

Ocaml Summer Project Proposal

Improvements to Menhir

François Pottier

April 16, 2008

Participants:

Name	Raja Boujbel	Guillaume Bau
Position	Student (M1)	Student (M1)
Institution	Université Pierre & Marie Curie	Université Pierre & Marie Curie
Personal address	12 rue Geoffroy Saint-Hilaire 75005 Paris France	21-23 rue Pasteur, appt. 603 94270 Le Kremlin Bicêtre France
Phone	+ 33 6 62 71 77 63	+33 6 61 78 50 20
Email	boujbel.raja@gmail.com	guillaume.bau@gmail.com

Faculty member:

Name	François Pottier
Position	Senior Researcher
Institution	INRIA Paris-Rocquencourt
Address	BP 105 78153 Le Chesnay Cedex France
Phone	+33 1 39 63 52 64
Fax	+33 1 39 63 54 00
Email	francois.pottier@inria.fr

Title: Improvements to Menhir

Abstract

The purpose of this project is to make several improvements to the [Menhir](#) parser generator. Three distinct items are identified:

- equip Menhir with a reference interpreter for LR automata;
- equip Menhir with a new, table-based, back-end;
- equip Menhir with a new error reporting mechanism.

The desired effect of this project is to make Menhir a more appealing tool, so that its use becomes more widespread. Menhir offers many advantages over `ocamlyacc`, but the lack of a table-based back-end limits its adoption by potential users. Furthermore, the current error handling mechanism, inherited from `yacc`, is difficult to use. Although the more declarative mechanism that we have in mind would constitute an incompatible change, it could be an important improvement.

Project description

Motivation Menhir is superior to ocaml yacc in many ways. Two features that make an important difference are:

- Menhir’s grammar specification language is significantly more flexible, and allows grammars to be developed more modularly;
- Menhir explains LR(1) conflicts in a way that humans can understand, by reporting two distinct parse trees for a single partial sentence.

However, Menhir’s adoption has been hindered by the fact that it produces large parsers, which the ocaml compiler finds difficult to deal with. This usually means large compilation times and large executables. Sometimes, the ocaml compiler fails, and there is no executable at all.

One goal of this project is to remove this obstacle to Menhir’s adoption. If the project is successful, it will become conceivable for Menhir to be shipped as part of the standard ocaml distribution, and officially recommended for use in new projects.

Another goal is to replace the error handling mechanism with a different one. Menhir currently attempts to emulate yacc’s (and ocaml yacc’s) “error” pseudo-token, whose semantics is quite subtle. To be honest, I (François Pottier) do not know what its exact semantics is supposed to be. Should the “error” token be inserted in the token stream in front of, or instead of, the current token? Is termination guaranteed? Menhir’s implementation has a few known problems. Furthermore, the “error” pseudo-token is difficult to use. The introduction of new productions affects the automaton, possibly introducing new conflicts.

The project is divided into three items, as follows.

Item 1: a reference interpreter The first item consists in implementing a reference interpreter. A reference interpreter allows running the LR automaton over a user-supplied sentence. It can print various kinds of information, such as:

- is the sentence valid or invalid?
- if valid, what is the parse tree?
- if invalid, in what state is the error detected?
- which sequence of actions does the automaton perform?
- which sequence of states does the automaton go through?
- does the automaton go through a conflict state?

The reference interpreter also allows running the automaton over a set of user-supplied sentences, and provides coverage information: for instance, are all conflict states reached?

There are several reasons for writing a reference interpreter:

- this will help the students become familiar with LR parsing in general, and with Menhir’s implementation in particular;
- the interpreter should make it easier to test the new back-end, by allowing comparisons between the reference interpreter and the two back-ends;
- the interpreter provides a mapping of user-supplied sentences to automaton states, which is required by the new error reporting mechanism.

Sentences can be supplied by the user in one of two forms: either in abstract form, as a sequence of symbolic token names, or in concrete form, as a sequence of characters. Abstract form is more powerful, as it allows sentences to be composed of both non-terminal and terminal symbols. Concrete form seems more pleasant than abstract form, but is limited to terminal symbols, and requires a user-provided lexer. Should this lexer be provided as an ocaml library, or as a stand-alone executable? This could be a somewhat thorny issue.

Item 2: a new, table-based back-end The current back-end compiles the LR automaton directly to ocaml code. This approach produces code that is competitive in terms of efficiency, but usually quite large. In fact, it is sometimes so large that the ocaml compiler is unable to accept it.

The proposed new back-end will compile the LR automaton to a compressed transition table, which is then interpreted by a piece of C or ocaml code. This traditional approach, implemented in `ocamlyacc`, produces compact tables.

The architecture of Menhir is fairly simple. Like most compilers, it is organized as a sequence of phases: the grammar specification is parsed and checked for well-formedness; some simple analyses of the grammar are performed; an LR(1) automaton is built; last, the automaton is translated to ocaml code.

In principle, very few changes to this existing code base will be necessary. The new back-end will compile the automaton down to a compressed transition table. An interpreter for these tables, written in C or ocaml, will be designed and implemented. The end user will be provided with a command line switch for choosing between the old and new back-ends. When using the new back-end, the user will link the interpreter together with his or her executable.

