ML-Lex and ML-Yacc for Isabelle

Achim D. Brucker
Department of Computer Science
University of Exeter
Exeter, UK
a.brucker@exeter.ac.uk

Burkhart Wolff
Université Paris-Saclay
LMF
Paris, France
wolff@lmf.cnrs.fr

Abstract

Developing concrete syntax and parsers for Domain-Specific Languages (DSLs) within interactive theorem provers, such as Isabelle, is a common task. Isabelle supports this through rich, yet often brittle to configure, mixfix annotations and syntax translations. Moreover, Isabelle provides parser combinators that can be used to build parsers for the content of cartouches. An alternative to the latter are traditional parser generators, e.g., Lex and Yacc, that take a formal grammar description as input and generate a parser.

Traditionally, the use of parser generators for defining DSLs in Isabelle relies on invoking the lexer and parser as external tools, often modifying the generated source code, and importing the generated files. This requires users to manage complex external preprocessing toolchains.

In this AFP entry, we address this challenge by providing a native integration of standard ML-Lex and ML-Yacc into Isabelle/HOL. In more detail, we introduce an Isar-level interface, **ml_lex_yacc**, that allows users to write lexical and grammatical specifications directly within theory files. The framework compiles these specifications into Standard ML structures on-the-fly, links them, and loads them into the current theory context. Moreover, the integration automatically hooks into Isabelle's Prover IDE (PIDE), empowering users to provide real-time syntax highlighting, semantic tooltips, and precise error localization for their custom languages.

Key words: Isabelle/ML, Isabelle/PIDE, ML-Lex, ML-Yacc, Parser Generators

Contents

1 Introduction

Developing concrete syntax and, hence, parsers for Domain-Specific Languages (DSLs) within interactive theorem provers, such as Isabelle, is a common task. Isabelle supports this already through various mechanisms such as:

- Isabelle's mixfix annotations [15] that allow us to define custom concrete syntax for constants. For instance, if we use the **definition** command to introduce a new constant, we can immediately assign a syntactic representation to it:

  definition *xor* :: *bool* ⇒ *bool* ⇒ *bool* (**infixl** ⊕ *60*) **where**
  xor A B ≡ (*A* ∧ ¬ *B*) ∨ (¬ *A* ∧ *B*)

- Isabelle's syntax translations [15] provide support syntactic rewrite rules on term-representations, e.g.:

  syntax
  list :: *args* ⇒ *'a list* (‹(‹*indent=1 notation=‹mixfix list enumeration›*›[])›)
  syntax_consts *_list* ⇌ *Cons*
  translations [*x*, *xs*] ⇌ *x*#[*xs*]
  [*x*] ⇌ *x*#[]

- Isabelle's native ML-structures `Scan` and `Parse` implement functional parser combinators [15]. They are highly composable, naturally handle context-dependent grammars, and allow dynamic parser construction.

These mechanisms have the advantage that they are deeply integrated into the PIDE Framework [13] in general and, in particular, Isabelle/jEdit [14]. On the downside, Isabelle's mixfix notation and syntax translation can be brittle to configure and, moreover, share the "syntactic universe" with HOL, i.e., the syntax of the DSL needs to be disjoint from the already defined HOL syntax or one needs to overload syntactic constants (which can be brittle in itself). While parser combinators are powerful and flexible, they can, sometimes, be slow. In contrast, traditional lexer and parser generators, such as ML-Lex [1] and ML-Yacc [10], trade the flexibility of parser combinators for efficiency.

Using ML-Lex and ML-Yacc in the context of Isabelle is nothing new. Prominent examples include:

- The *TPTP Parser* (part of the Isabelle distribution) [9], which is used for parsing the Thousands of Problems for Theorem Provers (TPTP) format.

This is a critical piece of infrastructure that allows Isabelle's automated reasoning tools to read and write problem files. Technically, the TPTP Parser is built by executing ML-Lex and ML-Yacc manually as external tools, manually adapting the generated files, which are then loaded via the standard **ML_file**. A similar approach is also used by the AFP entry *Automated Stateful Protocol Verification (PSPSP)* [4, 5], which uses ML-Lex and ML-Yacc generated parsing for security protocol specifications and their (often very large) fixed-points.

