UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Compilers: Introduction to Fasto

Alessandra Di Pierro

alessandra.dipierro@univr.it

Using material from Cosmin Oancea, Jost Berthold and Troels Henriksen (DIKU - University of Copenhagen).

Department of Computer Science
University of Verona

2017 Compiler Lecture Notes

1. FASTO Language Semantics
   - Implementing a Lexer (Mosml-Lex) and Parser (Mosml-Yacc)

Fasto Language: Function Declaration and Types

$$
\begin{array}{lcl}
\textit{Program} & \rightarrow & \textit{Funs} \\[1em]
\textit{Funs} & \rightarrow & \texttt{fun}\ \textit{Fun} \\
\textit{Funs} & \rightarrow & \texttt{fun}\ \textit{Fun}\ \textit{Funs} \\[1em]
\textit{Fun} & \rightarrow & \textit{Type}\ \mathbf{id}\ (\ \textit{TypeIds}\ ) = \textit{Exp} \\[1em]
\textit{TypeIds} & \rightarrow & \textit{Type}\ \mathbf{id} \\
\textit{TypeIds} & \rightarrow & \textit{Type}\ \mathbf{id}\ ,\ \textit{TypeIds} \\[1em]
\textit{Type} & \rightarrow & \texttt{int} \\
\textit{Type} & \rightarrow & \texttt{char} \\
\textit{Type} & \rightarrow & \texttt{bool} \\
\textit{Type} & \rightarrow & [\textit{Type}] \\[1em]
\textit{Exps} & \rightarrow & \textit{Exp} \\
\textit{Exps} & \rightarrow & \textit{Exp}\ ,\ \textit{Exps}
\end{array}
$$

Fasto Language: Function Declaration and Types

$$
\begin{array}{lcl}
\textit{Program} & \rightarrow & \textit{Funs} \\
\\
\textit{Funs} & \rightarrow & \texttt{fun}\ \textit{Fun} \\
\textit{Funs} & \rightarrow & \texttt{fun}\ \textit{Fun}\ \textit{Funs} \\
\\
\textit{Fun} & \rightarrow & \textit{Type}\ \mathbf{id}\ (\ \textit{TypeIds}\ ) = \textit{Exp} \\
\\
\textit{TypeIds} & \rightarrow & \textit{Type}\ \mathbf{id} \\
\textit{TypeIds} & \rightarrow & \textit{Type}\ \mathbf{id}\ ,\ \textit{TypeIds} \\
\\
\textit{Type} & \rightarrow & \texttt{int} \\
\textit{Type} & \rightarrow & \texttt{char} \\
\textit{Type} & \rightarrow & \texttt{bool} \\
\textit{Type} & \rightarrow & [\textit{Type}] \\
\\
\textit{Exps} & \rightarrow & \textit{Exp} \\
\textit{Exps} & \rightarrow & \textit{Exp}\ ,\ \textit{Exps}
\end{array}
$$

- First-order functional language & mutually recursive functions.
- Program starts by executing “main”, which takes no args.
- Separate namespaces for vars & funs.
- Illegal for two formal params of the same function to share the same name.
- Illegal for two functions to share the same name.

Fasto Language: Basic Expressions

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{charlit} \\[1em]
Exp & \rightarrow & Exp \; \texttt{+} \; Exp \\
Exp & \rightarrow & Exp \; \texttt{-} \; Exp \\
Exp & \rightarrow & Exp \; \texttt{<} \; Exp \\
Exp & \rightarrow & Exp \; \texttt{==} \; Exp \\[1em]
Exp & \rightarrow & \texttt{if} \; Exp \; \texttt{then} \; Exp \; \texttt{else} \; Exp \\
Exp & \rightarrow & \texttt{let} \; \textbf{id} = Exp \; \texttt{in} \; Exp \\
Exp & \rightarrow & \textbf{id} \; () \\
Exp & \rightarrow & \textbf{id} \; (\; Exps \;)
\end{array}
$$

