TP2 : SLR-based Parser Generation with Lex-Yacc-alikes

Thibaut Benjamin[1] and Burkhart Wolff[2][0000−0002−9648−7663]

[1] LMF, Université Paris-Saclay, Paris, France
thibaut.benjamin@universite-paris-saclay.fr
[2] LMF, Université Paris-Saclay, Paris, France
wolff@lmf.cnrs.fr

Abstract. This TP gives an introduction to the use of SLR-based, Lex-Yacc-style parsing generators. Lex-Yacc style generators exist in many languages; the one used here is a MLLex/MLYacc port to polyml by Takayuki Goto distributed under a debian open-source licence (see `https://github.com/eldesh/mllex-polyml/tree/master`).
The corresponding TP starts with an introduction into the Lex-Yacc language supported and its integration into Isabelle. This integration has the advantage that no different installation procedures have to be applied; the installation is just the import of an Isabelle theory.
After a general introduction into Lex and Yacc specifications of concrete programming languages, we proceed by two larger exercises where an existing core setup has to be extended to a richer language and extended by a generator for a suitable abstract syntax.
The accompanying material can be found on the the following site: [2]

Keywords: Grammars, Parsers, Railroad Diagrams, Programming in SML

1 Using Isabelle Lex Yacc

1.1 Isabelle Lex Yacc - an Introduction

Isabelle Lex-Yacc is distributed as an Isabelle theory `LexYacc` in the folder `isabelle_lex-yacc/`. Installing it just means importing `LexYacc` from it. Additional samples/examples can be drawn from theories `Calc` (a simple calculator), `Fol` (a weird fragment of First-order Logic) and `Pascal` (a non-trivial fragment of the Pascal language). The handling follows just the standard rules of the Isabelle IDE; however, note that the leight-weight integration approach can not remedy the somewhat arkane error messages and syntax of the underlying MLLex/MLYacc. An incremental development approach, starting from existing sources to a more and more richer theory, is therefore recommended.

A general documentation of MLLex/MLYacc is `https://rogerprice.org/ML-Lex-Yacc-Guide/ML-Lex-Yacc-Guide.pdf`.

From the Isabelle side, after importing theory `LexYacc`, there is a new command of the format:

ml-lex-yacc *<spec−name>*
with-lex‹
<ML−Lex−specification>
›
and
with-yacc‹
<ML− Yacc−specification>
›

There is an option that, when set before the `ml_lex_yacc`-command via *declare* [[*lexyacc-verbose=true*]], produces some more output like this:

```
Running lex ...
Running yacc ...
Binding ...
See theory exports
```

When activating (mouse-click) on `theory exports`, the user may get access to a presentation of the underlying SLR-automaton in the output window.

1.2 The *ML-Lex - specification*

An ML-Lex specification has the general format:

```
ML-Lex user declarations (this is actually SML code!)
%%
ML-Lex definitions
%%
ML-Lex rules
```

Note that each section is separated from the others by a `%%` delimiter. Further details on `ML-Lex user declarations`, `ML-Lex definitions` and `ML-Lex rules` can be found in the sections 4.1, 4.2, 4.3 and following sections in `https://rogerprice.org/ML-Lex-Yacc-Guide/ML-Lex-Yacc-Guide.pdf`.

The ML environment underneath the Isabelle Platform comes actually in two flavors, which are considered as different languages with mutual import-export interfaces for clarity: there is the (pure polyml) *SML environment* and the *Isabelle/ML environment*, which are basically distinct wrt. base-libraries, IO-access and, last but not least, the presence or absence of an Isabelle macro-mechanism called Antiquotations.

Now, note that the `ML-Lex user declarations` are SML-code which is interpreted in the context as the SML-environment; if you want to extend the former before compiling the code generated by `ML-Lex`, you have to include this code via the `SML_file <file>` command. Exporting SML-definitions into Isabelle-ML (polyml) is done via `SML_export <...>`.

1.3 The *Yacc - specification (according MLYacc)*

An ML-Yacc specification consists of three parts, each of which is separated from the others by a `%%` delimiter. The general format is:

```
ML-Yacc user declarations (this is actually SML code!)
%%
ML-Yacc declarations
%%
ML-Yacc rules
```

Comments have the same lexical definition as they do in SML and can be placed in any ML-Yacc section. More details of the sections `ML-Yacc user declarations`, `ML-Yacc declarations` and `ML-Yacc rules` are described in more detail in sections 9.1 ff. in `https://rogerprice.org/ML-Lex-Yacc-Guide/ML-Lex-Yacc-Guide.pdf`.

They contain the declaration of terminal and non-terminal symbols, operator precedences and general production rules, as well as the linking to actions (which may be calculations in SML or just constructions of abstract syntax declared in SML-files earlier).

In contrast to the original lex-yacc implementations in C, ML-Lex and ML-Yacc are *statically typed* syntax specification languages. This means that complex inter-dependencies between the different components of a syntax specification (terminals were declared in ML-Yacc declarations, but must be taken into account in the ML-Lex part nevertheless, actions depend on types yielded by the abstract syntax used in ML-Lex, etcpp.). These interdependencies were resolved by ML-Lex and ML-Yacc by generating *parameterized structures* called *functors* that must be instantiated in the end. So, each `ml_lex_yacc`-command in Isabelle has to be followed by some glue-code, that fiddles the different pieces together[3]

It is instructive to consider the `Calc.thy`-example to get a glimpse on how this glue code works:

- `ml_lex_yacc "Calc" ...` generates an SML structure which must be imported to Isabelle/ML: **SML-export** ‹*structure LrParser = struct open LrParser end*›
- the command also generates the two functors `CalcLrValsFun` and `CalcLexFun` (which have to agree on the definitions of `Tokens`)