- *AutoCorres2* [2], which uses ML-Lex and ML-Yacc generated parsing for translating C code into the Simpl [8] language for the verification of low-level systems code (like the seL4 microkernel [6]). Technically, the C parsing component of AutoCorres also provides an integration of ML-Lex and ML-Yacc providing the commands *mllex* and *mlyacc*. These commands read the lex and yacc specifications from an external file and invoke ml-lex and ml-yacc on them, which also generate external files on the physical file system. The generated files are modified "on-the-fly" using the sed utility. This requires write access to the directory in which the files of the AutoCorres2 AFP entry are stored and, moreover, external file operations in Isabelle need to be implemented carefully to avoid race conditions during concurrent operations.

- *Isabelle/C* [11, 12] provides a deep integration of C11 syntax into the Isabelle/PIDE document model, allowing for semantic annotations (comments containing HOL assertions) directly inside C code. Technically, it relies on ML-Lex and ML-Yacc style grammars (originally developed for Happy [7], a parser generator for Haskell) that are heavily modified (and, therefore likely hard to maintain) to support syntax-highlighting and semantic annotations.

All of these integrations are focused on one specific use case and, except for Isabelle/C, do not support the error reporting, syntax-highlighting, and annotation infrastructure provided by PIDE in general and, in particular, Isabelle/jEdit. In contrast, we provide a highly reusable native integration of ML-Lex [1] and ML-Yacc [10] into Isabelle/HOL.[1]

From an end-user perspective, we provide a new Isar command **ml_lex_yacc**, that allows users to write lexical and grammatical specifications directly within theory files. Generated lexers and parsers can support syntax highlighting and

[1] We started our work with the port of ML-Yacc and ML-Lex to Poly/ML, which are available at https://github.com/eldesh/mllex-polyml and https://github.com/eldesh/mlyacc-polyml.

PIDE-style error reporting. The generated parser is directly reflected into the current theory context. This allows, for instance, using ML-Lex/ML-Yacc parsers as first-class front ends for deeply embedded languages or within Isabelle/ML for the development of backend tools (see Section 5 for an overview of the examples provided as part of this AFP entry). Moreover, compared to standard ML-Lex and ML-Yacc, the generated parsers support, out-of-the-box, syntax-highlighting, PIDE-style error reporting, and annotations within PIDE. Moreover, in most applications, the code for linking the lexer and parser is also automatically generated.

From an Isabelle system-developer's perspective, we modified ML-Lex and ML-Yacc to work "in memory", i.e., avoiding the need for access to the physical file system completely and, therefore, minimizing the risk of race conditions. Furthermore, we provide an extended library that provides support for Isabelle's `Position.T` type and also supports the generation of code linking the lexer and the parser.

Notably, the integration of ML-Lex and ML-Yacc into Isabelle has been used in teaching a compiler construction module at Université Paris-Saclay/PolyTech [3]. Moreover, the development version of PSPSP [4, 5] has already been ported to this integrated framework.

The rest of the manual is structured as follows: we start by showing how to implement a simple calculator (the "Hello World" example of parsing) in Section 2. We then explain the new **ml_lex_yacc** in detail (Section 3), followed by revising the calculator in "expert mode" (Section 4). Finally, we provide an overview of the examples that are part of this AFP entry (Section 5).

2 First Steps: A Simple Calculator

In this section, we present the "Hello World!" of parser generators: a calculator. This example is taken directly from the ML-Lex/Yacc manuals [1, 10]. Hence, we recommend consulting them while working through this manual.

2.1 The Lex/Yacc Specification

Let us start with a brief recap of the high-level structure of ML-Lex and ML-Yacc specifications. Each of them consists of three parts: user declarations, definitions, and rules. When ML-Lex and ML-Yacc specifications are stored in external files, these three parts are separated by %%, i.e., a typical ML-Lex specification file is structured as follows:

```
ML-Lex user declarations
%%
ML-Lex definitions
%%
ML-Lex rules
```

