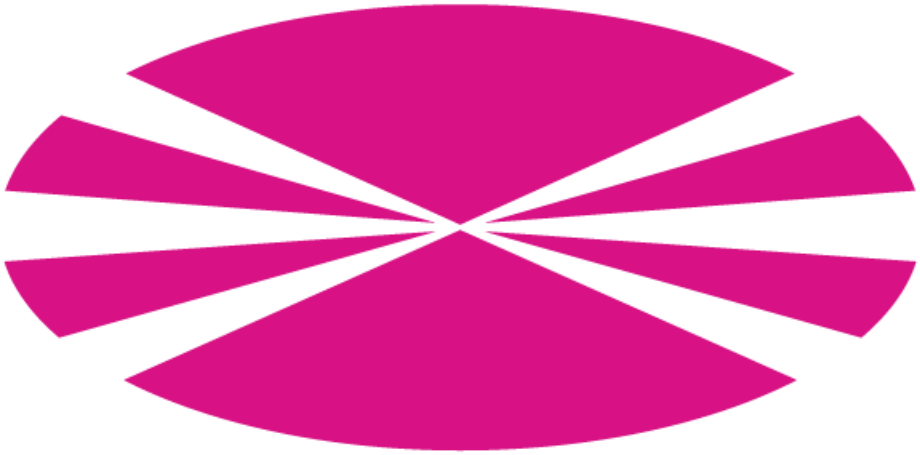


UNIVERSIDADE DA CORUÑA

# DESEÑO DE LINGUAXES DE PROGRAMACIÓN

Assignment 3

Lambda-Calculus implementation

Mateo Gende Lozano – m.gende@udc.es

Diego Suárez García – d.suarez@udc.es

# Contents:

# USER MANUAL:

## · Introduction

This program allows its user to “emulate” untyped lambda-calculus. It does so by evaluating a series of expressions, or commands, given by the user and then printing the results. These commands can be read from a file or introduced by the user in an interactive or toplevel command line. Each of them must be followed by a semicolon for separation.

## · Compilation

The implementation includes a Makefile, allowing the user to compile it just by opening a terminal and typing

```
$ make
```

## · Usage

The program can be run at any time by executing f. This can be done by opening a command terminal and typing

```
$./f
```

There are two main ways this program can be executed. One is as an interactive command line and the other as a file parser that will print the result of its execution. Running the program with no arguments will automatically run the interactive toplevel mode, while opening a file requires the user to specify the file’s name and location.

It is recommended the toplevel is used with the readline wrapper rlwrap.

```
$ rlwrap ./f
```

- **File mode:**

To open the program in file parsing mode, one must specify the filename. By default, the program expects a filename.f to be passed to it, and it provides a means to specify said file's location.

To simply run the program over a file located in the same directory as the program, one may simply run

```
$./f filename.f
```

Where filename is the name of the file to be parsed.

If the file is not located in the current directory, there's always the option to specify the file's location. To do this, type –I before the path to the file and after that, specify the filename, like this

```
$./f –I /path/to/file filename.f
```

The program only supports one file per execution, and providing more than one filename will produce an error. However, multiple paths can be provided for the program so it can search for the file in all of them. If it doesn't find it in the first one, it will just keep looking in the rest until it finds a filename that matches the one provided.

```
$./f –I /first/path/ -I /second/path/ filename.f
```

If the program finds the file, it will evaluate the commands written on it and return the result of each evaluation.

- **File structure:**

The program expects a file containing lambda-calculus expressions, strings or comments. Anything not in-between /*commentary marks*/ or "quotation marks" will be treated as a command.

- **Interactive mode:**

Whenever the program is executed without any file argument, it starts in interactive, or toplevel mode.

```
$./f
```

In this mode the user can enter any lambda-calculus command they want followed by a semicolon (;) and press enter to see the result of the execution. Whenever there is a parsing error, the system just continues running to prevent the user from losing all progress.

Additionally, some directives have been added to make it more usable in interactive mode.

