Parsing Programming Languages

Lorenzo Ceragioli

November 6, 2024

IMT Lucca

Lexical Analysis: Scanning

From

$$\texttt{if } \textit{true} \texttt{ then } x \texttt{ := } y + 5 \texttt{ else } \textit{skip}$$

To

$$\texttt{IF}, \texttt{TRUE}, \texttt{THEN}, (\texttt{ID}, x), \texttt{ASSIGN}, (\texttt{ID}, y), \texttt{PLUS}, (\texttt{INT}, 5), \texttt{ELSE}, \texttt{SKIP}$$

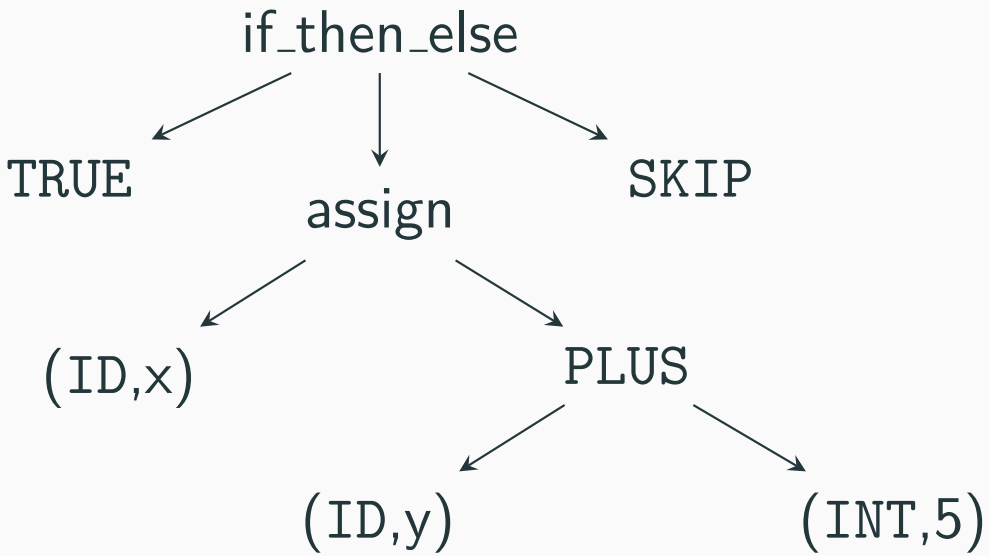
- Recognize lexemes (e.g. `:=`) and return tokens (e.g. `ASSIGN`)
- Some token contains the lexeme (or a function of it) as attribute, e.g. (`ID`, `x`) and (`INT`, 5)
- Also verifies that constructs are well written

Syntactic Analysis: Parsing

From

$\texttt{IF}, \texttt{TRUE}, \texttt{THEN}, (\texttt{ID}, x), \texttt{ASSIGN}, (\texttt{ID}, y), \texttt{PLUS}, (\texttt{INT}, 5), \texttt{ELSE}, \texttt{SKIP}$

To



Also verifies that programs are well constructed

Scanning with ocamllex

Implementing Scanners Automatically

Scanner generators

- **Input:**
  - a set of **tokens**
  - a list of **rules** associating Regular Expressions and tokens
- **Output:**
  - the **parser code**

ocamllex

A scanner generator for OCaml

- **Input:** `Mylexer.mll`
  - a set of **tokens** (will be defined externally)
  - some code to be used by the generated scanner
  - a list of **rules** mapping from Regular Expressions to tokens
- **Command:** `ocamllex Mylexer.mll`
- **Output:** `Mylexer.ml` and `Mylexer.mli`
  - the **scanner module**

ocamllex file details

```
1  (* code to be copied in the scanner module *)
2  {
3  open Parser  (* <-- where we define the tokens *)
4  exception LexingError of string
5  }
6
7  (* some named RExp *)
8  let somename = someRExp
9
10 (* lexing rules *)
11 rule read = parse
12 | RExp {token or command}
13 | RExp {token or command}
14 | somename {token or command}
15 ...
```

