammar generates a parser/unparser (i.e. the printing routines are generated automatically).

```
(zb:read-parser "begin A; B ; C end"
                :grammar (zb:find-grammar "mini-la"))
```

returns a structure of type `PROGRAM` which is printed in the syntax of "mini-la":

```
begin A;B;C end
> (describe *)
begin A;B;C end is a structure of type PROGRAM.
It has 1 slot, with the following values:
 -STMTS:                        A;B;C

(describe (PROGRAM--STMTS *))
A;B;C is a structure of type KB-SEQUENCE.
It has 2 slots, with the following values:
 FIRST:                         A
 REST:                          B C
```

# 5 Using the Compiler

## 5.1 Compiling a grammar

The `Zebu`-compiler[10] can be called using any of the following functions:

`zebu-compile-file` *function*

(*grammar-file* `&key` *output-file grammar verbose*)

[10] For installation see appendix A.

This compiles the LALR(1) grammar in a file named *grammar-file*. The *output-file* defaults to a file with the same name as *grammar-file* but type "`tab`". The grammar used for compilation defaults to the null-grammar. If *verbose* is *true*, conflict warnings will be printed. `zebu-compile-file` returns the pathname of *output-file*.

**Example:**

```
(let ((*warn-conflicts* t)
      (*allow-conflicts* t))
  (zebu-compile-file "dangelse.zb"
                     :output-file "/tmp/dangelse.tab"))

; Zebu Compiling (Version 2.0)
; "~/zebu/test/dangelse.zb" to "/tmp/dangelse.tab"

Reading grammar from dangelse.zb

Start symbols is: S

4 productions, 8 symbols
.........9 item sets
.........
.........
;;; Warning: ACTION CONFLICT!!!-- state: 8
;;;          old entry: (6 :S 2)  new entry: (6 :R 2)
;;;
;;; Warning: Continuing to build tables despite conflicts...
;;;          Will prefer old entry: (6 :S 2)

Dumping parse tables to /tmp/dangelse.tab
#P"/tmp/dangelse.tab"
```

`*warn-conflicts*` *variable*

If *true* during LALR-table construction, shift-reduce conflicts will be reported. By default, `*warn-conflicts*` is false.

`*allow-conflicts*` *variable*

If *true* during LALR-table construction, shift-reduce conflicts will be resolved in favor of the old entry. By default, `*allow-conflicts*` is *true*.

`*check-actions*` *variable*

If *true* the semantic action associated with a production will be compiled at grammar compilation time in order to display possible warning messages. By default, `*check-actions*` is false.

(`compile-slr-grammar` (*grammar-file* `&key` *output-file grammar*)   *function*

This is like `zebu-compile-file`, but an SLR-table will be made.
Example:

```
(compile-slr-grammar "dangelse.zb"
    :output-file "/tmp/dangelse.tab")

Reading grammar from dangelse.zb

Start symbols is: S

4 productions, 8 symbols
.........9 item sets

Dumping parse tables to /tmp/dangelse.tab
#P"/tmp/dangelse.tab"
```

`compile-lalr1-grammar` (*grammar-file* `&key` *output-file grammar*)   *function*

This is like `zebu-compile-file`, but does not expand logical pathnames.
Example:

```
(compile-lalr1-grammar "dangelse.zb"
                       :output-file "/tmp/dangelse.tab")

Reading grammar from dangelse.zb

Start symbols is: S

4 productions, 8 symbols
.........9 item sets
.........
.........
Dumping parse tables to /tmp/dangelse.tab
#P"/tmp/dangelse.tab"
```

## 5.2 Loading a grammar

`zebu-load-file` (*filename* `&key` *verbose*)   *function* *filename* should be the name of a compiled grammar file, i.e. a file of type "`tab`". If such a file can be found, it will be loaded, returning the grammar object needed for parsing. In case a domain-file was generated by compiling the grammar, it will also be loaded. The type of the domain-file is the first for which a file exists, by examining the lists

`*load-binary-pathname-types*` and
`*load-source-pathname-types*`

in turn.

The keyword argument *verbose* defaults to *true*.

**Example:**

```
(zebu-load-file "/tmp/dangelse.tab")
<Zebu Grammar: dangelse>
```

It is possible to have many grammars loaded concurrently. Given the name of a grammar, one can find a grammar that has been loaded by:

`find-grammar` (*name*) *function*

*name* must be a string. If a grammar of the same name (ignoring case) has been loaded, the grammar object is returned, else *false* is returned.

**Example:**

```
(find-grammar "dangelse")
<Zebu Grammar: dangelse>
```

## 5.3 Parsing a string with a grammar

`read-parser` *function*
(*string* `&key` *grammar junk-allowed print-parse-errors error-fn start*)

The argument of the `:grammar` keyword defaults to `*current-grammar*` initially is bound to the null-grammar, e.g.

```
(read-parser ⟨string⟩ :grammar (find-grammar ⟨name⟩))
```

is equivalent to

```
(setq zebu:*current-grammar* (find-grammar ⟨name⟩))
(read-parser ⟨string⟩)
```

`read-parser` parses the string starting at the position indicated by `:start` (default 0).