ML-Yacc files are structured similarly. We put every part into their own cartouche, as part of a new Isar command **ml_lex_yacc**:

```
ml_lex_yacc Calc where
 lex_user_declarations‹›
 lex_definitions‹
  alpha=[A-Za-z];
  digit=[0-9];
  ws = [\ \t\r];
 ›
 lex_rules‹
  \n      => (lex());
  {ws}+   => (lex());

   {digit}+ => (tok_val (yypos, yytext, Markup.numeral, NUM, , Tokens.NUM, valOf
(Int.fromString yytext)));

  +     => (tok (yypos, yytext, Markup.keyword2, PLUS, , Tokens.PLUS));
  ;     => (tok (yypos, yytext, Markup.delimiter, SEMI, , Tokens.SEMI));

  .       => (lex());
 ›
 and yacc_user_declarations‹
  fun lookup bogus = 10000
    | lookup s = 0
 ›
 yacc_definitions‹
  %eop EOF SEMI

  %left PLUS

  %term NUM of int | PLUS | SEMI | EOF
  %nonterm EXP of int | START of int option

  %noshift EOF
 ›
 yacc_rules‹
  START : EXP (SOME EXP)
```

```
    | (NONE)
  EXP : NUM          (NUM)
    | EXP PLUS EXP   (EXP1+EXP2)
›
```

Inside your *lex_rules* section, you should use the globally provided `tok` and `tok_val` functions to emit tokens. These functions automatically register Isabelle markups for syntax highlighting:

- The function `tok` is used for valueless tokens (like keywords and symbols):

  ```
  tok: int * string * Markup.T * string * string * (Position.T
  * Position.T -> 'a) -> 'a
  ```

  This function takes six arguments. The first two are the *yypos* and *yytext* variables provided by the ML-Lex/Yacc tools. The next three arguments are PIDE-specific:

  - the markup that should be used in syntax highlighting (as defined in the ML structure `Markup`, e.g., `Markup.keyword1`).
  - the information that should be displayed in the "type" part of the PIDE tooltip.
  - the information that should be displayed in the "sort" part of the PIDE tooltip.

  The final argument is the token parsed. For example:

  tok (*yypos*, *yytext*, *Markup.keyword2*, *Type Hint*, *Sort Hint*, *Tokens.PLUS*)

- The function `tok_val` is used for tokens that carry semantic values (like integers or identifiers):

  ```
  tok_val : int * string * Markup.T * string * string * ('a *
  Position.T * Position.T -> 'b) * 'a -> 'b
  ```

  It takes the same parameters (in the same order) as the `tok` function plus one additional argument, the value. For example:

  tok_val (*yypos*, *yytext*, *Markup.numeral*, *NUM*, , *Tokens.NUM*, *valOf* (*Int.fromString yytext*))

Furthermore, the ML-Lex/Yacc for Isabelle framework automatically:

1. Injects standard aliases and PIDE-reporting functions (like `Isabelle_lex_yacc.tok` and `Isabelle_lex_yacc.tok_val`) via the `Isabelle_lex_yacc` environment (as discussed previously).

2. Provides the standard eof token definition.

3. Generates the ML-Lex *%header* and ML-Yacc *%name* and *%pos* directives (which should be omitted).

4. Automatically applies the Join functor to fuse the Lexer and Parser components.

5. Exposes a unified structure, based on the name provided to the **ml_lex_yacc** command.

In this "standard mode", the aim is to minimize boilerplate and handle the tricky wiring of the lexer, parser, and Isabelle PIDE infrastructure automatically.

2.2 The Generated ML API

The **ml_lex_yacc** generates the lexer and parser, as well as the necessary glue code for linking them. Ultimately, it produces a unified ML structure named after your parser (e.g., `Calc`). The most important automatically generated function is:

```
Calc.parse_source: Proof.context -> Input.source -> int option
```

where the return type (here: `int option`) matches the *START* nonterminal.

You can use it directly in a custom Isar command:

```
ML ‹
  fun run_calc source thy =
    let
      val ctxt = Proof_Context.init_global thy
      val res = Calc.parse_source ctxt source
      val _ = writeln (case res of SOME v => Int.toString v
                                 | NONE => No result)
    in thy end

  val _ = Outer_Syntax.command @{command_keyword simple_calc}
        A simple inline calculator
        (Parse.input Parse.cartouche >> (fn source =>
Toplevel.theory (run_calc source)))
```