ocamllex Example: `mylexer.mll`

```
1  (* code to be copied in the scanner module *)
2  {
3  open Myparser  (* <-- where we define the tokens *)
4  exception LexingError of string
5  }
6
7  (* some named RExp *)
8  let integer = '-'?['0'-'9']['0'-'9']*
9  let white = [' ' '\t']+ | '\r' | '\n' | "\r\n"
10
11 (* lexing rules *)
12 rule read = parse
13 | white {read lexbuf}
14 | integer {INT(int_of_string (Lexing.lexeme lexbuf))}
15 | "+" {PLUS}
16 | "-" {MINUS}
17 | "*" {TIMES}
18 | eof {EOF}
19 | _ { raise (LexingError (Lexing.lexeme lexbuf)) }
```

ocamllex Example – details

```
1  (* lexing rules *)
2  rule read = parse
3  | white {read lexbuf}
4  | integer {INT(int_of_string (Lexing.lexeme lexbuf))}
5  | "+" {PLUS}
6  | "-" {MINUS}
7  | "*" {TIMES}
8  | eof {EOF}
9  | _ { raise (LexingError (Lexing.lexeme lexbuf)) }
```

- `read` – the name of the function from a buffer to tokens
- `read lexbuf` – a command saying to ignore the lexeme
- `Lexing.lexeme lexbuf` – the lexeme

Usage

```
1  let lexbuf = Lexing.from_channel in_file in
2  let mylexerfunction = Mylexer.read lexbuf in
3    ...
```

Parsing with menhir

Implementing Parsers Automatically

Parser generators

- **Input:**
  - associativity rules
  - grammar
- **Output:**
  - the **parser code**

menhir

An LR(1) parser generator for OCaml

- **Input:** `Myparser.mly`
  - some code to be used by the parser
  - a set of **tokens**
  - associativity rules
  - grammar
- **Command:** `menhir Myparser.mly`
- **Output:** `Myparser.ml` and `Myparser.mli`
  - the **parser module**

menhir file details

```
1  (* code to be copied in the scanner module *)
2  %{
3  open MyAST  (* <-- definition of the Abstract Syntax Tree *)
4  %}
5
6  (* tokens *)
7  %token TOKENNAME
8  %token <lexeme type> OTHERTOKENNAME
9
10 (* start nonterminal *)
11 %start <AST type> startnonterminal
12
13 %%  (* <-- this delimit the grammar rules *)
14
15 (* grammar *)
16 startnonterminal:
17   | case1 {resulting value (a part of the AST)}
18   ...
19   | casen {resulting value (a part of the AST)}
20 othernonterminal:
21   | ...
```

Example: `Aexp.ml`

The module `Aexp` defining our abstract syntax tree

```
 1  type aexp =
 2    | Intliteral of int
 3    | Plus of aexp * aexp
 4    | Minus of aexp * aexp
 5    | Times of aexp * aexp
 6
 7  let rec eval = function
 8    | Intliteral n -> n
 9    | Plus (a1, a2) -> (eval a1) + (eval a2)
10    | Minus (a1, a2) -> (eval a1) - (eval a2)
11    | Times (a1, a2) -> (eval a1) * (eval a2)
```

The interface `Aexp.mli` makes both definitions visible.

menhir Example: `Myparser.mly`

```
%{
  open Aexp (* <-- code copied in the scanner module *)
%}

(* tokens *)
%token <int> INT
%token PLUS MINUS TIMES EOF

(* start nonterminal *)
%start <aexp> prg

%%

(* grammar *)
prg:
    | t = trm; EOF                  {t}
trm:
    | i = INT                       {Intliteral i}
    | t1 = trm; PLUS; t2 = trm      {Plus (t1, t2)}
    | t1 = trm; MINUS; t2 = trm     {Minus (t1, t2)}
    | t1 = trm; TIMES; t2 = trm     {Times (t1, t2)}
```

menhir Example – details

- `%token <int> INT` – INT token holds the lexeme parsed as integer
- `%start <aexp> trm` – `trm` is the starting non-terminal, and the result of parsing the file will be a value of type `aexp` (defined in `Aexp.ml`)
- `t1 = trm; PLUS; t2 = trm Plus (t1, t2)` – while parsing something of the form `trm PLUS trm`, we take `t1` and `t2`, the result of parsing the two subterms, and we return an `aexp` value `Plus (t1, t2)`.

