COMS 4115 PLT LRM: **Orlang**

Andrei Coman (ac4808, Systems Architect)
Scott Geng (skg2152, Tester)
Eumin Hong (eh2890, Manager)
Christopher Yoon (cjy2129, Language Guru)
Alan Zhao (asz2115, Systems Architect)

December 21, 2022

Contents

1 Introduction

Orlang is a functional programming language modelled after Haskell, OCaml, and the Lambda Calculus. Orlang is aimed to be a "mini"-OCaml, an educational general-purpose functional programming language showcasing the implementation and behavior of features popular in functional programming languages. Accordingly, Orlang is syntactically similar to OCaml, and supports strict evaluation, a static, strong and polymorphic type system based on the Damas-Hindley-Milner type system, as well as more fundamental features such as currying, higher-order functions, recursion (no loops!), and various list operations via our Prelude library.

2 Type System, Types, and their Representations

Orlang is a statically typed language supporting polymorphic types and the Damas-Hindley-Milner (HM) Type System. The lexical conventions of built-in types are deferred to Section 3.

2.1 Type System

Orlang supports the HM Type System. In particular, Orlang implements a variant of the Algorithm W, which performs type inference and checking on the Abstract Syntax Tree during the semantic checking stage via the substitution and the unification algorithm.

A noteworthy aspect of our (and the HM) type system is that polymorphism can only be introduced via `let` bindings (i.e. let polymorphism), with every other construct - lambdas, most importantly, being monomorphic. Additionally, a name introduced recursively via let polymorphism must be used monomorphically within its definition, with polymorphism allowed only afterwards.

Our type system is defined on the following types:

Orlang Types

```
type typ =
    Concrete of string
  | Unit
  | TypVar of string
  | ArrowTyp of typ * typ
  | ListTyp of typ
```

where the `Concrete` type may represent primitive types such as `Int`s, `Float`s, `Bool`s, and `Char`s; the `Unit` type represents a "nothing" and is used to denote the return type of the top-level `main` function, or functions with side-effects, such as `printing`; `TypVar` represents type variables; `ArrowTyp` represents the types of functions; and `ListTyp` represents list types and is parametrically polymorphic. All of these, as well as their representation in memory, are discussed in the rest of this section.

An important note is that type variables (`TypVar`s) always remain in scope after they are introduced, so two type annotations using type variables with the same name will reference the same type variable. For instance, the following will fail:

Orlang Explicit Annotation

```
val x : ‘a
let x = 3

val y : ‘a
let y = true
```

2.2 Type Annotations

Orlang allows the annotation of top-level let-bindings via the `val` keyword and annotation of expressions via the `:` operator. Our type annotation is different (and a bit more restrictive) then that of Haskell and OCaml in the following ways:

(i) Orlang does not allow the inline type annotation of names introduced via `let` (e.g. `let x : int = ...` is not allowed). However, we support inline type annotations on the right-hand side for any expression (e.g. `let x = 1 : Int` works).

(ii) All type annotations are monomorphic, and are maximally generalized in the context of let-bindings; this means that, in the context of let-bindings, type annotations assume maximal polymorphism, whereas, in any other case, they remain monomorphic.

(iii) Any type annotation is valid as long as it can be unified with its inferred type, even if it is more general than the most general unifier.

Our type annotations work as followed: Functions and variables can be explicitly annotated by users. Orlang borrows OCaml's syntax for type annotations. For a given expression `e`, the type annotation follows after the `:` symbol, as below:

Orlang Explicit Annotation

```
e : type
```

For a top-level let-binding, the type annotation precedes the definition, as below:

Orlang Type Annotations

```
val pi : Float
let pi = 3.141592

val add : Int -> Int -> Int
let add x y = x + y
```

Annotations are not strictly necessary; the types of un-annotated functions and variables will be inferred by the type system anyways. Type-annotated programs will have their types checked against the corresponding types inferred by the type system. A type annotation is considered valid insofar as it can be unified with the inferred type.

2.3 Primitive Types

The following are the primitive types in Orlang:

- `Unit '()'`: a 64-bit `0`, like `null`.
- `Int`: A 64-bit signed integer type, allowing the unary operator `-` to denote negative integers (e.g. `-1`).
- `Float`: A floating-point numerical type in decimal representation stored in 64-bits. Orlang uses a dot `.` to indicate the fractional part; we allow the unary operator `-` to denote negative numbers (e.g. `-3.14`); we do not support exponents yet (e.g. `1.0E5` is not recognized).
- `Bool`: A boolean type, stored in 64 bits; it has two literals, `true` and `false`.
- `Char`: A character type, representing a single character and stored in 64 bits. Characters are encapsulated in single quotes (e.g. `'a'`, `'!'`, `'x'`). Special characters are escaped with a backslash (e.g.

'\'', '\n').

In the IR, all integral types are stored as `void *`, so they all occupy 64 bits.

2.4 Lists

Orlang supports lists. We first discuss their representation in memory before their abstraction in our type.