**`#quit;`** Quit automatically ends the execution closing the program.

**`#var;`** Var allows the user to see all bindings that have been defined in the current toplevel context.

**#debug;** The debug command toggles debug mode, which in turn tries to show the steps followed in evaluating entered commands. This should be toggled off when using recursive functions as the output can completely blot the terminal.

# · Commands

- **Boolean commands:**

These cover the basic Boolean commands, true and false:

· **true; :** represents the “true” value in Boolean.
· **false; :** represents the “false” value in Boolean.

- **Numeric commands:**

These cover numerical representations, this program can use integers and floats.

· **0; :**0 represents the integer zero.

· **#; :** Any other integer number inputted will be assumed an integer. Internally, the program will use succ-pred notation but for input and output, numbers are used.
· **succ; :** Succ represents a successor, or next of an integer, thus succ ( 0 ) will correspond to 1, succ ( 1 ) to 2 and so on.
· **pred; :** Pred represents predecessor, or previous of an integer, thus pred ( 1 ) corresponds to 0 and so on. The predecessor of 0 is 0.
· **iszero; :** IsZero is a Boolean command that allows us to check if an integer is 0 or not.
· **#.#;** : Any number with a dot in-between represents a float. Float numbers can be multiplied but they can't be "succed", "iszeroed" or "predded".
· **timesfloat#.# #.#; :** Timesfloat is used to multiply two floating point numbers. It goes before its "inputs" so to multiply 2 × 3 one should input **`timesfloat 2.0 3.0;`**

- **String commands:**

The program allows its user to define strings.

· **"{string}"; :**Anything inserted between quotation marks will be treated as a literal string. This includes escaped characters like newlines (\n), tabs (\t), backslashes (\\), double quotes (\"), and single quotes (\') as well as any ASCII character annotated by its decimal value (\###).

- **Abstraction commands:**

An important pillar of lambda-calculus are abstractions. The program allows us to declare them as follows.

· **lambda {symbol} . {term}; :** Lambda allows us to define abstractions. Between the lambda and the dot, the symbol is defined, and after the dot the abstraction can be inserted. For example the abstraction that multiplies any given number by two can be expressed as **`lambda n . timesfloat 2.0 n;`**. Symbol may be replaced by an underscore '_' if the symbol used in the abstraction can be disregarded.

- **Binding commands:**

The user can define top-level bindings. These may be used to simply name variables or to assign them to terms.

· **{symbol}/; :** Typing the name of a variable followed by a slash allows us to declare a NameBind. This binds the variable to be used without assigning a term to it.
· **{symbol} = {term};** : This binds the inserted symbol to a term after evaluating said term.

- **Let commands:**

The program allows us to define let-clauses.

· **let {symbol} = {term1} in {term2}; :** These clauses "replace" any appearance of symbol in term2 with term1. An underscore '_' can be used instead of a symbol if it can be ignored.

- **Conditional commands:**

We can use If-then-else clauses to define conditional behaviour.

· **if {term1} then {term2} else {term3}; :**If term1 evaluates to true, then this becomes term2. If term1 evaluates to false, this becomes term3. This is a common way to define conditional clauses.

- **Record commands:**

Records can be defined and their fields accessed, either by name or by index, depending on the definition.

· **{ {term1}, {term2} …};** :Any series of terms separated by commas and betweencurly brackets (poor formatting choice in this document) will be treated as a record. Each of the fields can be accessed by their index, so if we define:

```
rec = { 10, 11};
```

We can then access 11 by calling

```
rec.2;
```

· **{ {symbol} = {term1}, {symbol} = {term2} …};** : Record fields can also be named if we want to access them by name. If we want rec to have fields foo and bar we can type:

```
rec = { foo = 10, bar = 11};
```

And access 11 by calling

```
rec.bar;
```

· **{term}.#;**: To access a field by index, just type the term followed by a dot and the index of the field to access.

· **{term}.{symbol};** :To access a field by field name, type the term followd by a dot and the field's name.

# · Directives

Typing number sign (#) followed by a reserved word will use toplevel functionality that is not available in file mode. These directives allow the user to better interact with what the program does.

- **#quit; :**
- **#exit; :**

Quit allows the user to exit the program when in the toplevel loop. It can be called at any time and will simply return the user to the terminal.