`read-parser` takes the keyword argument `:junk-allowed`, which if *true* will return as second value an index to the unparsed remainder of the string in case not the entire string was consumed by the parse.

The keyword `:junk-allowed` has the same meaning as in the Common Lisp function `read-from-string`.

`:print-parse-errors` controls the printing of errors during parsing and defaults to *true*.

`:error-fn` is a function used to report errors, it defaults to the Common Lisp `error` function.

**Example:**

```
(read-parser "if f then if g then h else i"
             :grammar (find-grammar "dangelse"))
("if" F "then" ("if" G "then" H "else" I))

(read-parser "1 + a" :grammar (find-grammar "ex1"))
(+OP (EXPRESSION (TERM (FACTOR 1)))
     (TERM (FACTOR A)))
```

## 5.4 Parsing from a file with a grammar

`file-parser` (*file* `&key` *grammar print-parse-errors verbose*) *function*

`file-parser` parses expressions using the grammar specified by `:grammar`, reading from *file*. It returns a list of the parse-results, i.e. what would have been returned by `read-parser`.

The `:grammar` argument defaults to `*current-grammar*` – which initially is bound to the "null-grammar".

`:print-parse-errors` controls the printing of errors during parsing and defaults to *true*.

`:verbose` controls whether printing of parse-results occurs, and defaults to *true*.

The processing of comments by `file-parser` can be influenced by the following variables:

- `zb:*comment-brackets*` is a list of bracket pairs. Everything between any of bracket pairs is ignored. Initially `zb:*comment-brackets*` is set to `(("#\|" . "|#"))`.

- `zb:*comment-start*` A line beginning with this character is ignored. Initially `zb:*comment-start*` is set to the semicolon character: `#\;`

**Example:**

```
(zb:file-parser "sample-ex1" :grammar (zb:find-grammar "ex1"))
...
```

### 5.5 Parsing from a list of tokens

`list-parser` (*token-list* `&key` *grammar junk-allowed*) *function*

`list-parser` is like `read-parser` except that the tokens that are passed by the scanner to the driver are already given as the elements of *token-list*. This function is useful if the options for controlling lexical analysis given in section 3.1 are insufficient.

**Example:**

```
(let ((*current-grammar* (find-grammar "ex1")))
   (list-parser '(1 "+" x "*" y)))
(+OP (EXPRESSION (TERM (FACTOR 1)))
     (*-OP (TERM (FACTOR X)) (FACTOR Y)))
```

## 6 Lexical Analysis

### 6.1 Customization and Regular Expressions

It should only seldomly be necessary to write a lexical analyser. Before you write your own lexical analyzer, check whether the following variables and keywords would suffice to parameterize lexical analysis:

```
zb:*comment-start*
zb:*comment-brackets*
zb:*disallow-packages*
zb:*preserve-case*
zb:*case-sensitive*
:case-sensitive
:identifier-start-chars
:identifier-continue-chars
:string-delimiter
:symbol-delimiter
:white-space
:lex-cats
```

## 6.2 Introducing new Categories by Regular Expressions

The keyword `:lex-cats` takes as argument an association list of the form:

```
((⟨Category⟩ ⟨Regular Expression⟩) ...)
```

⟨Category⟩ is a symbol naming a lexical category and ⟨Regular Expression⟩ is a string representing a regular expression as defined in the GNU Emacs Lisp Manual [7]. The regular expression will be compiled into a Common Lisp function and invoked by `read-parser` before the built-in categories (Identifier, String, Number) are examined. The categories can be used in rules like any of the built-in categories.

The regular expression compiler[11] handles the following constructs:

**.** Period matches any single character except a newline.

***** repeats preceding regular expression as many times as possible.

**+** like * but must match at least once.

**?** like * but must match once or not at all.

**[...]** '[' begins a character set, which is terminated by ']'.
Character ranges can be indicated, e.g. a-z, 0-9.

**[^...]** forms the complement character set.

**$** matches only at the end of a line.

**\\(... \\)** is a grouping construct.

**\\ ⟨digit⟩** means: accept the same string as was matched by the group in position ⟨digit⟩.

**Example:**

```
:lex-cats ((BibTeX-in-braces "{[^\\n}]*}"))
```

defines a new category `BibTeX-in-braces` which matches anything starting with "{", ending in "}", and not containing either a newline or "}".

```
:lex-cats ((Ratio_Number "-?[0-9]+/[0-9]+")
   (Simple_Float "-?[0-9]*\\.[0-9]+"))
```

defines the syntax for rationals and floating point numbers. Note that the period needs to be escaped, since it is a special character of the regular expression language.

[11] Thanks to Lawrence E. Freil who wrote the main part of the Regular Expression Compiler.