Lists are implemented as linked lists in memory, where each element is stored in a node, not necessarily adjacent to its neighboring elements in the list in memory. The node contains a pointer to the actual value, as well as the pointer to the next node. For instance, the following list

Orlang Lists

```
val lst : List Int
let lst = [4, 1, 1, 5]
```

would be represented as below in memory:

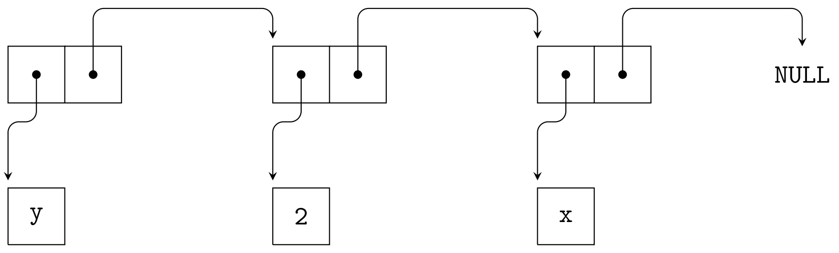

Figure 1: List representation in memory

At the compiler level, we support the following primitive functionalities for lists:

Orlang Language-level List Functions

```
let lst1 = 1::1::[5] in (* list consing *)
let lst = 4::lst1 in (* list consing *)
let (x::xs) = lst in (* unpacking lists, x = 4, xs = [1, 1, 5] *)
let (a, b, c) = xs in (* unpacking lists, a = 1, b = 1, c = 5 *)
in |.lst.| (* list length computation *)
```

With the above, we provide more standard list functions (e.g. `head`, `tail`, `map`, `reverse`, `filter`, `sort`, `sort_by`, `foldl`, `foldr`, `extend`, `concatenate`) as Prelude functions.

As a type, a list is parametrically polymorphic. They may be defined on type variables, any primitive type in 2.3, as well as lists (i.e. nested lists). Consequently, functions that act on lists concern their structure, and not the specific type of the values they contain. For instance, our Prelude function `map`:

Orlang list map

```
val map : ('mapa -> 'mapb) -> List 'mapa -> List 'mapb
let rec map f lst =
  match |.lst.| with
  | 0 => []
  | otherwise => let (x :: xs) = lst in (f x) :: (map f xs)
  ;
```

works on any function `f : ‘a -> ‘b` and list `lst : List ‘a`.

2.4.1 Strings

Strings in Orlang are represented as a `List Char`, list of characters.

2.5 Arrow Types

The types of functions and operators are defined by the arrow type, ‘`->`’. For instance,

Orlang Arrow Types

```
let mult10 x = 10 * y
(* mult10: Int -> Int *)
```

the function `mult10` is of type `Int -> Int`, a function that takes an `Int` argument and returns an `Int`.

3 Lexical Conventions

3.1 Comments

Our language supports multi-line comments, which are opened with a `(*`, and closed with a `*)`, following OCaml syntax. These comments can be nested at arbitrarily many levels:

Orlang Comments

```
(* this is
   ignored *)
(* this is (* doubly *) ignored *)
```

This is achieved at the lexer level via the following method, written in OCaml:

OCaml Comments Lexing

```
rule tokenize = parse
...
| "(*" {comment 1 lexbuf}
and comment level = parse
| "*)" { if level = 1 then tokenize lexbuf
        else comment (level - 1) lexbuf }
| "(*" { comment (level + 1) lexbuf }
| _ { comment level lexbuf }
```

3.2 Identifiers

Identifiers are strings of (case-sensitive) letters and digits. The first character of an identifier must be a lowercase letter; consecutive letters may be arbitrary-case letters and numerical digits. Users are encouraged to use camelCase.

OCaml Identifier Lexing

```
| [‘a’-‘z’] [‘a’-‘z’ ‘A’-‘Z’ ‘0’-‘9’ ‘_’]* as id { VARIABLE(id) } }
```

3.3 Reserved Keywords/Symbols

The following keywords/symbols are reserved in Orlang and may not be redefined or used as otherwise:

Orlang Keywords

```
    if then else
    match with otherwise
    let rec in
    where and
    val
    true false
    ,
    -> =>
    : :: ;
    |. .|
    \ (for lambda)
    =
    [ ] List
    ( )
    (* *)
    ()
    && || !
    + +. - -. * *. / /. %
    == > < => <=
    print_internal, ord_internal, chr_internal
```

3.4 Literals

3.4.1 Integers

Integer literals are sequences of digits; negative numbers begin with the negative (-) sign.

OCaml Integer Literal

```
let digit = ['0'-'9'] in
let int_literal = (-)? digit+
```

3.4.2 Floating Points

Floating-point numbers, introduced in Section 2.3, would look like

Orlang Float

```
val pi: Float
let pi = 3.141592
```

Formally, float numbers consist of an optional sign, an integer part, a decimal, and a fractional part. Either the integer part or the fractional part may be excluded, but not both. Floats are lexed via the regex:

OCaml Float Literal

```
let float_literal = (-)? (digit+ \. digit* | digit* \. digit+)
```

3.4.3 Boolean

A Boolean literal is either `true` or `false`.

Orlang Boolean

```
val x : Bool
let x = true

val y : Bool
let y = false
```

3.4.4 Characters

Character literals hold single characters enclosed with single quotes:

Orlang Character

```
val x: Char
let x = ‘a’
```

3.5 Lists

Lists will be enclosed by square brackets, as followed:

Orlang List

```
let l = [1, 2, 3]
```

At the grammar level, lists will be parsed as followed:

OCaml List Grammar

```
arg:
  ...
  | LBRACKET lst RBRACKET { NoHint(ListLit($2)) }
  ...

lst:
  | { [] }
  | expr { [$1] }
  | expr COMMA lst { ($1)::($3) }
```

3.5.1 List Construction

Lists can also be constructed be cons-ing, as followed:

OCaml List Grammar

```
let lst = 1::2::[3, 4, 5] in
let zlst = 0::lst
```

Their grammar is defined as simply as

OCaml List Grammar

```
expr
  ...
  | expr DCOLON expr { NoHint(LCons($1, $3)) }
  ...
```

and at the IR level, creates a new node for the element being prepended, whose "next" pointer points to the head of right-hand side list. During our semantic checking, given a sequence `a::b::c::...::xs`, we check that (only) the last identifier in the sequence is of type `List a`, and all of the preceding identifiers `a, b, c, ...` are of type `a`.