Fasto Language: Basic Expressions

$$
\begin{array}{lcl}
\mathit{Exp} & \to & \textbf{id} \\
\mathit{Exp} & \to & \textbf{num} \\
\mathit{Exp} & \to & \textbf{charlit} \\[1em]
\mathit{Exp} & \to & \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \\
\mathit{Exp} & \to & \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\
\mathit{Exp} & \to & \mathit{Exp}\ \texttt{<}\ \mathit{Exp} \\
\mathit{Exp} & \to & \mathit{Exp}\ \texttt{==}\ \mathit{Exp} \\[1em]
\mathit{Exp} & \to & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\[0.5em]
\mathit{Exp} & \to & \texttt{let}\ \textbf{id}\ \texttt{=}\ \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\[0.5em]
\mathit{Exp} & \to & \textbf{id}\ \texttt{()} \\
\mathit{Exp} & \to & \textbf{id}\ \texttt{(}\ \mathit{Exps}\ \texttt{)}
\end{array}
$$

- +, − defined on ints.
- =, < defined on basic-type values of same type.
- Static Scoping: `let` bindings and function declarations create new scopes.
- A `let id ...` may hide an outer-scope var also named `id`.
- Call by Value.

Demonstrating Recursive Calls and IO in Fasto

Fibonacci Example and Use of Read/Write

```
fun int fibo(int n) = if         (n = 0) then 1
                        else if (n = 1) then 1
                        else fibo(n-1) + fibo(n-2)

fun int main () = let w = write("Enter Fibonacci's number: \ n")
                  in  let n = read(int)
                      in let w = write("Result is: \ n")
                         in let ww = write(w)  //what is printed?
                             in write( fibo(n) )
```

Demonstrating Recursive Calls and IO in Fasto

Fibonacci Example and Use of Read/Write

```
fun int fibo(int n) = if          (n = 0) then 1
                        else if (n = 1) then 1
                        else fibo(n-1) + fibo(n-2)

fun int main () = let w = write("Enter Fibonacci's number: \ n")
                  in  let n = read(int)
                      in let w = write("Result is: \ n")
                         in let ww = write(w)  //what is printed?
                            in write( fibo(n) )
```

Polymorphic Functions `read` and `write`:

- the only constructs in FASTO exhibiting side-effects (IO).
- valid uses of read: `read(int)`, `read(char)`, or `read(bool)`; takes a type parameter and returns a (read-in) value of that type.
- `write` : $\alpha \rightarrow \alpha$, where α can be `int`, `char`, `bool`, `[char]`, or **stringlit**. `write` *returns a copy of its input parameter.*

Fasto Language: Array Constructors & Combinators

$$
\begin{array}{lcl}
\textit{Exp} & \rightarrow & \texttt{read} \texttt{ (} \textit{Type} \texttt{)} \\
\textit{Exp} & \rightarrow & \texttt{write} \texttt{ (} \textit{Exp} \texttt{)} \\
\textit{Exp} & \rightarrow & \textbf{stringlit} \\
\textit{Exp} & \rightarrow & \{ \textit{ Exps } \} \\
\textit{Exp} & \rightarrow & \texttt{iota} \texttt{ (} \textit{Exp} \texttt{)} \\
\textit{Exp} & \rightarrow & \texttt{map} \texttt{ (} \textbf{id} \texttt{ , } \textit{Exp} \texttt{)} \\
\textit{Exp} & \rightarrow & \texttt{reduce} \texttt{ (} \textbf{id} \texttt{ , } \textit{Exp} \texttt{ , } \textit{Exp} \texttt{)} \\
\textit{Exp} & \rightarrow & \textbf{id} \, [\, \textit{Exp} \,]
\end{array}
$$

- `read` / `write` polymorphic operators,
- array constructors: string literals, array literals, and `iota` (**latter is project task**)
- second-order array combinators (SOAC): `map` and `reduce` (**project task**),
- array indexing: check if index is within bounds.

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [char]`

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [char]`
- `iota(n)` $\equiv$ `{0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello"` $\equiv$ `{'H','e','l','l','o'} : [char]`
- `iota(n)` $\equiv$ `{0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`

Second-Order Array Combinators:

- `map(f, {`x_1`, .., `x_n`}) = {f(`x_1`), .., f(`x_n`)}`, where type of x_i : α, of `f` : $\alpha \rightarrow \beta$, of `map` : $(\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$

Fasto's Array Constructors & Combinators

Array Constructors:

- literals: `{ {1+2, x+1, x+y}, {5, ord('e')} } : [[int]]`
- **stringlit**: `"Hello" ≡ {'H','e','l','l','o'} : [char]`
- `iota(n) ≡ {0,1,2,...,n-1}`; *type of* `iota : int` $\rightarrow$ `[int]`