›

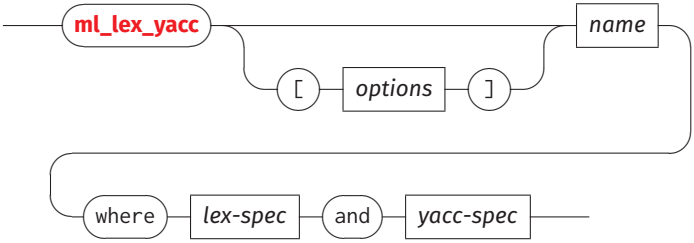
We can now run simple calculations directly in Isabelle:

simple_calc‹21+21› — *Prints 42 in Isabelle's Output panel*

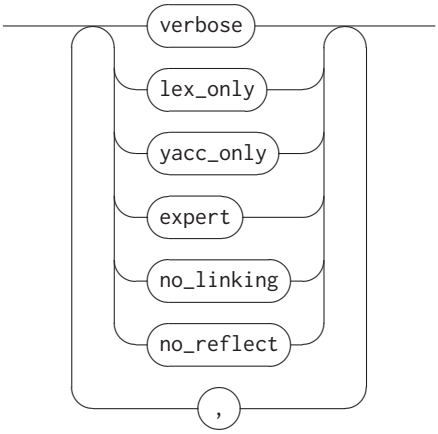
Note that the theory *Isabelle_Lex—Yacc.Calc* contains an extended version of this simple calculator.

3 Defining Lex/Yacc Specifications

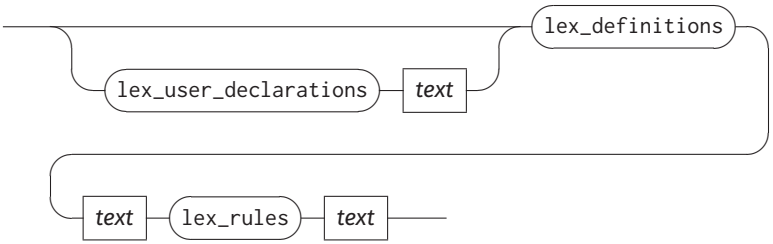
The theory *Isabelle_Lex—Yacc.LexYacc* (which also is the main entry point for the Isabelle Lex/Yacc framework) provides **ml_lex_yacc** command provides an integrated, Isar-level interface for defining and generating Standard ML parsers using ML-Lex and ML-Yacc directly within Isabelle theories. It processes lexical and grammatical specifications, compiles them into SML structures, and loads them into the current Isabelle theory context. Its general syntax is as follows:



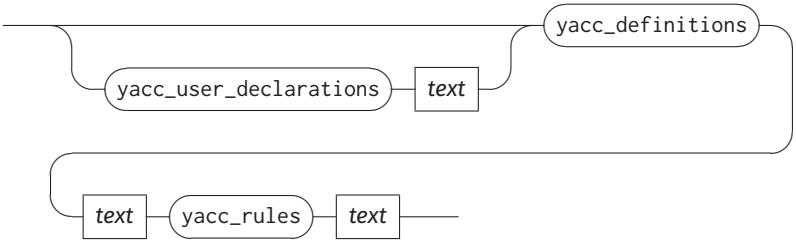
options



lex-spec



yacc-spec



- *name*: Specifies the name of the parser. By default, it is also used as a prefix for the generated SML structures. For example, providing the name "MyLang" will generate underlying ML structures like *MyLangLex*, *MyLangLrVals*, and a unified *MyLangParser*. In expert mode (see below),

the SML structures will be named based on the lex and yacc directives that are part of the lex and yacc specifications (e.g., *%name*).

- *options*:
  - *verbose*: Instructs the underlying ML-Yacc/ML-Lex generator to output verbose information. Generated artifacts (such as SML code or the automaton description) are stored for inspection in Isabelle's virtual file system.
  - *lex_only*: Only process the ML-Lex part of the specification. Useful during development for faster turn-around times. This implies *no_linking* and *no_reflect*.
  - *yacc_only*: Only process the ML-Yacc part of the specification. Useful during development for faster turn-around times. This implies *no_linking* and *no_reflect*.
  - *expert*: Enables advanced/expert mode where the specified lex and yacc specifications are passed unmodified to ML-Lex and ML-Yacc (except adding the %% separators between the three blocks of each specification). Furthermore, automated linking is disabled in expert mode.
  - *no_linking*: Skips the automatic generation of the boilerplate "linking" structure (the code that normally joins the Lexer, ParserData, and LrVals together). This is useful if you intend to manually wire the generated ML functor blocks together later.
  - *no_reflect*: Does not reflect the generated code into the Isabelle/ML environment. This implies *no_linking*.
- The lex specification is broken into three parts. In the original lex specification, these parts are separated by %% (which should be omitted here). Furthermore, by default no directives specifying functors or names should be included, as they break the automated linking:
  - *lex_user_declarations* (optional): An embedded ML source block containing user-level SML code (e.g., token type aliases, state variables, or helper functions).
  - *lex_definitions*: The ML-Lex definitions, such as regular expression macros (e.g., *alpha*=[*A*–*Za*–*z*];) and lexer state declarations (e.g., *%s COMMENT*;). Only in expert mode, a *%header* declaration should be provided; i.e., when porting an ML-Lex specification, remove the *%header* declaration.