3.6 Operators

Orlang currently supports several built-in operators. We formally specify the output and types of each operator as follows:

Operator	Type	Function
+	`Int -> Int -> Int`	integer addition
−	`Int -> Int -> Int`	integer subtraction
∗	`Int -> Int -> Int`	integer multiplication
/	`Int -> Int -> Int`	integer division
%	`Int -> Int -> Int`	integer modulo
+.	`Float -> Float -> Int`	float addition
−.	`Float -> Float -> Int`	float subtraction
∗.	`Float -> Float -> Int`	float multiplication
/.	`Float -> Float -> Int`	float division
&&	`Bool -> Bool -> Bool`	boolean conjunction
||	`Bool -> Bool -> Bool`	boolean disjunction
!	`Bool -> Bool`	boolean negation
==	`'a -> 'a -> Bool`	physical equality between any types
>	`'a -> 'a -> Bool`	greater than
>=	`'a -> 'a -> Bool`	greater than or equal to
<	`'a -> 'a -> Bool`	less than
<=	`'a -> 'a -> Bool`	less than or equal to
`ord`	`Char -> Int`	char to int conversion
`chr`	`Int -> Char`	int to char conversion
`sitofp`	`Int -> Float`	int to float conversion
`fptosi`	`Float -> Int`	float to int conversion
`string_of_int`	`Int -> List Char`	int to string conversion
`string_of_bool`	`Bool -> List Char`	bool to string conversion
`string_of_char`	`Char -> List Char`	char to string conversion

4 Top-Level Definitions

At the top-level, Orlang programs consist of let-bindings and type-annotations. The entry-point into the program is the main function, which is mandatory for a syntactically correct source file (e.g. a library which should not be run by itself will not contain a main function). Top-level let-bindings are discussed in the following section together with expression let-bindings.

OCaml Top-Level Definitions

```
topLevel:
```

```
| letBinding topLevel { NoHint(Let($1, $2)) }
| typeAnn letBinding topLevel { let (v1, tp) = $1 in
                             let v2 =
                                match $2 with
                                | Binding(LVar(v), _, _) -> v
                                | _ -> raise(Failure("topLevel parsing error"))
                             in
                             if v1 = v2
                             then Hint(Let($2, $3), tp)
                             else raise(Failure("type annotation must be \
                                               immediately followed by \
                                               an accompanying definition"))
                           }
| letBinding { NoHint(Let($1, Hint(Var "main", Unit))) }
| typeAnn letBinding { let (v1, tp) = $1 in
                             let v2 =
                                match $2 with
                                | Binding(LVar(v), _, _) -> v
                                | _ -> raise(Failure("topLevel parsing error"))
                             in
                             if v1 = v2
                             then Hint(Let($2, Hint(Var "main", Unit)), tp)
                             else raise(Failure("type annotation must be \
                                               immediately followed by \
                                               an accompanying definition"))
                           }
letBinding:
| LET REC binding { let b =
                              match $3 with
                              | Binding(v, e, _) -> Binding(v, e, true)
                              | _ -> raise(Failure("cannot do tuple/list \
                                                                  in recursive let
                                                                     binding"))
                             in b
                           }
| LET REC binding WHERE wheres{ let b =
                              match $3 with
                              | Binding(v, e, _) ->
                                    let body = fold_right
                                       (fun bd acc ->
                                          match bd with
                                          | Binding(w, f, _) ->
                                              NoHint(Call(NoHint(Lambda(w, acc)), f))
                                          | _ -> raise(Failure("nope"))
                                       )
                                       $5 e in
                                       Binding(v, body, true)
                              | _ -> raise(Failure("cannot do tuple/list in recursive \
                                                  let binding"))
                             in b
                           }
| LET binding { $2 }
| LET binding WHERE wheres { let b =
                              match $2 with
```

```
                               | Binding(v, e, _) ->
                                  let body = fold_right
                                      (fun bd acc ->
                                         match bd with
                                         | Binding(w, f, _) ->
                                             NoHint(Call(NoHint(Lambda(w, acc)), f))
                                         | _ -> raise(Failure("nope"))
                                      )
                                      $4 e in
                                  Binding(v, body, false)
                               | _ -> raise(Failure("cannot do tuple/list with let
                                   binding with where"))
                             in b
                           }

wheres:
| binding AND wheres { $1 :: $3 }
| binding { [$1] }
```