Second-Order Array Combinators:

- `map(f, {x`$_1$`, .., x`$_n$`}) = {f(x`$_1$`), .., f(x`$_n$`)}`, where type of `x`$_i$ `:` α, of `f :` $\alpha \rightarrow \beta$, of `map :` $(\alpha \rightarrow \beta)$ `*` $[\alpha] \rightarrow [\beta]$
- `reduce(`$\odot$`, e, {x`$_1$`,x`$_2$`,..,x`$_n$`}) = (..(e`$\odot$`x`$_1$`)..`$\odot$`x`$_n$`)`, where type of `x`$_i$ `:` α, of `e :` α, type of $\odot$ `:` $\alpha * \alpha \rightarrow \alpha$
  type of `reduce :` $(\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$.

Demonstrating Map-Reduce Programming

Can We Write Main Using Map and Reduce?

foldl : $(\beta * \alpha \to \beta) * \beta * [\alpha] \to \beta$ (* Haskell's foldl *)

foldl($\odot$, e, $\{x_1, .., x_n\}$) $\equiv$ $(..(e \odot x_1) .. \odot x_n)$

```
fun bool f(bool b, int a) = b && (a > 0)
fun bool main() = let x = {1, 2, 3}
                  in  foldl(f, True, x)
```

Demonstrating Map-Reduce Programming

Why Does Fasto *Not* Support Foldl?

```
foldl : (β * α → β) * β * [α] → β (* Haskell's foldl *)
foldl(⊙, e, {x1, .., xn}) ≡ (..(e ⊙ x1) .. ⊙ xn)

fun bool f(bool b, int a) = b && (a > 0)
fun bool main() = let x = {1, 2, 3}
                  in  foldl(f, True, x)
```

Because It Is Typically The Composition of a Map With a Reduce:

```
map :     (α → β) * [α] → [β]
map       (f,    {x1, .., xn}) ≡ {f(x1), .., f(x1)}

reduce : (α * α → α) * α * [α] → α
reduce   (⊙, e, {x1, .., xn}) ≡ (..(e ⊙ x1) .. ⊙ xn)

fun bool f(int a) = a > 0
fun bool main() = let x = {1, 2, 3} in
                  let y = map(f, x)
                  in  reduce(op &&, True, y)
```

Array Combinators & Read and Write

What Is Printed If The User Types in: "1 2 8 9"?

```
fun  int  writeInt  ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, {2, 2}) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

Array Combinators & Read and Write

What Is Printed If The User Types in: "1 2 8 9"?

```
fun  int  writeInt  ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, {2, 2}) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

`readIntArr(n)` "reads" a $1D$ array of n elements.

`map(readIntArr,{2,2})` $\equiv$ `map(readIntArr,{2,2})` $\equiv$ `{readIntArr(2),readIntArr(2)}` builds $2D$ array `{{1,2},{8,9}}`.

Array Combinators & Read and Write

What Is Printed If The User Types in: "1 2 8 9"?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, {2,2}) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

So, `map(readIntArr,{2,2})` $\equiv$ {{1,2},{8,9}}.

`reduce( plusV2, {0,0}, {{1,2},{8,9}} )` $\equiv$

Array Combinators & Read and Write

What Is Printed If The User Types in: “1 2 8 9”?

```
fun  int writeInt   ( int i ) = write(i)
fun  int  readInt   ( int i ) = read(int)
fun [int] readIntArr( int n ) = map( readInt, iota(n) )

fun [int] plusV2([int] a, [int] b) = { a[0]+b[0], a[1]+b[1] }

fun [int] main() = let arr2 = map(readIntArr, {2,2}) in
                   let arr1 = reduce(plusV2, {0,0}, arr2)     in
                       map( writeInt, arr1 )
```

So, `map(readIntArr,{2,2})` ≡ {{1,2},{8,9}}.

`reduce( plusV2, {0,0}, {{1,2},{8,9}} )` ≡
`plusV2( plusV2({0,0}, {1,2}), {8,9} )` ≡
`plusV2( {0+1,0+2}, {8,9} )` ≡ {1+8, 2+9} ≡ {9,11}.

Finally, `map(writeInt,arr1)` prints the elements of `arr1`, i.e., 9 11!

