Essence—An LR Parser Generator for Scheme

Version 2.0

Mike Sperber
sperber@deinprogramm.de
Peter Thiemann
thiemann@informatik.uni-freiburg.de

Abstract

Essence is a generator for LR(k) and SLR(k) parsers in Scheme. The generated parsers perform error recovery, and are highly efficient. Testing and debugging a parser does not require an edit—generate—compile—test cycle. Rather, the parser generator results from a general parser which takes the input grammar as a parameter; no generation and recompilation is necessary to try out changes to a grammar. The generated parsers result from the general parser by an automatic program transformation called *partial evaluation* [9, 10]. This guarantees consistency and ensures correctness. However, no specific knowledge of partial evaluation is required to use Essence.

This document assumes elementary knowledge about S-attributed grammars and LR parsing, available in almost any compiler construction textbook [2, 7, 1, 11]. It may also be helpful to study the documentation of more traditional parser generation packages such as Yacc [4] or Bison [3].

1 Introduction

The pragmatics of using Essence are slightly different from that of using other parser or parser generation packages.

At the heart of parsing is, as usual, a context-free grammar. Essence provides a new syntactic form `define-grammar` (section 3) which embeds a language for attributed context-free grammars into Scheme. This is different from other parser generation packages which either represent a context-free grammar as an S-expression object or in some special syntax in a special file.

Given a grammar, parsing can proceed in one of two modes:

- A general `parse` procedure will accept a grammar, a parsing method (SLR or LR), a lookahead size, and an input, and produce the result of parsing and attribute evaluation. This mode of operation allows instant turnaround, but also parses very slowly. It is good for incremental development, but impractical for production parsers.

- A parser generator (automatically generated from the implementation of `parse`) produces a specialized parser from a grammar, a parsing method, and a lookahead. The specialized parser only accepts an input as an argument, but is otherwise identical in operation to the general parser. It is highly efficient, and therefore good for production use.

2 Prerequisites

Currently, Essence itself only runs under Scheme 48 [6]. However, the generated parsers run under any R^{5}RS [5] Scheme. Moreover, the dependencies on non-standard features of Scheme48 which Essence uses have been carefully factored out with the help of the Scheme 48 module system; making it work in any given Scheme implementation is not hard.

The Essence distribution contains three files to be loaded into Scheme 48's configuration package: `interfaces.scm`, `packages.scm`, and `src/cps-lr-genext.config.scm`. To make Essence available within Scheme 48, it is always necessary to load `interfaces.scm` (via `,config ,load interfaces.scm`) and `packages.scm` (via `,config ,load packages.scm`). To use the parser generator, it is also necessary to load `genext/genext-packages.scm` from the Essence distribution, as well as `src/cps-lr-genext.config.scm`. To make this process more convenient, Essence includes a file `load-essence.scm` which is written in the exec language of Scheme 48. To load it, invoke this in the Scheme 48 REPL:

```
,exec ,load load-essence.scm
```

This will output a warning which can safely be ignored:

```
Warning: undefined variables
         #{Package 250 config}
         essence-cps-lr-genext
         (&warning)
```

3 Grammars

Context-free grammars are at the heart of parser generation. Essence allows specifying so-called *S-attributed grammars* with evaluation rules for synthesized attributes. The assumption is that each node in the parse tree carries an instance of exactly one synthesized attribute, and an Essence grammar provides an expression describing how to compute the attribute along with each production.

The structure `essence-grammars` provides a defining form for grammars. (It also provides numerous accessors and algorithms over grammars. However, these are not relevant for using Essence.)

An Essence grammar consists of two data objects: a representation of the grammar itself, and an enumeration which is needed to symbolically encode the input to the parser. The macro `define-grammar` from the `essence-grammars` structure form defines both:

(`define-grammar` ⟨*variable*$_1$⟩ ⟨*variable*$_2$⟩
⟨*terminals*⟩ ⟨*start-symbol*⟩ ⟨*rules*⟩) syntax

Syntax: ⟨Terminals⟩ has the form

(⟨terminal⟩ ...)

where each ⟨terminal⟩ is an ⟨identifier⟩. ⟨Start-symbol⟩ must be a an identifier. ⟨Rules⟩ has the form:

(⟨rule⟩ ...)

where each indivial ⟨rule⟩ has the form

(⟨nonterminal⟩ ((⟨grammar-symbol⟩ ...) ⟨attribution⟩) ...)

where each ⟨grammar-symbol⟩ is either a ⟨nonterminal⟩, a ⟨terminal⟩, or `$error`. (The latter is for directing error recovery (see sec. 6) ⟨Attribution⟩ is a Scheme expression.