5 Expressions

OCaml Expressions Grammar

```
expr:
| call { $1 }
| expr COLON monoType { match $1 with
                             | NoHint(e) -> Hint(e, $3)
                             | Hint(e, t) -> if t = $3
                                            then Hint(e, $3)
                                            else raise(Failure(
                                                 "multiple distinct type annotations \
                                                  referencing the same construct"))
                           }
| LAMBDA lambda { $2 }
| letBinding IN expr { NoHint(Let($1, $3)) }
| expr PLUS expr { NoHint(Binop(ADD, $1, $3)) }
| expr MINUS expr { NoHint(Binop(SUB, $1, $3)) }
| expr TIMES expr { NoHint(Binop(MLT, $1, $3)) }
| expr DIV expr { NoHint(Binop(DIV, $1, $3)) }
| expr MOD expr { NoHint(Binop(MOD, $1, $3)) }
| expr FPLUS expr { NoHint(Binop(FADD, $1, $3)) }
| expr FMINUS expr { NoHint(Binop(FSUB, $1, $3)) }
| expr FTIMES expr { NoHint(Binop(FMLT, $1, $3)) }
| expr FDIV expr { NoHint(Binop(FDIV, $1, $3)) }
| expr BAND expr { NoHint(Binop(AND, $1, $3)) }
| expr BOR expr { NoHint(Binop(OR, $1, $3)) }
| expr DCOLON expr { NoHint(LCons($1, $3)) }
| expr DOUBLEEQUALS expr { NoHint(Binop(EQ, $1, $3)) }
| expr LT expr { NoHint(Binop(LT, $1, $3)) }
| expr LTE expr { NoHint(Binop(LTE, $1, $3)) }
| expr GT expr { NoHint(Binop(GT, $1, $3)) }
| expr GTE expr { NoHint(Binop(GTE, $1, $3)) }
| BNOT expr { NoHint(Unop(NOT, $2)) }
| IF expr THEN expr ELSE expr { NoHint(If($2, $4, $6)) }
| MATCH expr WITH patternMatrix SEMICOLON { patternsToIfElse(PatternMatch($2, $4)) }
| PRINT expr { NoHint(Print($2)) }
| ORD expr { NoHint(Ord($2)) }
| CHR expr { NoHint(Chr($2)) }

multVars:
| VARIABLE COMMA multVars { (LVar($1))::($3) }
| VARIABLE { [LVar($1)] }

consVars:
| VARIABLE DCOLON consVars { (LVar($1))::($3) }
| VARIABLE { [LVar($1)] }

lst:
| { [] }
| expr { [$1] }
| expr COMMA lst { ($1)::($3) }

patternMatrix:
| GUARD expr DARROW expr { [PatternRow(Pattern($2), $4)] }
| GUARD OTHERWISE DARROW expr { [PatternRow(PatDefault, $4)] }
| GUARD expr DARROW expr patternMatrix { PatternRow(Pattern($2), $4)::($5) }

lambda:
| VARIABLE ARROW expr { NoHint(Lambda(LVar($1), $3)) }
| VARIABLE lambda { NoHint(Lambda(LVar($1), $2)) }

call:
| arg { $1 }
| call arg { NoHint(Call($1, $2)) }

arg:
| LITERAL { NoHint(IntLit($1)) }
| CLITERAL { NoHint(CharLit($1)) }
| FLITERAL { NoHint(FloatLit($1)) }
| TRUE { NoHint(BoolLit(1)) }
| FALSE { NoHint(BoolLit(0)) }
| VARIABLE { NoHint(Var($1)) }
| UNIT { NoHint(UnitLit) }
| LPAREN expr RPAREN { $2 }
| LBRACKET lst RBRACKET { NoHint(ListLit($2)) }
| GUARDDOT expr DOTGUARD { NoHint(LLen($2)) }
```

5.1 Lambda Expressions

Lambda expressions follow their corresponding definition from Lambda Calculus and can be either defined as single-variable lambda expressions or multi-variable lambda expressions (which is simply syntactic-sugar for nested lambdas). As expected, lambda expressions "greedily" consume the following input until they are forcefully stopped, using parenthesizing. In this case, our syntax is taken from Haskell:

Orlang Lambda Expressions

```
let five =
  let f = (\x y -> 1 * (x - 2) * y) in
  let g x y = x * y in
  let x = 3 in
  let y = 4 in
  f (g x 2 - 3) ((\z -> z + 1) y)
```

At the grammar level, lambdas are parsed variable by variable:

OCaml Variable Parsing Grammar

```
expr:
| LAMBDA lambda { $2 }
...

lambda:
| VARIABLE ARROW expr { NoHint(Lambda(LVar($1), $3)) }
| VARIABLE lambda { NoHint(Lambda(LVar($1), $2)) }
```

Lambda expressions are implemented as an anonymous structs with associated constructors and call operators. Since Orlang supports higher-order functions, Lambda expressions are always allocated on the heap.

The constructor stores the lambda capture `by value` (as Haskell and OCaml do), and the call operator performs the associated computation.

5.2 If-then-else Expressions

If-then-else expressions behave the same as their OCaml/Haskell counterparts. The clauses are evaluated lazily.

Orlang If-then-else

```
let x = 3
let y = (\z -> \t ->
           if x
           then 1
           else 0 + 2) 0 0
```

Grammar:

OCaml If-then-else Grammar

```
expr:
| IF expr THEN expr ELSE expr { NoHint(If($2, $4, $6)) }
```

5.3 Let Bindings

Let Bindings are, by default, non-recursive (i.e. the scope of the name being defined does not include its definition). For recursive definitions, the keyword `rec` will be used, as in OCaml. Each top-level let binding can contain an optional where-clause modelled from its Haskell equivalent. Names defined in the where-clause can be used in the let-binding. Similarly, names defined in a where-clause can depend on previously such-defined names. For disambiguation, multiple definitions in a single where-clause are separated by the keyword `and`. Note that let bindings are not syntactic sugar for lambda expressions because of requirements imposed by the HM type inference [3] with respect to polymorphism; this is discussed in the previous chapters.

Syntax:

Orlang Let Bindings

```
let y =
  let x = 5 in
  let x = x + 1 in
  x

let five2 =
  f (g x 2 - 3) ((\z -> z + 1) y)
  where f = (\x y -> 1 * (x - 2) * y)
    and g x y = x * y
    and x = 3
    and y = 4

val fac : Int -> Int
let rec fac n =
  if n == 0 then 0
  else if n == 1 then 1
  else (fac (n - 1)) + (fac (n - 2))
```

Top-level let-bindings are not expressions and can be accompanied by a type-hint introduced by the val keyword.

However, inline let-bindings are expressions and can be used in any context where another expression is expected. Similar to lambdas, our grammar parses them variable by variable.

