TP1 : An Introduction to Compiler Construction

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

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

(TP NOTE)

Abstract. This TP gives an introduction to the concepts of parsing. The corresponding TP starts with an introduction to functional programming, and introduces to a specific parsing technique called *combinator parsing* which is particulary simple to implement in a functional language.
The accompanying material can be found on the the following site: [1]

Keywords: Programming in SML, Parsers, Comboinaor Parsing

1 Revision : Functional Programming in SML

1.1 Revision from the TD

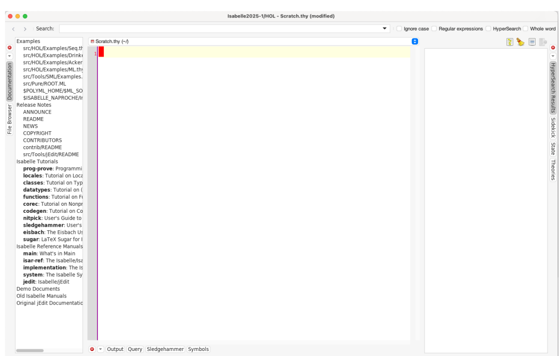

Fig. 1: The global Isabelle Window.

We will use Isabelle as an advanced IDE for functional programming in SML. After starting Isabelle in a shell by:

– `isabelle jedit <thyname>.thy`

(where `isabelle` is set to be the alias to your Isabelle2025-2 installation) a window should appear as shown in Fig. 1.

The central window is the editing window; by klicking on the "output"-button one gets additionally another window that shows the evaluation results corresponding to the cursor in the editing window.

Inserting the setup `theory Scratch imports Main begin` brings the system into a mode where you can start the `ML " ... "` (or `ML` ‹› ... ‹›)sub-environment that we will use. This will look like Fig. 2

You may open up `ML` one after the other to separate them; and also put your SML snippets in a file (like "tp1.sml") and include this inside Isabelle by:

– `ML_file "tp1.sml"`

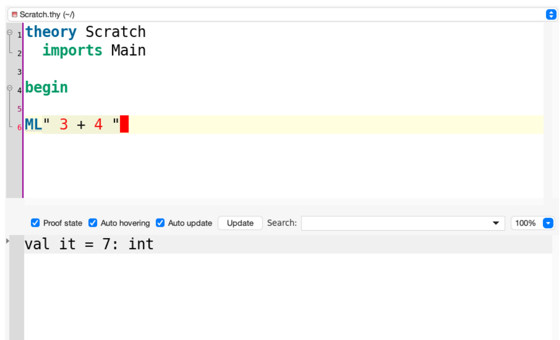

Fig. 2: The editing window with basic content.

Standard ML (SML) is a general-purpose, high-level, modular, functional programming language with compile-time type checking and type inference. It is popular for writing compilers, for programming language research, and for developing theorem provers.

A nice Reference/Tutorial is the following:

– `https://learnxinyminutes.com/standard-ml/`

1.2 Exo 1: Writing a simple Expression Evaluator

Write in SML a little evaluator for an expression language. The abstract symtax, i.e. the term language of expressions can be represented by:

$$\begin{aligned} \mathit{datatype}\ \mathit{expr} = {} & \mathit{const}\ \mathit{of}\ \mathit{int} \\ & \mid \mathit{var}\ \mathit{of}\ \mathit{string} \\ & \mid \mathit{mul}\ \mathit{of}\ (\mathit{expr} * \mathit{expr}) \\ & \mid \mathit{add}\ \mathit{of}\ (\mathit{expr} * \mathit{expr}) \end{aligned}$$

Load an SML-file belonging to the TP1:

ML-file‹*tp1.sml*›

It provides the abstract syntax of expressions and boolean expressions, and statements together with a dictionary (called *environment* `env`) derived from the library structure `Symtab.table`. An environment associates to a name (a string) a value, in our case an integer. If you open the sml file, you may hover over the use of `Symtab.table` and get the signature of this structure with operations such as `lookup` and `make`. Note that `Symtab.key` is just a synonym to ”string”.

Tasks:

1. Construct an environment where the variable `"a"` has the value *5* and `"b"` the value *10*.
2. Construct a function `eval_expr: expr -> env -> int` that assigns to an expression in abstract syntax its value.
3. Construct a function `eval: stmt -> env -> env` that assigns to a statement `stmt` in the abstract syntax its resulting environment. Hint: Here is a partial solution:

```
fun eval skip env = env
  | eval (ass(s,e)) env = (Symtab.update (s, eval-expr e env) (env))
  | eval (seq(s1,s2)) env = eval s2 (eval s1 env)
```

4. Test your evaluation with some small examples.

2 Combinator-based Parsing.

Parsing-Combinators are a popular technique to construct parsers of medium size and medium efficiency. Nowadays, implementations are common in many functional and modern imperative programming languages (such as Scala). Parsing combinators allow for *dynamic* syntax extensions in interpreter environments such as OCaml, SML, and ... Isabelle.

In SML, the concept is represented as follows: a *('b,'a)parser* is a function that 'grabs' a prefix list of elements from an input *'a list* and converts this into some value of type *'b* and the rest-list of type *'a list*. In case that the prefix does not match the language the parser is constructed for, the function may raise the exception `FAIL`.

The combination of parsers can be done particularly elegantly in a functional programming language. The definition of the sequential composition, the alternative and the piping is a classical example for the use of higher-order functions and its definition is is straight forward. We just show the types:

ML‹

```
type ('b, 'a) parser = 'a list -> 'b * 'a list

(*sequential pairing*)
val - = op -- : ('b,'a) parser * ('c,'a) parser -> ('b * 'c,'a) parser
(*alternative*)
```

val - = *op* || : ('b,'a) *parser* ∗ ('b,'a) *parser* −> ('b,'a) *parser*
(∗*piping*∗)
val - = *op* >> : ('b,'a) *parser* ∗ ('b −> 'd) −> ('d,'a) *parser*
›

In Isabelle/ML, these combinators were provided by the ML structure `Scan` — note that the types in its interface for the operators above are even more general as the one in this presentation. `Scan` provides a large collection of parsing combinators, among them the combinators for options and repetitions, and even a mechanism to construct relatively efficient scanners (which we will not use here; we will use a simplistic approach instead).

`fun rghr ( ert : b345 , trz : b123 )`

we assume for simplicity that all lexical tokens are clearly separated from each other by blanks, tabs or carriage returns.

2.1 Exo 2: Conceive a parser and implement it in SML.

Tasks:

1. Conceive a Syntax-Definition as Railroad Diagram for a language of function headers. The syntax should comprise examples such as:
   – *fun rghr* (*ert* : *b345* , *trz* : *b12*)
   – *fun rghr* () : *int*
   – *fun rghr* (*sc* : *b345* , *rt* : *b345* , *nn* : *b345*) : *bool*
   Note that we use blanks, tabs and carriage returns as separators between lexems.
2. Conceive a simple lexical analysis (Lexer). (Hint: use the function `String.tokens` from the SML library. Add an additional symbol for the end of file.
3. From the basic combinator for keywords $$: *string* −> (*string*,'a) *parser* construct a parser for your syntax definition and test it with a number of example: **Hint**: The basic parsers for a string can be defined by `val parse_int = Scan.some (Int.fromString)` resp. `val parse_string = Scan.some (SOME: string -> string option)`

3 Hints for the Rendu

Write a little report (result: .pdf) containing the solution elements of Exo 1 and Exo 2. 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.

References

1. 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]