- *lex_rules*: The ML-Lex scanning rules and their corresponding SML semantic actions. The structure `Isabelle_lex_yacc` provides the utility functions `tok` and `tok_val`.

  * The function `tok` is used for valueless tokens (like keywords and symbols):

    ```
    tok: int * string * Markup.T * string * string *
    (Position.T * Position.T -> 'a) -> 'a
    ```

    This function takes six arguments. The first two are the *yypos* and *yytext* variables provided by the ML-Lex/Yacc tools. The next three arguments are PIDE-specific:

    · the markup that should be used in syntax highlighting (as defined in the ML structure `Markup`, e.g., `Markup.keyword1`).

    · the information that should be displayed in the "type" part of the PIDE tooltip.

    · the information that should be displayed in the "sort" part of the PIDE tooltip.

    The final argument is the token parsed. For example:

    tok (*yypos*, *yytext*, *Markup.keyword2*, *Type Hint*, *Sort Hint*, *Tokens.PLUS*)

  * The function `tok_val` is used for tokens that carry semantic values (like integers or identifiers):

    ```
    tok_val : int * string * Markup.T * string * string
    * ('a * Position.T * Position.T -> 'b) * 'a -> 'b
    ```

    : It takes the same parameters (in the same order) as the `tok` function plus one additional argument, the value. For example:

    tok_val (*yypos*, *yytext*, *Markup.numeral*, *NUM*, , *Tokens.NUM*, *valOf* (*Int.fromString yytext*))

    Note that in expert mode, these functions need to be called using their fully qualified name (e.g., `Isabelle_lex_yacc.tok`) or, alternatively, the *lex_user_declarations* needs to include an *open Isabelle_lex_yacc* statement.

- The yacc specification is broken into three parts. In the original yacc specification, these parts are separated by %% (which should be omitted here). Furthermore, by default no directives specifying functors or names should be included, as they break the automated linking:
  - *yacc_user_declarations* (optional): An embedded ML source block containing user-level SML code to be injected at the top of the generated parser. Useful for defining custom datatypes or helper functions used in semantic actions.
  - *yacc_definitions*: A cartouche containing ML-Yacc definitions, including %*term* and %*nonterm* declarations, associativity, and start symbols. Only in expert mode, the %*name* and %*pos* declarations should be provided; i.e., when porting an ML-Yacc specification, remove the %*name* and %*pos* declarations.
  - *yacc_rules*: A cartouche containing the ML-Yacc grammar productions (BNF format) and their corresponding SML semantic actions.

Note that the generated code is reflected into the SML environment of Isabelle. Hence, if your specification uses ML code defined within an **ML**-environment (i.e., Isabelle/ML), it needs to be imported into SML using the **SML_import** command. For example:

SML_import ‹*structure Datalog_AST = Datalog_AST*›

The theory *Isabelle_Lex_Yacc.Datalog* contains an example of using ML code defined within an **ML**-environment.

4 Expert Mode: The Calculator Example Revisited

When you invoke **ml_lex_yacc** with the [*expert*] option, the integration backs off and expects you to write a fully compliant, standalone ML-Lex/ML-Yacc specification. In particular:

1. No Hidden Inclusions: You must declare *Tokens*, *lexresult*, *pos*, and *svalue* manually in *lex_user_declarations*.
2. No Auto-Directives: You must manually provide the %*header* directive for Lex and the %*name* and %*pos* directives for Yacc.

3. No Auto-Linking: The framework will compile the *.lex.sml* and *.grm.sml* files into the ML environment, but it will **not** generate the final parser structure or the *parse_source* function. You must write the Join functor application yourself.

4. No Default PIDE hooks: The `Isabelle_lex_yacc.tok` and `Isabelle_lex_yacc.tok_val` helpers from `Isabelle_lex_yacc` are not automatically imported. If you want syntax highlighting, you must implement the position lookup and report logic yourself.