- **#var; :**

Var shows all currently bound variables (either NameBound or bound to a term) in the toplevel context. This can be useful to check what has been previously defined. To see the content of any of the variables, one can simply call its name.

- **#debug; :**

Debug toggles “debug mode”. When in this mode, the program will try to print each evaluation step it does, so instead of seeing just the result, the user will see a series of terms separated by → where each of them is a step in evaluation.

# · Function examples

These are simply examples of what lambda sentences may be defined with our program:

- **Identity function :**

```
id = lambda x . x ;
```

Id function is the simplest combinatory we can get. It receives a term and returns it as it is.

- **Multiply by two :**

```
mult2 = lambda x . timesfloat 2.0 x ;
```

This function applies timesfloat 2.0 to anything passed to it, effectively multiplying its input by two.

- **Fix function :**

```
fix = lambda f . (lambda x . f ( lambda y . x x y )) (lambda
x . f (lambda y . x x y));
```

This function is used to add recursiveness. It allows a called function to unfold and call itself.

- **Sum function :**

```
sumaux = lambda f . ( lambda n . ( lambda m . if (iszero n)
then m else succ ( f ( pred n ) m )));
sum = fix sumaux;
```

This is the “integer sum” function. It allows us to sum two integers together, returning the result.

- **Less-or-equal function :**

```
leqaux = lambda f . ( lambda m . ( lambda n . ( if iszero m
then true else ( if iszero n then false else f (pred m) (pred
n)))));
leq = fix leqaux;
```

This “less than or equal to” function takes two integers, **a** and **b** and returns true if **a <= b** and false otherwise.

- **Times function :**

```
timesaux = lambda f . ( lambda m . ( lambda n . (if (iszero
m) then 0 else sum n (f (pred m) n))));
times = fix timesaux;
```

This function allows us to multiply two integer numbers.

- **Factorial function :**

```
factaux = lambda f . ( lambda n . ( if (iszero n) then 1 else
times n ( f (pred n))));
fact = fix factaux;
```

This function defines the recursive factorial of an integer.

# CHANGES FROM ORIGINAL IMPLEMENTATION:

- **main.ml**
  - Originally when executing the program without specifying a file, the program would show an error message and close. Now *{toplevel}* is returned to a function that checks this case.
  - When *{toplevel}* is detected, it calls toplevel's functions to start the command line instead of crashing.

- **toplevel.ml**
  - This module was created from scratch to implement toplevel functionality. This receives the commands from the command line and passes them to the inner functions. It doesn't crash when detecting an error, it just doesn't execute the faulty command and remains in a safe state.
  - Its main function *top_level* writes a prompt and waits for the user's input. When it reads an input, it first checks if it is a toplevel-define directive like #quit or #var and acts accordingly if it is one. If it is a regular command, it calls *process_line*. *process_line* will return a context that will be passed again to top_level in a recursive loop.
  - *process_line* is an edited version of the original *process_file*. Instead of calling *parse_file* like in the original implementation, this calls *parseLine* with the input line. Here is where we catch exit exceptions to prevent the program from closing on a simple parse error. If an exception of this type is caught, *process_line* returns the same context that it received. After calling *parseLine* it calls *process_command* on each of the resulting commands.
  - *parseLine* works similarly to *parse_file* but instead of creating a lexingbuf from a file, it's created from the line as string calling *Lexing.from_string*. As with the original implementation, it returns a list of commands and a context.
  - *process_command* is analogous to the original *process_command* but this one also updates the binding list to be shown with the #var directive.

- **support.ml**
  - The pervasive module defined an abbreviation of *Format.print_string* which we changed to *Printf.prinf*. This change also had to be done on to be used.
  - The original *warning* and *warningAt* were unused functions. We renamed and used them to see toplevel error messages without crashing the program.

- **core.ml**
  - Here we have added the debug flag, which is enabled from toplevel when the corresponding directive is used. It is used in the eval function to print intermediate evaluations if needed, originally it just evaluated.