[3] Future versions of `ml_lex_yacc` should do this automatically ...

```
structure CalcLrVals =
  CalcLrValsFun(structure Token = LrParser.Token)

structure CalcLex =
  CalcLexFun(structure Tokens = CalcLrVals.Tokens)
```

– combining these via a generated functor `Join` gives us the structure:

```
structure CalcParser =
  Join(structure LrParser = LrParser
       structure ParserData = CalcLrVals.ParserData
       structure Lex = CalcLex)
```

... and we are utmost there. In the structure `CalcParser` provides basically the two functions `CalcParser.makeLexer` which provides an interface for linewise reads of lexer-tokens and `CalcParser.parse` the access to the actual generated parser (to be initialized correctly).

Let's return to the example used in TP1 to illustrate the lexical aspects:

```
function f(x : integer) return integer is
z : integer := -5;
-- on suppose l'existence de la variable y ci-dessous
begin x := y + z * 2;
x := -x-2;
if (x = y or z >= y) then return(x-123);
end;
```

The technology provided by Lex/Yacc is perfectly able to treat the lexical issues of this example faithfully. In another setting, a Lex/Yacc generated parser as the one presented here is able to parse a 20 kloc C-file (with its very complicated lexical conventions) in about 2 seconds inside Isabelle[1].

2 Tasks

The exercises below are a superset of the given exercises. Choose 2 out of the 4 for the "rendu".

2.1 Exo 3

Provide an unambiguous grammar for a restricted version of C language expressions. Consider the following operators, listed in order of increasing precedence:

1. = (binary, right-associative)

2. +, - (binary, left-associative)
3. *, / (binary, left-associative)
4. * (unary, dereferencing pointers)

The grammar must allow manipulation of identifiers, constants, and access to one-dimensional arrays (e.g., `*(p+1) = a[x*(2+y)][z+3] = **z*2`) and function calls. Not all aspects of a language can be expressed syntactically, but if a check can be performed at the syntactic level, it should be done at this stage.

text‹*HINT*:› *Start with the theory* ‹*Calc.thy*› *and reduce/increment it task by task.*

2.2 Exo 4

Complete the previous exercise to incorporate:

- equality and inequality operators (== and !=)
- comparison operators (<, <=,>,>=)
- logical connectives (&&,|| and !)
- unary + and − operators

while respecting the priorities and associativity of the C language.

text‹*HINT*:› *Start with the theory* ‹*Calc.thy*› *and reduce/increment it task by task.*

2.3 Exo 5

We consider the language of propositional logical formulas constructed using atomic propositions, hereafter symbolized by the lexical unit P, the binary operators ∧ (logical "and"), ∨ (logical "or"), ⇒ (implication), ⇔ (equivalence), and the unary operator ¬ (negation). Such a language can be described by the following grammar:

F ::= P | (F) | F *and* F | F *or* F | F *implies* F | F *equiv* F | *not* F

1. Give the syntax tree for the formula (p *and* q) *or not*r, where p, q, and r are instances of P.
2. Show that the grammar is ambiguous. Give an unambiguous grammar for the same language, but imposing the following precedence on the operators (a "<" indicates that the left-hand symbol has lower precedence than the right-hand one, an "=" indicates that they have the same precedence). Assume the binary operators are left-associative: ‹*equiv*› < ‹*implies*› < ‹*and*› = ‹*or*› < ‹*not*›
3. Give the syntax tree associated with the word p *and* q *or* g *implies not* r *and* p.
4. Give an unambiguous grammar that respects the precedence rules above but prohibits unparenthesized mixtures of *and* and *or*, as well as any associativity with *implies* and *equiv*. The operators *and* and ∨ remain left-associative.

5. Present the SLR automaton that Yacc generates.

The following words must be invalid: p *and* q *or* r *and* g, p *implies* q *implies* r, p *equiv* q *equiv* r
The following words are valid: p *and* (q *or* r)*and* g, p *or* q *or* r, p *implies* (q *implies* r), (p *equiv* q) *equiv* r .

HINT: Start with the theory *Fol.thy* and reduce/increment it task by task.

2.4 Exo 6

Tasks:

- Develop an Abstract Syntax for a fragment of the Pascal language.
- Export is to ML.
- Define the action-bindings in the ML-Yacc Rules.
- Provide a reasonable test-set for your implementation.

text‹*HINT*:› *Start with the theory* ‹*Pascal.thy*› *and reduce/increment it task by task.*

3 Hints for the Rendu

Write a little report (result: .pdf) containing the solution elements of Exo 3 to Exo 6. You may use screenshots to document your code and system responses. You may work on the entire TP as a 'binome'. Send in the solution **today per mail** to your responsable TP.

Acknowledgement: Prof. Achim Brucker (Univ. Exeter, GB) provided substantial help for the integration of ML-Lex and ML-Yacc into Isabelle. See also the corresponding AFP entry.

end

References

1. Tuong, F., Wolff, B.: Deeply integrating C11 code support into isabelle/pide. In: Monahan, R., Prevosto, V., Proença, J. (eds.) Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019. EPTCS, vol. 310, pp. 13–28 (2019). https://doi.org/10.4204/EPTCS.310.3, https://doi.org/10.4204/EPTCS.310.3
2. Wolff, B.: Teaching Website: PolyTech Compiler Course. https://usr.lmf.cnrs.fr/~wolff/teach-material/2025-2026/ET4-Compil/index.html (2025), [Online; accessed 8-Dec-2025]