1. FASTO Language Semantics
   - Implementing a Lexer (Mosml-Lex) and Parser (Mosml-Yacc)

Project Example: Abstract-Syntax Tree (AbSyn)

Lexer implemented in `Lexer.lex` and compiled with
`mosmllex Lexer.lex`,

Parser implemented in `Parser.grm` and compiled with
`mosmlyac -v Parser.grm`,

AbSyn produce a program representation, e.g., the one in `Fasto.sml`:

```
type pos = int * int  (* position: (line, column) *)

datatype Exp =
    Constant of Value * pos       (* 345 *)
  | StringLit of string * pos     (* "lala" *)
  | Var of string * pos           (* identifier/variable name*)
  | Plus of Exp * Exp * pos       (* a + b *)
  | Minus of Exp * Exp * pos      (* a - b *)
  | If of Exp * Exp * Exp * pos  (* if a<2 then 1 else 2 *)
  | ...
```

Exp records the position in the original text file, so that compiler messages (errors) can be reported.

Mosml-lex: Lexical Analysis Magic (Lexer.lex)

```
{ (* boilerplate code for all lexer files ... *)
  open Lexing;
  exception LexicalError of string * (int * int) (* (message, (line, column)) *)

  val currentLine  = ref 1   (* reference to an int *)
  val lineStartPos = ref [0] (* reference to a list, holding the index of the *)
                             (* chars that start a new line, in reverse order *)
  fun resetPos ()  = (currentLine := 1; lineStartPos := [0])

  (* getLexemeStart token: the index of the start char of token in whole string*)
  (* getPos: computes the (line,column) corresponding to an input token *)
  fun getPos token = getLineCol (getLexemeStart token) (* useful for errors *)
                                (!currentLine) (!lineStartPos)
  and getLineCol pos line (p1::ps) =
        if pos>=p1 then (line, pos-p1)
        else getLineCol pos (line-1) ps
    | getLineCol pos line [] = raise LexicalError ("",(0,0))
  fun lexerError lexbuf s = raise LexicalError (s, getPos lexbuf)

  fun keyword (s, pos) = case s of  (* Language keywords and identifiers. *)
                         "if" => Parser.IF   pos    (* datatype for *)
                          ...                       (* IF, ..., ID   *)
                         | _  => Parser.ID (s, pos) (* in Parser.grm file*)
}
```

Mosml-lex: Lexical Analysis Magic (Lexer.lex)

```
rule Token = parse
    [‘ ‘ ‘\t‘ ‘\r‘]+  { Token lexbuf } (* ignore whitespace *)
  | "//" [^ ‘\n‘]*    { Token lexbuf } (* ignore comments   *)
  | [‘\n‘ ‘\012‘]     { currentLine := !currentLine+1;
                        lineStartPos:= (getLexemeStart lexbuf)::!lineStartPos;
                        Token lexbuf } (* update new-line positions *)

                      (* getLexeme returns the string representation of token *)
  | [‘0‘-‘9‘]+        { case Int.fromString (getLexeme lexbuf) of
                             NONE   => lexerError lexbuf "Bad integer"
                           | SOME i => Parser.NUM (i, getPos lexbuf)  }
  | [‘a‘-‘z‘ ‘A‘-‘Z‘] [‘a‘-‘z‘ ‘A‘-‘Z‘ ‘0‘-‘9‘ ‘_‘]* (* language keywords/ids *)
                      { keyword (getLexeme lexbuf, getPos lexbuf) }

    (*string literal, e.g., ""abc"" *)
  | ‘"‘ ([‘ ‘ ‘!‘ ‘#‘-‘&‘ ‘(‘-‘[‘ ‘]‘-‘~‘] | ‘\‘[‘ ‘-‘~‘])* ‘"‘
           { case (String.fromCString (getLexeme lexbuf)) of
                  NONE   => lexerError lexbuf "Bad string constant"
                | SOME s => let s1 = String.substring(s,1,String.size s-2)
                            in  Parser.STRINGLIT(s1, getPos lexbuf)  end
                     (* s1 = "abc", i.e., the string with quotes removed *)
  | ‘+‘                 { Parser.PLUS   (getPos lexbuf) }
  | ‘-‘                 { Parser.MINUS  (getPos lexbuf) }   ...
  | _                   { lexerError lexbuf "Illegal symbol in input" }    ;
```

Mosml-yacc: Parser Magic (Parser.grm)

```
%token <(int*int)>         PLUS MINUS LTH IF
%token <string*(int*int)> ID   STRINGLIT
%token <int*(int*int)>    NUM
...
%nonassoc ifprec
%left LTH
%left PLUS MINUS
```