OCaml Inline Let-Bindings Grammar

```
expr:
...
| letBinding IN expr { NoHint(Let($1, $3)) }
...

letBinding:
| LET REC binding ...
| LET REC binding WHERE wheres ...
| LET binding ...
| LET binding WHERE wheres ...

wheres:
| binding AND wheres ...
| binding ...

binding:
| VARIABLE EQUALS expr ...
| VARIABLE binding ...
| LPAREN VARIABLE COMMA multVars RPAREN EQUALS expr ...
| LPAREN VARIABLE DCOLON consVars RPAREN EQUALS expr ...
```

5.3.1 MBindings and CBindings: List unpacking

Lists may be unpacked in the manner of cons and tuples:

```
let lst = [1, 2, 3]
in let (x::y::xs) = lst (* x = 1, y = 2, xs = [3] *)
in let (a, b, c) = lst (* a = 1, b = 2, c = 3 *)
...
```

They are defined as `CBinding` and `MBinding` respectively in the parser ('C' for 'Cons' and 'M' for 'Multiple'), and appear in the left-hand side of a `let` binding. As shown, each variable is separated by `::` and `,` respectively, and must be encapsulated by parenthesis. They are defined as followed in our parser:

OCaml List Unpacking Grammar

```
binding:
  ...
  | LPAREN VARIABLE COMMA multVars RPAREN EQUALS expr
                           { MBinding((LVar($2))::($4), $7) }
  | LPAREN VARIABLE DCOLON consVars RPAREN EQUALS expr
                           { CBinding(LVar(($2))::($4), $7) }
  ...

multVars:
  | VARIABLE COMMA multVars { (LVar($1))::($3) }
  | VARIABLE { [LVar($1)] }

consVars:
  | VARIABLE DCOLON consVars { (LVar($1))::($3) }
  | VARIABLE { [LVar($1)] }
```

An important note is that MBindings and CBindings may only appear in let bindings.

5.4 Function Application

Function application appears in two distinct contexts: built-in functions (+, -, *, / etc) which translate easily to machine-level instructions and user-defined functions (add2, subtract etc). Function application has the highest precedence (and, so, more complicated expressions require parenthesizing) and is left-associative. Because our language is following the lambda-calculus, function definition is curried, and function application can consequently be done greedily.

Orlang Function Application

```
val add : Int -> Int -> Int
let add x y = x + y

val subtract : Int -> Int -> Int
let subtract x y = x - y

val add2 : Int -> Int
let add2 x = add 2 x

val uselessMath : Int -> Int -> Int
let uselessMath x y = add (subtract x 1) (subtract y 1)
```

At the grammar level, function application/the juxtaposition operator is represented as its own non-terminal (as opposed to other operators). This choice has been made to establish the high precedence of function application. Since juxtaposition corresponds to whitespace, its precedence cannot be fixed as with the other operators.

OCaml Function Application Grammar

```
expr:
| call { $1 }
 ...
| expr PLUS expr { NoHint(Binop(ADD, $1, $3)) }
| expr MINUS expr { NoHint(Binop(SUB, $1, $3)) }
| expr TIMES expr { NoHint(Binop(MLT, $1, $3)) }
| expr DIV expr { NoHint(Binop(DIV, $1, $3)) }
| expr MOD expr { NoHint(Binop(MOD, $1, $3)) }
| expr FPLUS expr { NoHint(Binop(FADD, $1, $3)) }
| expr FMINUS expr { NoHint(Binop(FSUB, $1, $3)) }
| expr FTIMES expr { NoHint(Binop(FMLT, $1, $3)) }
| expr FDIV expr { NoHint(Binop(FDIV, $1, $3)) }
| expr FMOD expr { NoHint(Binop(FMOD, $1, $3)) }
| expr BAND expr { NoHint(Binop(AND, $1, $3)) }
| expr BOR expr { NoHint(Binop(OR, $1, $3)) }
| expr DCOLON expr { NoHint(LCons($1, $3)) }
| expr DOUBLEEQUALS expr { NoHint(Binop(EQ, $1, $3)) }
| expr LT expr { NoHint(Binop(LT, $1, $3)) }
| expr LTE expr { NoHint(Binop(LTE, $1, $3)) }
| expr GT expr { NoHint(Binop(GT, $1, $3)) }
| expr GTE expr { NoHint(Binop(GTE, $1, $3)) }
| BNOT expr { NoHint(Unop(NOT, $2)) }
...
| PRINT expr { NoHint(Print($2)) }
| ORD expr { NoHint(Ord($2)) }
| CHR expr { NoHint(Chr($2)) }
| SITOFP expr { NoHint(SItoFP($2)) }
| FPTOSI expr { NoHint(FPtoSI($2)) }

call:
| arg { $1 }
| call arg { NoHint(Call($1, $2)) }

arg:
| LITERAL { NoHint(IntLit($1)) }
| CLITERAL { NoHint(CharLit($1)) }
| FLITERAL { NoHint(FloatLit($1)) }
| TRUE { NoHint(BoolLit(1)) }
| FALSE { NoHint(BoolLit(0)) }
| VARIABLE { NoHint(Var($1)) }
| UNIT { NoHint(UnitLit) }
| LPAREN expr RPAREN { $2 }
| LBRACKET lst RBRACKET { NoHint(ListLit($2)) }
| GUARDDOT expr DOTGUARD { NoHint(LLen($2)) }
```

5.5 Printing

Any printing functionality (`print`, `print_endline`, `print_int`, `print_int_endline`) is a function that returns a unit, `Unit`. Any program in Orlang that calls printing is thus not pure.