The set of nonterminals is defined by the ⟨nonterminals⟩s of the ⟨rule⟩s. The nonterminals must be disjoint from the terminals. Moreover, lThe start symbol must be a nonterminal.

Semantics: `Define-grammar` defines a context-free grammar along with an enumeration type for its symbols. `Define-grammar` binds a data object representing the grammar to its first argument, and an enumeration type for its symbols to its second argument.

The third argument to `define-grammar` is a list of nonterminals, the fourth a list of terminals. The fifth argument is the start symbol (which must be one of the nonterminals), then comes a list of the grammar rules.

A grammar rule specifies a list of productions for the specified nonterminal. Each subform ((⟨grammar-symbol⟩ ...) ⟨attribution⟩) specifies a right-hand side and an attribution.

The attribution is a Scheme expression, which may have free variables `$`i, where i ranges from 1 to the number of symbols on the right-hand side of the production. During parsing, the Essence parser binds `$`i to the attribute instance of the ith symbol on the right-hand-side when evaluating the attribution.

Here is a simple example grammar for arithmetic expressions:

```
(define-grammar g10 g10-symbol
  (+ - * / l r n)
  E
  ((E ((T) $1)
      ((T + E) (+ $1 $3))
      ((T - E) (- $1 $3)))
   (T ((P) $1)
      ((P * T) (* $1 $3))
      ((P / T) (/ $1 $3)))
```

```
(P ((n) $1)
   ((l E r) $2))))
```

This definition establishes an enumeration type `g10-symbol` with components (in that order):

- `$Start` (for the fresh start symbol generated by `define-grammar`),
- the nonterminals in the same order as in the `define-grammar` form,
- `$Error`, and
- the terminals in the same order as the `define-grammar` form.

The members of the enumeration may be accessed using the `enumerated` structure that comes with Scheme 48: `(enum g10-symbol +)`, for instance, is an expression whose value is an exact non-negative integer, corresponding to the position of `+` in the enumeration, in this case 5 (it's the first terminal, after three nonterminals, `$start`, and `$error`).

4 Running a Parser

Parsing with respect to a grammar does not require generating a specialized parser along with the associated overhead of compiling and loading. Essence provides general parsers which accept a grammar as input and parse "right away." This allows incremental debugging and development of attributed grammars to be used with Essence.

Essence actually comes with a number of different implementations of LR parsing. The `packages.scm` configuration file contains definitions for a range of structures all with the interface `essence-parser-interface`. The one intended for production use is in the `essence-cps-lr` structure whose implementation resides in `src/cps-lr.scm`. `Essence-parser-interface` describes only one binding called `parse`:

(`parse` *grammar lookahead method trace-level input*) procedure

- *Grammar* is a grammar defined by `define-grammar`.
- *Lookahead* is a non-negative integer denoting the lookahead the parser uses.
- *Method* is a symbol, either `lr` or `slr`, specifying the parsing method used—either full LR parsing or SLR parsing.
- *Trace-level* is 0, 1, 2, 3 depending on the amount of tracing desired. 0 means no tracing, 1 means that the closure of the state where an error is encountered is passed to `parse-error`. 2 means that the states encountered during parsing are printed, 3 means that the complete closures are printed.

- *Input* is a list of pairs; each pair consists of an enumerand of the terminals of *grammar* and the corresponding attribute value. It is the input to the parser.

  The enumerands are the *ordinal numbers* of the enumerated values. These are easiest obtained via the `enumerated` structure of Scheme 48. In the absence of an implementation of `grammar` and `enumerated`, the list of components (see section 3) describes the mapping between grammar symbols and enumerands; the enumerands are 0-based.