Usage: the interpreter

```
1    let in_file = open_in Sys.argv.(1) in
2    let lexbuf = Lexing.from_channel in_file in
3    let program = (Myparser.prg Mylexer.read lexbuf) in
4      ... something using eval(program)
```

Precedence and Ambiguity

Ops

The parser we generated is useless: no precedence is specified!

(indeed menhir complains about these problems)

Precedence

How to parse?

```
4 * 5 + 2
```

In general we want * to bind more strictly than +

Ambiguity in the Grammar — Precedence

$$\mathit{term} ::= \texttt{INT} \mid \mathit{term} + \mathit{term} \mid \mathit{term} - \mathit{term} \mid \mathit{term} * \mathit{term}$$

In general we want $*$ to bind more strictly than $+$

We can either update the grammar

$$\mathit{term} ::= \mathit{sterm} \mid \mathit{term} + \mathit{term} \mid \mathit{term} - \mathit{term}$$
$$\mathit{sterm} ::= \texttt{INT} \mid \mathit{sterm} * \mathit{sterm}$$

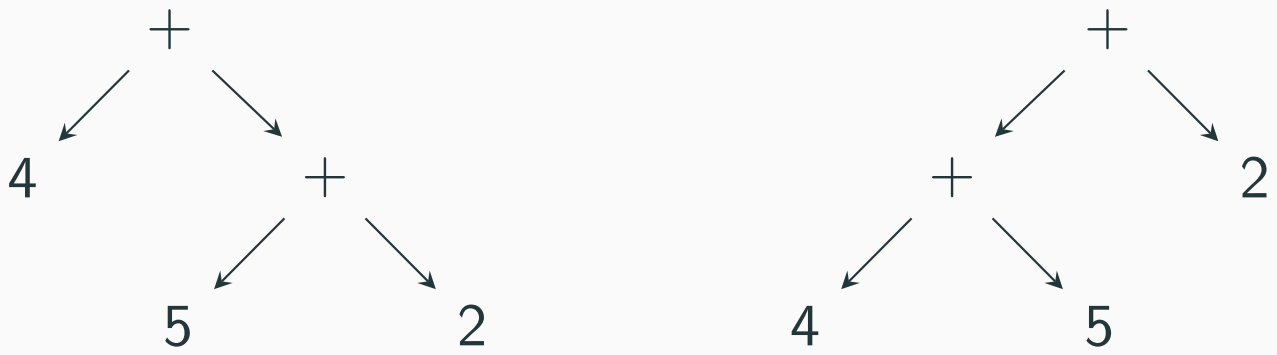
Or inform menhir to give higher precedence to `TIMES`

(we will see how)

Associativity

How to parse?

`4 + 5 + 2`



For plus it does not change the result, still menhir complains!

Notice that in other cases the associativity may alter the result

- With `f g x = f (g x)` we have `f` : $B \longrightarrow C$, `g` : $A \longrightarrow B$, and `x` : A
- With `f g x = (f g) x` we have `f` : $C \longrightarrow (A \longrightarrow B)$, `g` : C, and `x` : A

Ambiguity in the Grammar — Associativity

$$term ::= sterm \mid term + term \mid term - term$$

$$sterm ::= \texttt{INT} \mid sterm * sterm$$

We can either update the grammar

$$term ::= sterm \mid term + sterm \mid term - sterm$$

$$sterm ::= \texttt{INT} \mid sterm * \texttt{INT}$$

Or inform menhir to give higher precedence to `TIMES` **and** to associate left

(now we see how to do this)

Specifying Precedence and Associativity to `menhir`

```
1  (* code to be copied in the scanner module *)
2  %{
3  open MyAST  (* <-- definition of the Abstract Syntax Tree *)
4  %}
5
6  (* tokens *)
7  %token TOKENNAME
8  %token <lexeme type> OTHERTOKENNAME
9
10 (* start nonterminal *)
11 %start <AST type> nonterminal
12
13 (* associativity in order of precedence *)
14 %left/nonassoc/right precname or token
15 %left/nonassoc/right precname or token
16
17 %%
18
19 (* grammar *)
20 nonterminal:
21   | case1 %prec precname {resulting value}
22   ...
```

menhir Example: `Myparser.mll`

```
1  (* code to be copied in the scanner module *)
2  %{
3    open Aexp
4  %}
5
6  (* tokens *)
7  %token <int> INT
8  %token PLUS MINUS TIMES EOF
9
10 (* start nonterminal *)
11 %start <aexp> prg
12
13 (* associativity in order of precedence *)
14 %left PLUS MINUS  /* lowest precedence */
15 %left TIMES       /* highest precedence */
16
17 %%
18
19 (* grammar *)
20 prg:
21 ...
```