LTH stands for <. At least three precedence levels:

Mosml-yacc: Parser Magic (Parser.grm)

```
%token <(int*int)>        PLUS MINUS LTH IF
%token <string*(int*int)> ID   STRINGLIT
%token <int*(int*int)>    NUM
...
%nonassoc ifprec
%left LTH
%left PLUS MINUS
```

LTH stands for <. At least three precedence levels:

- +, − are defined left associative, and bind tighter than tt ¡, e.g., `a + b < c + d` is parsed as `(a+b) < (c+d)`.
- < is left-associative and binds tighter than `if...then... else`,
- An `if ... then ... else` expression has the lowest precedence, meaning `if a < 3 then a + 2 else a + 1` is parsed as `if (a < 3) then (a + 2) else (a + 1)` and NOT AS `(if (a < 3) then (a + 2) else a) + 2`

Mosml-yacc: Parser Magic (Parser.grm)

```
%{
open Fasto
open Fasto.UnknownTypes
%}
%token <(int*int)>        PLUS MINUS LTH IF
%token <string*(int*int)> ID   STRINGLIT
%token <int*(int*int)>    NUM
...
%nonassoc ifprec
%left LTH
%left PLUS MINUS
%start Prog
%type <Fasto.UnknownTypes.Exp> Exp

Exp :     NUM          { Constant (IntVal (#1 $1),
                                          (#2 $1)) }
        | STRINGLIT       { StringLit $1 }
        | Exp PLUS  Exp   { Plus ($1, $3, $2) }
        | Exp MINUS Exp   { Minus($1, $3, $2) }
        | Exp LTH   Exp   { Less ($1, $3, $2) }
        | IF Exp THEN Exp ELSE Exp %prec ifprec
                          { If ($2, $4, $6, $1) }
        ...
```

- type of terminal `PLUS` is `(int*int)`, i.e., line-column position.
- each rule for nonterminal `Exp` produces a value of type `Fasto.UnknownTypes.Exp` (between arcolades).
- each nonterminal `Exp` in the right-hand side of a rule has type and carries the representation of `Fasto.UnknownTypes.Exp`.
- `$i` refers to the value carried by symbol *i* in a production.
- `#i` refers to element *i* of a tuple.
- for example `Plus ($1, $3, $2)` uses the constructor datatype `Exp = Plus of Exp*Exp*pos`, in which the two `Exp`-type args come from 1^{st} and 3^{rd} symbols in the rule, while the position comes from 2^{nd}, i.e., `PLUS`.

Project Example: Abstract-Syntax Tree (AbSyn)

Lexer implemented in `Lexer.lex` and compiled with `mosmllex Lexer.lex`,

Parser implemented in `Parser.grm` and compiled with `mosmlyac -v Parser.grm`,

- `Parser.Prog Lexer.Token (Lexing.createLexerString prgstr)` builds the

AbSyn : program representation, e.g., the one in `Fasto.sml`:

```
type pos = int * int  (* position: (line, column) *)

datatype Exp =
    Constant of Value * pos      (* 345 *)
  | Var of string * pos          (* identifier/variable name*)
  | Plus of Exp * Exp * pos      (* a + b *)
  | Minus of Exp * Exp * pos     (* a - b *)
  | If of Exp * Exp * Exp * pos  (* if a<2 then 1 else 2 *)
  | ...
```

Mosml-yacc: Parser Magic (Parser.grm)

```
%{
open Fasto
open Fasto.UnknownTypes
%}
%token <(int*int)>          PLUS MINUS LTH IF
%token <string*(int*int)> ID   STRINGLIT
%token <int*(int*int)>    NUM
...
%nonassoc ifprec
%left LTH
%left PLUS
%start Prog
%type <Fasto.UnknownTypes.Exp> Exp

Exp :   ...
        | Exp PLUS  Exp  { Plus ($1, $3, $2) }
        | Exp LTH   Exp  { Less ($1, $3, $2) }
        | IF Exp THEN Exp ELSE Exp %prec ifprec
                         { If ($2, $4, $6, $1) }
        ...
```

1) Explain production *Exp* $\rightarrow$ *Exp* LTH *Exp*:

1.a) What does it produce and from what?

1.b) How is it disambiguated (in the grammar)?

2) Explain the use of %*ifprec*.