`Parse` returns the result of attribute evaluation on the parse tree induced by *input*. This is ultimately the result of the attribution associated with the start production.

The `parse` procedure can be applied to sequence representations other than lists: To this end, Essence includes a parameterized structure `make-essence-cps-lr`, which takes a structure with interface `essence-list-inputs-interface` as an argument. This interface includes only the three procedures `input-null?`, `input-car`, and `input-cdr`, which are used in the same way as `null?`, `car`, and `cdr`. The "default" implementation used by the `essence-cps-lr` structure is `essence-list-inputs`, which defines these to the list procedures.

If the grammar contains productions containing `$error` symbols, the parser will attempt error recovery (see section 6) when possible.

5 Generating a Specialized Parser

In addition to simply calling `parse`, Essence also allows the generation of highly efficient specialized parsers with respect to a grammar, lookahead, and parsing method. Essence offers a (Unix) batch version of the parser generator, as well as a Scheme 48 package which allows access from within a REPL.

In order to run, the specialized parsers require definitions for `input-null?`, `input-car`, and `input-cdr` as described in the previous section. Moreover, they require an `parse-error` procedure. The specialized parser will call `parse-error` when an unrecoverable error occurs. It has the following signature:

(`parse-error` *message closure error-status recovering? symbol input*) procedure

Here, *closure* is either `#f` or the LR closure in which the error occurred, depending on the tracing level. *Error-status* is either `#f` if this is the first parse error, or a non-negative exact integer saying how many lexemes have been consumed since the last error. *Recovering?* says whether this is an error from the recovering action of the parser (see section 6, and hence `parse-error` may return, or whether the parser cannot recover, and hence `parse-error` should not return. *Symbol* is the symbol on which the parser tried to shift, and *input* is the remaining input.

Batch operation

Installation of Essence creates a binary called `essence`. When called with a `--help` or `-h` argument, it prints a synopsis of its syntax:

```
essence ( -g goal-proc | --goal-proc=goal-proc | --goal-procedure=goal-proc )
        ( -m method | --method=method )
        ( -l lookahead | -lookahead=lookahead )
        ( -s | --states)
        ( -p | --pp --pretty-print)
        ( -6 library-name | --r6rs-library=library-name )
        ( -i library-name | --r6rs-import=library-name )
        input-file grammar-name output-file
```

- `Input-file` is a Scheme source file which Essence will load into a package with the standard R^{5}RS Scheme bindings as well as `define-grammar` (section 3). The file must contain the definition for at least one grammar.
- `Grammar-name` is the name of the grammar defined in `input-file` for which Essence is to generate a specialized parser.
- `Output-file` is the name of the file into which Essence writes the specialized parser.
- `Goal-proc` is the name of entry procedure into the specialized parser. It will accept just one argument, an input list. The default is `parse`.
- `Method` is the parsing method, either `slr` or `lr`. The default is `slr`.
- `Lookahead` is the lookahead size, a non-negative number. The default is `1`.

If the `-s` or `--states` option is supplied, Essence will print the states of the LR automaton to standard output.

If the `-p`, `--pp`, or `--pretty-print`option is supplied, Essence will pretty-print the source code of the generated parser instead of justing using `write`. Note that this increases the size of the output substantially.

In addition to the `goal-proc` procedure, the output file also contains a `define-enumeration` form that defines the mapping between grammar symbols and enumeration values. The form has the following syntax:

(`define-enumeration` ⟨*identifier*⟩ (⟨*symbol*⟩ . . .))

The identifier is the name of the enumeration type of the grammar, and the ⟨symbol⟩s are the names of all terminals and nonterminals of the grammar (including `$error`). The first ⟨symbol⟩ is mapped to enumeration value 0, the second to 1, and so forth. The `define-enumeration` form is suitable for use with Scheme 48's `enumerated` package.