Details on Associativity and Precedence in `menhir`

```
1  (* associativity in order of precedence *)
2  %left TOKEN1 TOKEN2        /* lowest precedence */
3  %left TOKEN3 namedprec     /* medium precedence */
4  %left TOKEN4               /* highest precedence */
5
6  %%
7
8  (* grammar *)
9  nonterminal:
10   | case1 %prec precname {resulting value}
11   ...
```

- The order defines the precedence
- Associativity may be either `left` or `right`
- If you want to specify precedence but not associativity you can use `nonassoc`
- `%prec precname` specifies the associativity and precedence for a grammar production to be the ones of `precname`

Further Resources – links on my webpage (lceragioli.github.io)

- The ocamllex manual – if you have doubts
- The menhir manual – some more advanced features
- The Section "Parsing with OCamllex and Menhir" from the book Real World OCaml – for an overview
- The calculator OCaml example

Parsing MiniImp and MiniFun

Parsing MiniImp

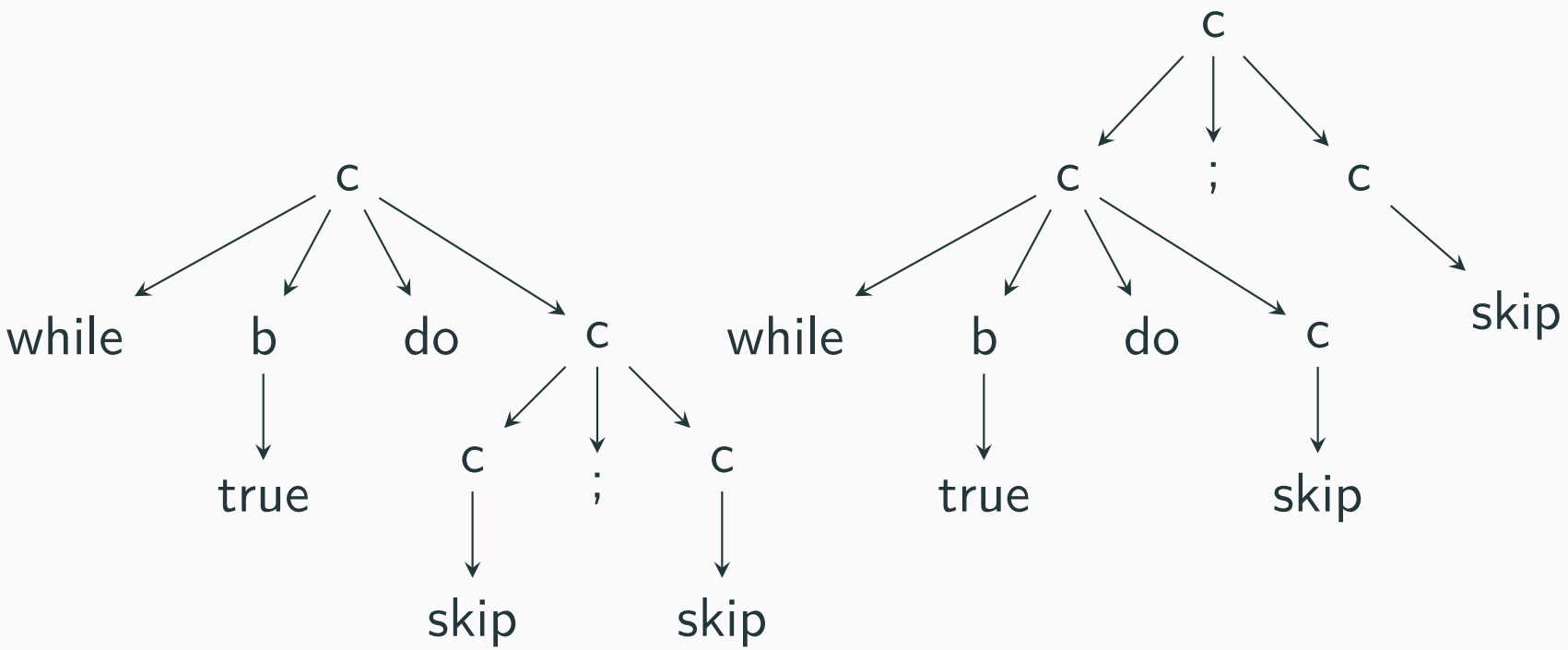
$$
\begin{aligned}
p &:= \texttt{def main with input } x \texttt{ output } y \texttt{ as } c \\
c &:= \texttt{skip} \mid x := a \mid c; c \\
&\quad \mid \texttt{if } b \texttt{ then } c \texttt{ else } c \mid \texttt{while } b \texttt{ do } c \\
b &:= v \mid b \text{ and } b \mid \text{not } b \mid a < a \\
a &:= x \mid n \mid a + a \mid a - a \mid a * a
\end{aligned}
$$