An open question is whether we will re-use the compression algorithm and table format used by `ocamlyacc`, or design a new algorithm and format, based on existing sources in the scientific literature [8, 2, 1, 7]. If technically possible, the former approach sounds simpler, and could perhaps allow us to re-use the interpreter that ships with ocaml's standard library (`Parsing`). However, there are differences between `ocamlyacc` and Menhir in the treatment of source code locations, and in the fact that Menhir-generated parsers are re-entrant, whereas `Parsing` is not. For this reason, a new interpreter will probably be needed.

Item 3: a new, declarative error reporting mechanism The “error” pseudo-token allows both error reporting and error recovery. We would like to explore a new mechanism, which is simpler, but only allows error reporting. Error recovery, if desired, can (in simple cases) be implemented outside the LR automaton, by letting the lexer segment the input stream.

The new mechanism, invented by Clinton Jeffery [5] and implemented in Merr [4] and Jacc [6], is based on a simple idea. In order to report good error messages, we wish to associate, with each state of the automaton, an error message. However, for the specification to be declarative and robust, we cannot refer to automaton states by their internal numbers. The only robust way of referring to them is via sample invalid sentences, which, thanks to the reference interpreter, can be turned into states. In summary, the idea is to let the user provide a set of pairs of an invalid sentence and an error message, and build this information into the error actions of the automaton.

The tool will check that the user-supplied sentences are indeed invalid, that no two sentences map to the same automaton state, and (possibly) that the collection of all user-supplied sentences is exhaustive, in the sense that it covers all states in which an error can be detected. It will provide a facility for generating an exhaustive set of invalid sentences.

Above, we have suggested associating error messages with states. In reality, we will probably want to associate error messages with pairs of a state and a lookahead token.

An error message could be just a string, or, more generally, could be an ocaml semantic action that does not terminate normally: that is, it must raise an exception or terminate the process.

Testing The back-end of a parser generator must be tested at two levels. First, one must check that it is able to compile a (large) number of (well-formed) grammar specifications. Then, one must check, for a number of specific grammars, that the generated parser behaves correctly with respect to a (large) number of (well-formed or ill-formed) input sentences.

The first level of testing is already implemented in the Menhir source code repository: the parser generator is tested against roughly 150 well-formed grammar specifications. We plan to re-use this existing test suite.

The second level of testing is not currently implemented. Defining a test suite at this level will be part of the project. Ensuring a good level of coverage is a difficult question: how does one guarantee that the chosen grammar and input sentences exercise every aspect of the table generator and of the

table interpreter? This will require some thought. One could wish to implement and use a generator of random, well-formed grammatical sentences.

Documentation The user documentation will be extended to document the reference interpreter, the new back-end (a few instructions for using it should be sufficient), and the new error reporting mechanism.

We will also augment the generic Makefile that ships with Menhir, and possibly update ocaml-build with support for the new compilation scheme.

A small technical documentation will be written in order to document the design and format of the parser tables. This could possibly take the form of comments within the code itself. The source code of Menhir is heavily commented, and we intend to keep up with this tradition.

Time frame The proposed time frame is 3 months (June, July, August). This could be decomposed in three stages of one month each, roughly as follows:

1. study the existing code base (including ocaml-yacc and Menhir) and the scientific literature; implement a first version of the reference interpreter; propose a detailed design of the new back-end;
2. implement the new back-end; run preliminary tests; propose a detailed design of the new error handling mechanism;
3. implement the error handling mechanism; strengthen Menhir's test suite; run extensive tests; produce documentation; polish everything.

It is not clear whether it is realistic to hope to implement all three items in this time frame. If, at some point, it appears that it is not, then we will limit the scope of the project to the first two items.

Deliverables The main four deliverables are: a reference interpreter; a table-based back-end; a declarative error handling mechanism; an extended test suite.

Platforms The work will be carried out under Linux. In principle, Menhir works on top of all of the platforms supported by ocaml, including Windows; however, no serious testing has been carried out yet.

License Menhir is currently distributed under the same license as ocaml. I (François Pottier) am open to suggestions as to better choices.

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] Peter Dencker, Karl Dürre, and Johannes Heuft. [Optimization of parser tables for portable compilers](#). *ACM Transactions on Programming Languages and Systems*, 6(4):546–572, 1984.
- [3] Dick Grune and Criel J. H. Jacobs. *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer Verlag, 2008.
- [4] Clinton L. Jeffery. *Merr User's Guide*, July 2002.
- [5] Clinton L. Jeffery. [Generating LR syntax error messages from examples](#). *ACM Transactions on Programming Languages and Systems*, 25(5):631–640, 2003.
- [6] Mark P. Jones. *jacc: Just Another Compiler Compiler for Java*, February 2004.

- [7] Satya Kiran Popuri. [Understanding C parsers generated by GNU Bison](#), September 2006.
- [8] Robert Endre Tarjan and Andrew Chi-Chih Yao. [Storing a sparse table](#). *Communications of the ACM*, 22(11):606–611, 1979.