## 6.3 The functional interface to the parsing engine

In case the above parameterization facilities for lexical analysis are insufficient or you want to use an existing lexical analyzer, you need to understand the functional interface to the parsing engine as implemented by the `lr-parse`.

`lr-parse` *function*
(*next-sym-fn error-fn grammar* `&optional` *junk-allowed last-pos-fn*)

`lr-parse` returns the result of parsing the token stream produced by *error-fn* with *grammar* by the LALR(1) method. In case *junk-allowed* is *true* it produces as second value a handle to the yet unconsumed token stream by calling the function *last-pos-fn*.

*next-sym-fn* should be bound to a generator function — a function of no arguments — that will be called to produce the next token. It should return two values: (1) the token found and (2) the category of the token (obtained by the function `categorize`.

*error-fn* is the function to be called in case of an error. *grammar* is the grammar object that contains important information for lexical analysis, (e.g. the table of keywords).

To understand the interface to `lr-parse`, consider how `list-parser` (described above) might have been defined:

```
(defun list-parser (token-list &key (grammar *current-grammar*)
                                    junk-allowed)
  (let ((last-position token-list)
        token1)
    (check-type token-list list)
    (lr-parse
     ;; The LEXER supplied to the parsing engine:
     #'(lambda ()
         (if (null token-list)
             (end-of-tokens-category grammar)
           (progn
             (setq last-position token-list
                   token1 (pop token-list))
             (categorize token1 grammar))))
     ;; The error function supplied to the parsing engine:
     #'(lambda (string)
         (error "~S~% Remaining tokens: ~S~{ ~S~}"
                string token1 token-list))
     grammar
     junk-allowed
     ;; Function that returns the remaining unparsed token-list
     #'(lambda () last-position))))
```

`end-of-tokens-category` (*grammar*) *function*

`end-of-tokens-category` returns two values: a token signifying the end of the token stream and the appropriate lexical category.

`categorize` (*token grammar*) *function*

`categorize` returns the *token* itself and its category, a number that represents one of `number`, `identifier`, `string` or a particular terminal token of the grammar.

# 7 Future Work

Translation involves three processes:

- parsing
- transformation
- generation

Zebu is a tool that helps in 1 and 3. There are cases where 2 reduces to the identity function, since the abstract syntax is the same for the source and the target language of translation. Examples for these "syntactic variants" are infix and prefix notation for arithmetic or boolean expressions.

In general, the situation is more complicated. For languages with the same expressive power, some transformation process can be defined. Between languages with different expressive power such a transformation is not always possible. For a language that is not Turing complete, it is not possible to express every computation, e.g. SQL cannot express rcursion, and hence it is not possible to express the "ancestor" relation (which is recursively defined). A technique to represent transformation are "rewrite rule systems". The Refine language contains a rewrite-rule mechanism in which the rules are in terms of patterns of the concrete syntax. We have implemented a rewrite-rule system based on typed feature structures, called Zebu-RR, which will be described in a future report.

# A Installation

There are two ways to install Zebu:

- Installation using `defsystem`

  This makes it easier to load and compile grammars, since one does not need to remember the location of a module in a directory structure and the particular compilation and loading functions. To install, follow the directions in `ZEBU-sys.l`. You need the portable `defsys` for that. This is available as `Defsys.tar.Z` at the same place as `zebu-???.tar.Z`.

  The file `ZEBU-sys.l` is used to load or compile Zebu, which actually consists of two systems (defined by `defsystem`)

  | | |
  |---|---|
  | Zebu | the runtime system |
  | Zebu-compiler | the compiler |

- Installation without `defsystem`
  If you don't want to use `defsystem`, load the file `COMPILE-ZEBU.l`, which compiles the Zebu files in the right order.

  After loading the file `ZEBU-init.l` you can call:

  `(zb:zebu)` to load the runtime system
  or
  `(zb:zebu-compiler)` to load the grammar compiler.

# References


[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison Wesley, New York, 1979.

[2] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988.

[3] Michael R. Genesereth. An agent-based framework for software interoperability. Technical Report Logic-92-02, Departmentof Computer Science, Stanford University, Stanford, 1992.

[4] Michael R. Genesereth, Richard Fikes, et al. Knowledge interchange format, version 3.0. reference manual. Report Logic-92-1, Logic Group Report, Computer Science Department, Stanford University, Stanford, June 1992.

[5] Mark Johnson. *Attribute Value Logic and the Theory of Grammar*. Center for the Study of Language and Information, Stanford, 1988.

[6] Joachim Laubsch and Derek Proudian. A case study in REFINE: interfacing modules via languages. Report HPL-STL-TM-88-11, Hewlett Packard, 1988.

[7] Bill Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.* The Free Software Foundation, Cambridge, MA, December 1990.

[8] Reasoning Systems, Palo Alto, 3260 Hillview Ave., CA 94304. *Refine User's Guide*, 1989.

[9] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at KESTREL institute. *IEEE Transactions on Software Engineering*, SE-11:1278–1295, November 1985.

# Contents