Printing is always done to `stdout` using ASCII characters, and supports escaped characters such as `\n`, `\t`, `\b`, `\r`, `\'`. Printing is achieved via translation to the LLVM IR instructions for printing.

Below is an example of printing:

Orlang Print "hello!"

```
let main =
  print_endline ['h', 'e', 'l', 'l', 'o', '!']
```

Printing is represented in our grammar as:

```
(* in parser *)
expr:
  ...
  | PRINT expr { NoHint(Print($2)) }
  ...

(* in scanner *)
 ...
| "print_internal" { PRINT }
```

At the compiler level, only the `print_internal` is implemented; all of the user-facing printing functionalities are provided via Prelude functions (to be discussed in Section 7) which call `print_internal` internally. That is, users are not expected to write `print_internal` directly in their source.

5.6 Precedence and Associativity of Expressions

The following directives establish the precedence and associativity of different expressions/operators.

Orlang Expression Directives

```
%right ARROW DARROW
%left GUARD
%left ELSE
%left IN
%right COLON DCOLON
%left DOUBLEEQUALS LT LTE GT GTE
%left BAND BOR
%left BNOT
%left PLUS MINUS FPLUS FMINUS
%left TIMES DIV MOD FTIMES FDIV
%left PRINT ORD CHR
%left LIST
```

Note that the arrow, corresponding to lambda expressions and pattern matching, has the lowest precedence whereas the juxtaposition operator (implemented by a non-terminal) has the highest precedence.

6 Syntactic Sugar

6.1 Lambda Expressions

Multi-variable lambda-expressions get desugared to nested single-variable lambda-expressions as follows. The example below will get translated to the the latter, which is a simple use of regular lambdas:

Orlang Multi-Variable Lambda-Expression

```
\x y -> x + y
```

Orlang Nested Single-Variable Lambda-Expression

```
\x -> \y -> x + y
```

This translation can be done at the grammar level as the variables are parsed progressively:

OCaml Multi-Variable to Nested Single-Variable Lambda-Expression Grammar

```
expr:
expr:
| LAMBDA lambda { $2 }
...

lambda:
| VARIABLE ARROW expr { NoHint(Lambda(LVar($1), $3)) }
| VARIABLE lambda { NoHint(Lambda(LVar($1), $2)) }
```

6.2 Pattern Matching

Orlang supports a restrictive version of pattern matching, structured as followed:

```
    match expr with
    | p1 => e1
    | p2 => e2
    | otherwise => e3
    ;
```

In particular, each case to be matched with the `expr` starts with a guard (`|`) followed by a pattern `p1`, the `=>` symbol, and the expression to use at a match. The final case of the pattern matching should be the keyword `otherwise` as the “default” pattern, and the whole pattern matching block ends with a semicolon (`;`).

Pattern matching is parsed into a “pattern matrix”

```
(expr, [(p1, e1), (p2, e2), ..., (otherwise, edefault)])
```

and is later desugared into a nested `if ... then ... else` statement, naively into (roughly)

```
    IF Binop(EQ, expr, p1)
    THEN e1
    ELSE
      IF Binop(EQ, expr, p2)
      THEN e2
      ELSE
      ...
        ELSE edefault
```

We note the following shortcomings/restrictions:

(i) Due to the way we desugar pattern matching (in particular, into a direct equality checking), the behavior for the equality operator (`Binop(EQ, expr, p)`) has to be defined for the expressions being matched and the patterns.

(ii) We do not support any checking for useless patterns/clauses or exhaustiveness. Exhaustiveness is forcefully achieved by the user having to include the `otherwise` default pattern, even if their pattern matching is exhaustive without it.

At the grammar level, pattern matching is formalized as follows:

OCaml Pattern Matching Grammar

```
(* in ast.mli*)
type patternMatch =
    PatternMatch of hExpr * (patternRow list)
and patternRow =
    PatternRow of pattern * hExpr
and pattern =
    Pattern of hExpr
  | PatDefault
```

```

(* in parser.mly *)
expr:
...
| MATCH expr WITH patternMatrix SEMICOLON { patternsToIfElse(PatternMatch($2, $4)) }
...

patternMatrix:
| GUARD expr DARROW expr { [PatternRow(Pattern($2), $4)] }
| GUARD OTHERWISE DARROW expr { [PatternRow(PatDefault, $4)] }
| GUARD expr DARROW expr patternMatrix { PatternRow(Pattern($2), $4)::($5) }
```

7 Prelude

We provide a useful set of standard library functions via our `prelude.orl` file. As Orlang does not yet have a package management and import system, the Prelude is automatically imported in every Orlang source (`.orl`) file. The way Orlang does it is similar to how the `C` programming language performs importing; it dumps the source code in the `prelude` file to the beginning of other source files. Users are able to freely override or shadow the Prelude declarations.

The functions we provide in the Prelude are:

- List functionalities: `head`, `tail`, `map`, `reverse`, `filter`, `sort`, `sort_by`, `foldl` `foldr`, `extend`, `concatenate`
- Printing: `print`, `print_endline`, `print_int`, `print_int_endline`
- String conversions: `string_of_int`, `string_of_bool`, `string_of_char`

8 Example Programs

8.1 A Highlight of Orlang

Below is a small program that showcases key features that Orlang supports:

```
let applyFnList flist alist =
  let rec applyHelper fs as acc =
    match |.fs.| with
    | 0 => reverse acc
    | otherwise => let (f::ff) = fs in
                   let (a::aa) = as in
                   applyHelper ff aa ((f a)::acc)
    ;
  in applyHelper flist alist []

val fnList : List (Int -> Int -> Int)
let fnList =
  let add = (\x y -> x + y) in
  let subtract = (\x y -> x - y) in
  let multiply = (\x y -> x * y) in
  [add, subtract, multiply]

val print_list : List Int -> List ()
let print_list lst = map (\x -> print (extend (string_of_int x) [' '])) lst

val main : ()
let main =
  let lst1 = [1, 2, 3] in
  let lst2 = [4, 5, 6] in
  let curried = applyFnList fnList lst1 in
  let res = applyFnList curried lst2 in
  let pnt = print_list res
  in ()
```