If the `-6` or `--r6rs-library` option is supplied, Essence will generate an R^{6}RS library [8]. The library will have the name supplied as an argument to the option. In this case, the `-i` or `--r6rs-import` option will also need to be supplied for each import the generated library is to have. Note that even `(rnrs base)` will need to be imported directly. For example, following command-line fragment:

```
-6 '(org example parser)' -i '(rnrs base)' -i '(org s48 essence support)'
```

will create an R^{6}RS library called `(org example parser)`, whose implementation imports `(rnrs base)` and `(org example parser)`.

The file `src/r6rs-support.scm` contains an example R^{6}RS library with parser support code suitable for import into an Essence parser library.

REPL operation

The `generator-packages.scm` configuration file defines a structure `essence-cps-lr-generate` which offers a procedure that generates specialized parsers:

(`generate-parser` *grammar lookahead method goal-name*) procedure

Grammar, *lookahead*, and *method* are as with `parse` (section 4).

The *goal-name* argument to `generate-parser` is a symbol which names the entry procedure into the parser. `Generate-parser` generates a list of S-expressions which, when written out sequentially, represent the code of the specialized parser.

6 Error Recovery

Essence parsers can perform recovery from parsing errors in the manner of Yacc [4] and Bison [3]. The basic idea is that the author of a grammar can specify special error productions at critical places in a grammar designed to "catch" parsing errors. This allows printing specially tailored error messages as well as some control over attribute evaluation in such a case.

Error productions contain a special grammar symbol `$error` on the right-hand side. (`$Error` must not be explicitly declared as a terminal or nonterminal in the `define-grammar` form.)

When an error occurs during parsing, an Essence parser pretends that it has just seen `$error` in the input. It will go back to the last LR state capable of accepting `$error` as the next symbol in the input. Moreover, it discards terminals from the input until the next input terminal is acceptable as the next input symbol *after* it has consumed `$error`. Subsequently, the parser resumes work as usual.

To prevent excessive avalanching of error messages, the `parse-error` procedure (see section 5) should examine the *error-status* argument, and typically

assure that a certain number of terminals have been consumed since the last error before reporting a new one.

Here is an example for the constant arithmetic expressions grammar guaranteed to catch all errors:

```
(define-grammar g10-error g10-error-symbol
  (+ - * / l r n)
  E
  ((E ((T) $1)
      (($error) 0)
      ((T + E) (+ $1 $3))
      ((T - E) (- $1 $3)))
   (T ((P) $1)
      ((P * T) (* $1 $3))
      ((P / T) (/ $1 $3)))
   (P ((n) $1)
      ((l E r) $2)
      ((l $error r) 0))))
```

Apart from the first catch-all rule containing `$error`, the parser will also, when encountering an error inside a parenthesized expression, skip until the next closing parenthesis to resume parsing.

7 Example Session

To see all this in action, we work through a little example involving one of the provided example grammars. First, to start the system, type

```
% scheme48 -i essence.image -h 8000000
```

at the shell prompt. The mechanism to define a grammar are available from structure `essence-grammars` (see 3). To open the structure type

```
> ,open essence-grammars
```

to the Scheme48 system, followed by

```
> ,load examples/toy-grammars.scm
```

to load the definitions for some simple grammars.

Loading the corresponding inputs requires enumerated values, hence

```
> ,open enumerated
> ,load examples/toy-inputs.scm
```

defines the example inputs.

Open the parser module by typing

```
> ,open cps-lr
```

to gain access to the `parse` function (see Sec. 4).

As a sample run, consider the grammar `g10` which specifies arithmetic expressions. The terminals `l` and `r` stand for opening and closing brackets, whereas `n` stands for a number.

```
> (parse g10 1 'lr 0 i10-1)
147
```

To specialize a parser requires to open the structure `essence-cps-lr-generate`:

```
> ,open essence-cps-lr-generate
```

The `generate-parser` function from this structure (see Sec. 5) performs the specialization:

```
> (generate-parser g10 1 'lr 'expr-parser)
```

To perform the same specialization task via the command line interface type

```
% ./essence -g expr-parser -m lr -l 1 \
            examples/toy-grammars.scm g10 /tmp/expr-parser.scm
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[3] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.

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

[5] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[6] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[7] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.

[8] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/final/r6rs.pdf`, Sep 2007.

[9] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, USA, June 1995. ACM Press.

[10] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[11] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.