   This mode allows for easily using already existing ML-Lex and ML-Yacc specifications, as it is only necessary to split them up into their three parts (and omitting the %% separators).

   Expert Mode is recommended when you have highly customized lexer states, require bespoke error-correction strategies directly within the Lexer, or are porting legacy SML codebases where the automatic Isabelle-specific wrappers conflict with existing user declarations.

   The theory *Isabelle_Lex—Yacc.CalcExpert* provides a complete example of the calculator in expert mode.

4.1 Lex and Yacc Specification

The following is a simplified version. In direct comparison to the standard use cases (e.g., recall Section 2), notice the explicit declarations that (in this example) provide similar functionality to standard mode (e.g., position computations using the type `Position.T`).

```
ml_lex_yacc [expert] CalcExpert where
lex_user_declarations‹
  structure Tokens = Tokens
  type pos = Position.T
  type svalue = Tokens.svalue
  type ('a,'b) token = ('a,'b) Tokens.token
  type lexresult = (svalue, pos) token

  fun eof () = Tokens.EOF(Position.none, Position.none)
  fun error' (e, p: Position.T, _) = ()
  (*
    You must implement your own 'tok' and 'tok_val' equivalent methods or
    use the Isabelle_lex_yacc provided functions manually, if you want PIDE support here
```

```
 *)
 ›
lex_definitions‹
 %header (functor CalcExpertLexFun(structure Tokens: CalcExpert_TOKENS));
 digit=[0–9];
›
lex_rules‹
     {digit}+  =>  (Tokens.NUM(valOf  (Int.fromString  yytext),  Position.none,
Position.none));
 +    => (Tokens.PLUS(Position.none, Position.none));
 ;    => (Tokens.SEMI(Position.none, Position.none));
 .     => (lex());
›
and yacc_definitions‹
 %name CalcExpert
 %pos Position.T
 %eop EOF SEMI

 %left PLUS
 %term NUM of int | PLUS | SEMI | EOF
 %nonterm EXP of int | START of int option
 %noshift EOF
›
yacc_rules‹
 START : EXP (SOME EXP)
     | (NONE)
 EXP : NUM          (NUM)
    | EXP PLUS EXP    (EXP1+EXP2)
›
```

4.2 Linking Expert Mode Output

Because Expert Mode disables auto-linking, you must manually assemble the parser in an ML block. This requires using the ML-Yacc Join functor:

```
ML ‹
structure CalcExpertParser =
struct
  structure CalcExpertLrVals =
    CalcExpertLrValsFun(structure Token = LrParser.Token)

  structure CalcExpertLex =
    CalcExpertLexFun(structure Tokens = CalcExpertLrVals.Tokens)
```

```
  structure Parser =
    Join(structure LrParser = LrParser
         structure ParserData = CalcExpertLrVals.ParserData
         structure Lex = CalcExpertLex)

  (* In a real implementation, you would write your own
     `parse_source` here, handling the Lexer initialization
      and the `Stream.get` loop. *)
end
›
```

5 Examples

To demonstrate the versatility of our Lex/Yacc framework for Isabelle, we provide several examples (see Figure 1 for the session graph, listing all provided examples, i.e., the direct predecessors of *Isabelle_Lex—Yacc.Examples*):

- The theory *Isabelle_Lex—Yacc.Calc* provides an example of a simple arithmetic expression evaluator (calculator) in standard mode. This example is a port of the calculator example provided by the ML-Yacc distribution.
- The theory *Isabelle_Lex—Yacc.CalcExpert* provides an example of a simple arithmetic expression evaluator (calculator) in expert mode. This example is a port of the calculator example provided by the ML-Yacc distribution.
- The theory *Isabelle_Lex—Yacc.Pascal* provides a simple parser for the Pascal programming language. This example is a port of the Pascal example provided by the ML-Yacc distribution.
- The theory *Isabelle_Lex—Yacc.Datalog* provides an example of a Lex/Yacc parser used as a front-end for a language (Datalog) that is deeply embedded into Isabelle/HOL. It shows how
  - to interact with ML code (here: the ML datatype defining the abstract syntax tree of Datalog) defined in an **ML**-environment during parsing,
  - how to translate the parsed expression into HOL-terms, and
  - how to define an Isar command that parses Datalog and binds the generated HOL-term to a logical constant (i.e., a domain-specific definition command).