- We have already defined the *abstract* syntax of the language
- Defining the lexer is not difficult
- For parsing, the first attempt would be to directly feed menhir with the grammar of MiniImp
- However, the grammar above is ambiguous, you will see warning messages!

MiniImp Ambiguity

```
1  while true do skip ; skip
```

Two possible parsing trees



MiniImp Ambiguity

This is a classical example of a shift-reduce conflict, because the parser will get in a state like the following, where both

- shift (i.e. push the remaining portion of the input to the stack)
- reduce (i.e. recognize the current state of the stack as the non-terminal c)

are reasonable moves and produce a correct syntax tree.

Stack	Input
`$while true do skip`	`; skip$`

The common choice here (for good reasons) is to parse this command as `(while true do skip);skip`.

Other Ambiguities

- This is just an example of an ambiguity in MiniImp
- There are more, e.g. in arithmetical expressions and in $c_1; c_2; c_3$
- Find, and fix them using `menhir`
- The final code should **not** have conflicts reported by `menhir` (you cannot rely on the default choices made by menhir, even when they are correct: your code must not have warnings!)

Parenthesis

- Sometimes we want to use different associativity rules and precedences than the standard ones of the language
  For example forcing $5 + 2$ before multiplication in $4 * 5 + 2$.
- For allowing this, we enrich our concrete semantics with parenthesis to guide the parsing
  For example, we can write $4 * (5 + 2)$.
- We can do the same with commands:

```
1  while true do (skip; skip)
```

Parenthesis

A more useful example:

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 0 do (
5      out := out + x;
6      x := x - 1
7    )
```

Parsing MiniFun

- Same problems as MiniImp
- For example `fun x => x 1` can be either `(fun x => x) 1` which gives 1 and `fun x => (x 1)` which gives a second order function that applies the function `x` to `1`
- This time some ambiguities may be better managed by refactoring the grammar instead of specifying the associativity and precedence in `menhir`, but it's up to you :P
- Recall to discuss your choices in the report!

First Part of the Project: MiniImp and MiniFun Interpreters

Project Fragment

(You should already have the modules for the AST and semantics)

1. Extend the concrete syntax with parenthesis for forcing the evaluation order (no need to change the abstract syntax)
2. Define lexers and parsers for MiniImp and MiniFun (or MiniTyFun, as you prefer) by using ocamllex and menhir
3. Get rid of ambiguities: menhir should not produce warnings!
4. Write in the report a section explaining the ambiguities you have found and how you have solved them
5. Write a pair of interpreters (ocaml programs), one for MiniImp and one for MiniFun/MiniTyFun that:
   - Read a MiniImp/Fun program passed as a parameter
   - Read an integer input for the MiniImp/Fun program, passed via standard input
   - Evaluate the program given the input and print the resulting integer in standard output

Project Fragment – Example

The program `sum.miniimp`

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 0 do (
5      out := out + x;
6      x := x - 1
7    );
8    skip
```

Running the interpreter:

```
$ miniimpinterpreter sum.miniimp
4     <-- your input
10    <-- the output of the program
```