8.2 Potpourri of fundamental language features

Monomorphism; non-recursive let bindings; lambdas; multi-variable lambdas; nested comments; function application; currying; where clause desugaring:

```
(* this is ignored *)
(* this is (* doubly *) ignored *)

val five : Int
let five =
  let f = (\x y -> 1 * (x - 2) * y) in
  let g x y = x * y in
  let x = 3 in
  let y = 4 in
  f (g x 2 - 3) ((\z -> z + 1) y)

val five2 : Int
let five2 =
  f (g x 2 - 3) ((\z -> z + 1) y)
  where f = (\x y -> 1 * (x - 2) * y)
    and g x y = x * y
    and x = 3
    and y = 4

val five3 : Int
let five3 =
  z
  where f = (\x y -> 1 * (x - 2) * y)
    and g x y = x * y
    and x = 3 and y = 4
    and u = g x 2 - 3
    and v = (\z -> z + 1) y
    and z = f u v

val six : Int
let six = (\x -> \y -> 6) 3 6

let main =
  print_int_endline (five * (10000 / 10) + five2 * (1100 % 1000) + five3 * 10 + six)
```

Let polymorphism; recursive let bindings used polymorphically outside of their definition and monomorphically inside of their definition; redundant recursion:

```
val fac : Int -> Int
let rec fac n =
  if n == 0 then 0
  else if n == 1 then 1
  else (fac (n - 1)) + (fac (n - 2))

val x : Int
let x = 20

val id : 'a -> 'a
let rec id = (\x -> x)

val applyFunc: ('b -> 'c) -> 'b -> 'c
let rec applyFunc f x = f x

let main = print_int_endline (applyFunc (applyFunc id fac) (applyFunc id x))
```

Booleans, and, or, not, if statement:

```
val f: 'a -> 'a
let f = (\x -> x)

let x = true
let tru = f (x || false)
let fls = f (x && (! true))
let main =
  print_int_endline
          (if tru
           then
```

```
            if fls
            then 0
            else 1
         else 0)
```

8.3 Type Systems

The following examples fail our type-checking, as expected:

```
(* unification error: main must be of type unit *)
let main = (\x -> x + 1)

(* unification error: cannot apply false to 1 + *)
let main = 1 + false

(* unification error: fac must be used monomorphically in definition *)
let rec fac n = if n then fac true
                   else fac 0
let main = fac false

(* unification error: if condition must be bool *)
let x = 3
let y =
  (\z ->
   \t -> if x
        then 1
        else 0 + 2) 0 0

(* unification error: no valid type for fac *)
let rec fac n = (\x -> x + (fac 1))
let main = fac 1

(* unification error: fac must be used monomorphically in def *)
let rec fac n = fac (n - 1) + (fac false)

(* unification error: id : int-> int cannot be called with bool *)
let id x = (x : Int)
let main = print_int_endline (id true)

(* unification error: Cannot unify concrete type Int with list type List Int *)
let main =
  let f x = x + 1 in
  let lst = [1, 2, 3]
  in f lst

(* unification error: Bool and Int cannot be unified *)
let main =
  let lst = [1, 2, 3]
  in true :: lst

(* cannot unify arrow type (Int) -> (Int) with concrete type Int *)
let main =
  let f x = x + 1 in
  let lst = [1, 2, 3]
  in f :: lst
```

8.4 Lists

List construction and unpacking:

```
let main =
  let lst = 4115 :: [4116, 4117] in
  let (x::xs) = lst in
  print_int_endline x

let main =
  let lst1 = [4115, 4116, 4117, 4118] in
  let newStuff = 4119 in
  let lst = newStuff :: lst1 in
  let (a :: b :: xs) = lst in
  print_int_endline (a + b)

let main =
  let lst = [4000, 100, 10, 5] in
  let (a, b, c, d) = lst in
  print_int_endline (a + b + c + d)

let main =
  let lst = [[1, 2, 3], [4, 5, 6]] in
  let (a, b) = lst in
  let (x::y::z::xs) = a in
  let (p::q::r::ss) = b in
  print_int_endline (x + y + z + p + q + r)
```

List length computation:

```
let main = |. [1, 2, 3, 4] .|

let main =
  let a = [0] in
  let b = (1 :: a) in
  let c = (2 :: b) in
  print_int_endline |.c.|
```

List construction with recursive functions:

```
val fibHelper : Int -> Int -> List Int -> List Int
let rec fibHelper n cnt acc =
  match (cnt == n) with
  | false => let (x::y::xs) = acc
             in fibHelper n (cnt + 1) ((x + y) :: acc)
  | otherwise => acc
  ;

val fibList : Int -> List Int
```

```
let fibList n =
  let start = [0, 1] in
  fibHelper n 1 start

val main : ()
let main =
  let fl = fibList 10 in
  print_int_endline |.fl.|
```

List types should be homogeneous. The following examples all fail to compile:

```
let main =
  let lst = [1, 2, 3]
  in true :: lst

let main =
  let lsta = [1, 2, 3, 4, 5] in
  let lstb = [true, false, false] in
  let lstc = extend lsta lstb in
  ()

let main =
  let f x = x + 1 in
  let lst = [1, f]
  in lst
```