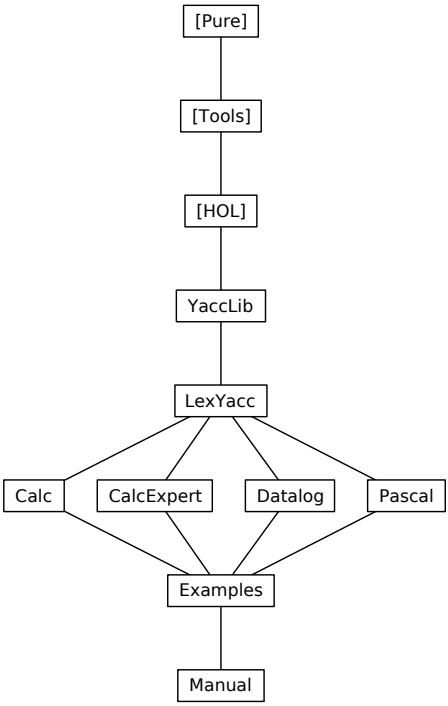


Figure 1: The Dependency Graph of the Isabelle Theories.

References

[1] A. W. Appel, J. S. Mattson, and D. R. Tarditi. *A Lexical Analyzer Generator for Standard ML*. Version 1.5.0. Princeton University. 1994. URL: https://www.smlnj.org/doc/ML-Lex/manual.html.

[2] M. Brecknell, D. Greenaway, J. Hölzl, F. Immler, G. Klein, R. Kolanski, J. Lim, M. Norrish, N. Schirmer, S. Sickert, T. Sewell, H. Tuch, and S. Wimmer. AutoCorres2. *Archive of Formal Proofs*, 2024. ISSN: 2150-914x. https://isa-afp.org/entries/AutoCorres2.html, Formal proof development.

[3] A. D. Brucker and B. Wolff. Isabelle in a compiler construction class: conception, tooling, and experiences. In M. Wenzel, editor, *Isabelle Workshop*, 2026.

[4] A. V. Hess, S. Mödersheim, A. D. Brucker, and A. Schlichtkrull. Automated stateful protocol verification. *Archive of Formal Proofs*, Apr. 8, 2020. ISSN: 2150-914x. https://www.isa-afp.org/entries/Automated_Stateful_Protocol_Verification.html, Formal proof development.

[5] A. V. Hess, S. A. Mödersheim, A. D. Brucker, and A. Schlichtkrull. PSPSP: a tool for automated verification of stateful protocols in Isabelle/HOL. USenglish. *Journal of Computer Security*, 33, 6, 2025. DOI: 10.1177/0926227X251358741.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, Big Sky, Montana, USA. Association for Computing Machinery, 2009. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[7] S. Marlow and A. Gill. Happy: the parser generator for Haskell, 1997. https://www.haskell.org/happy/.

[8] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, 2008. ISSN: 2150-914x. https://isa-afp.org/entries/Simpl.html, Formal proof development.

[9] N. Sultana. The Isabelle/TPTP parser, 2013. Part of the Isabelle distribution (`src/HOL/TPTP/`). https://isabelle.in.tum.de/.

[10] D. R. Tarditi and A. W. Appel. *ML-Yacc User's Manual*. Version 2.4. Princeton University. 2000. URL: https://www.smlnj.org/doc/ML-Yacc/.

[11] F. Tuong and B. Wolff. Deeply integrating C11 code support into Isabelle/PIDE. In R. Monahan, V. Prevosto, and J. Proença, editors, *Workshop on Formal Integrated Development Environment (F-IDE@FM)*, EPTCS, pages 13–28, 2019. DOI: 10.4204/EPTCS.310.3.

[12] F. Tuong and B. Wolff. Isabelle/C. *Archive of Formal Proofs*, 2019. ISSN: 2150-914x. https://isa-afp.org/entries/Isabelle_C.html, Formal proof development.

[13] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *Interactive Theorem Proving (ITP 2014)*, volume 8558 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2014. DOI: 10.1007/978-3-319-08970-6_33.

[14] M. Wenzel. *Isabelle/jEdit*. Part of the Isabelle distribution. Technische Universität München. URL: https://isabelle.in.tum.de/doc/jedit.pdf.

[15] M. Wenzel. *The Isabelle/Isar Reference Manual*. Part of the Isabelle distribution. Technische Universität München. URL: https://isabelle.in.tum.de/doc/isar-ref.pdf.