8.5 Orlang Prelude

Below is our `prelude.orl` which contains implementations of numerous list functionalities, printing functions, and data-type conversions:

Orlang Prelude

```
(*--- Conversions ---*)
val abs: Int -> Int
let abs x =
  match (x < 0) with
  | true => 0 - x
  | otherwise => x
  ;

val ord : Char -> Int
let ord x = ord_internal x

val chr : Int -> Char
let chr x = chr_internal x

val sitofp : Int -> Float
let sitofp x = sitofp_internal x

val fptosi : Float -> Int
let fptosi x = fptosi_internal x

```

```
val string_of_int : Int -> List Char
let string_of_int x =
  let rec helper x lst =
    if x <= 9
    then chr(x + (ord '0')) :: lst
    else let r = x % 10 in
         let x = x / 10 in
         helper x ((chr (r + (ord '0'))) :: lst)
  in
  let str = helper (abs x) [] in
  if x < 0
  then '-' :: str
  else str

(*---
val string_of_float: Float -> List Char
let string_of_float x =
  let rec helper x lst =
    let x = x *. 10. in
    if x %. 1. == 0.
    then (chr ((fptosi x) + (ord '0'))) :: lst
    else helper x lst
  in
  let str = helper (x %. 1.) (string_of_int (fptosi x)) :: '.') in
  if x < 0.
  then '-' :: str
  else str
---*)

val string_of_bool : Bool -> List Char
let string_of_bool x =
  if x == true
  then ['t', 'r', 'u', 'e']
  else ['f', 'a', 'l', 's', 'e']

val string_of_char : Char -> List Char
let string_of_char x = x :: []

val int_of_bool : Bool -> Int
let int_of_bool x =
  if x == true
  then 1
  else 0

(*--- Printing ---*)

val print : List Char -> ()
let print x = print_internal x

val print_int : Int -> ()
let print_int x = print (string_of_int x)

val print_int_endline : Int -> ()
let print_int_endline x =
  let useless = print_int x in
  print ('\n' :: [])

val print_endline : List Char -> ()
let print_endline x =
  let useless = print x in
  print ('\n' :: [])

(*--- List Operations ---*)

val head : List 'heada -> 'heada
let head lst =
  let (x :: xs) = lst in x

val tail : List 'taila -> List 'taila
let tail lst =
  let (x :: xs) = lst in xs

val reverse : List 'reversea -> List 'reversa
let reverse lst =
  let rec reverse_helper lst acc =
    if |.lst.| == 0
    then
      acc
    else
      let (x :: xs) = lst in
      reverse_helper xs (x :: acc)
  in
  reverse_helper lst []

val map : ('mapa -> 'mapb) -> List 'mapa -> List 'mapb
let rec map f lst =
  match |.lst.| with
  | 0 => []
  | otherwise => let (x :: xs) = lst in
                 (f x) :: (map f xs)
  ;

val foldl : ('foldla -> 'foldlb -> 'foldla) -> 'foldla -> List 'foldlb -> 'foldla
let rec foldl f acc lst =
  match |.lst.| with
  | 0 => acc
  | otherwise => let (x :: xs) = lst in
                foldl f (f acc x) xs
  ;

val foldr : ('foldrb -> 'foldra -> 'foldra) -> 'foldra -> List 'foldrb -> 'foldra
let rec foldr f acc lst =
  match |.lst.| with
  | 0 => acc
  | otherwise => let (x :: xs) = lst in
                f x (foldr f acc xs)
  ;

```

```
val extend : List 'extenda -> List 'extenda -> List 'extenda
let extend lsta lstb =
  foldr (\x lst -> (x :: lst)) lstb lsta

val filter : ('filtera -> Bool) -> List 'filtera -> List 'filtera
let rec filter p lst =
  match |.lst.| with
  | 0 => []
  | otherwise => let (x :: xs) = lst in
                 if (p x)
                 then (x :: (filter p xs))
                 else (filter p xs)
  ;

val sort_by : ('sortbya -> 'sortbya -> Bool) -> List 'sortbya -> List 'sortbya
let rec sort_by cmp lst =
  match |.lst.| with
  | 0 => []
  | otherwise => let (x :: xs) = lst in
                 let lt a = cmp a x in
                 let nlt a = cmp x a in
                 extend (sort_by cmp (filter lt xs)) (extend [x] (sort_by cmp (filter
                     (nlt) xs)))
  ;

val sort : List 'sorta -> List 'sorta
let sort lst = sort_by (\x y -> x < y) lst

val concatenate : List List 'concatenatea -> List 'concatenate
let concatenate lst =
  foldr extend [] lst
```

Example usage of Prelude functions:

```
val print_list : List Int -> List ()
let print_list lst = map (\x -> print (extend (string_of_int x) [' '])) lst

val main : ()
let main =
  let lst = [6, 8, 3, 5, 9, 1, 4, 2, 7] in
  let prntdsc = print_list ascending
    where descending = sort lst
      and ascending = reverse descending
  in
  let prntnl = print ['\n'] in
  let prntodd = print_list (filter (\x -> x % 2 == 1) (sort lst))
  in ()
```

More conversions:

```
let main =
  let y = (10 * 3) in
  print_int_endline (fptosi (sitofp y))
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Da Hua Chen, Hollis Lehv, Amanda Liu, and Hans Montero. Rippl Language Manual. `https://github.com/al3623/rippl/blob/master/reports/Rippl_LRM.pdf?fbclid=IwAR0syNzac5itkpN55dV352BHNB2cTKzUb2YsTSViLA4RrFGBbnLtj6lwAFY`.

[3] Stephen Diehl. *Write You a Haskell.* StephenDiehl.com, 2015.