- **syntax.ml**
  - *name2index* is now terminal.
  - The original implementation had an error on *printtm_AppTerm* where it printed timesfloat wrong. It originally printed twice the second term of the multiplication, now it prints correctly.

# REFERENCE MANUAL:

This manual is intended to go along with the code it refers to. So as to serve as a better structured explanation than code comments.

It is divided in two parts: The first and longest refers to the modules, the second one is a practical example of how the system treats a command.

The Modules part has seven sub-parts, one dedicated to each of the modules. These include a summary of the module's functionality as well as a breakdown of the code they provide. The code of these modules is also commented, but there are some limitations on what can be shown on a source file.

The Practical Example part shows how the program treats a command since it is parsed until it is shown. This is meant to illustrate how the different modules come together to make the program work as intended. The first example is applying the identity function to integer number 5, from parsing to the end of evaluation. The second example is meant to show a deeper evaluation using a let clause "if true then let word = 0 in iszero word else 1" inside of an if. Longer commands would show all of the cases being used, but they also become unbearably large to show in each intermediate term.

# Modules:

## 1 – Main.ml

**Summary:**

The *Main* module works as the basic structure to run all other modules. It interprets the user's command line, it calls *Lexer* and *Parser* modules to parse either a file or the command line and it also calls the expression evaluator to execute and print commands and their results.

The *main* method gets the program going, it is located inside *res*. *Printexec* module makes it easier to print exceptions. If *main* is executed without errors, *res* will return 0. If *main* catches an *Exit* exception, *res* will take the error code associated to that exception. If any uncaught exception is thrown, *Printexec.catch* will print that exception's name and abort the program with error code 2.

If no uncaught exception is thrown, the program ends with an *exit res* call, where *res* represents the exit code and *exit* passes it on to the OS.

**Variables:**

· **searchpath:** where the directories specified by the user using keyword "-I" are stored.

· **argDefs**: triplet where the first element is keyword "-I" of type string. The second is an anonymous function that takes a string and adds it to the list where *searchpath* is pointing. The third one is a brief description string about what this function does. *argDefs* is used when parsing the command line inserted by the user.

· **alreadyImported**: where all already imported files' names are stored. It has no real use in the current program.

· **res**: after program execution, res works as an exit code. Its value will be 0 in case *main* has run without errors. If the exception *Support.Error.Exit x* was thrown, *res* will take x's value.

**Functions:**

· **val parseArgs : unit → string = <fun>**

parseArgs receives unit as an argument and returns a string. Said string can either be the name of the file to process introduced by the user or the string "{toplevel}" if no filename has been given to start the top-level. The first thing this does is to create a pointer to *String*

*option*, *inFile* with initial value *None*. Next, it calls OCaml's *Arg.parse* to parse the commands given when starting the program:

*val Arg.parse : (key * spec * doc) list → anon_fun → usage_msg → unit*

**key**: option keyword that must start with a '-'. In this case we search for "-I".

**spec**: gives the option type and the function to call when tis option is found on the arguments. In this case it is an *Arg.string* and the function to call whenever the keyword "-I" is found just adds whatever follows to *searchpath*.

**doc**: description of "-I" option.

**anon_fun**: function called every time *Arg.parse* finds an anonymous (unannotated) argument. In this case, this option matches what the user should do to introduce the name of the file to be parsed. Running the program with "./f test.f" will search for the file in current directory.

**usage_msg**: string used to show how to use the program. It is displayed whenever the user introduces "-help" or "—help" before listing all the possible options. Here it is empty.

If no file has been specified, the function returns "{toplevel}" to indicate the program should start the interactive command line. In case it points to some string, this string is returned as the file to be processed.

· **val openfile : string → in_channel = <fun>**

*openfile* receives the filename to open as a string argument and returns an input channel to that file. Remember that if the user introduced "-I" and a path when opening the program, that path was saved in *searchpath*. What *openfile* does is try to open the file in the paths indicated by *searchpath*. It declares an auxiliary recursive function to travel through the list. For each path in *searchpath*, it concatenates said path to the left of the filename to form the complete path. It then tries to open the file in that path calling *Pervasives*'s *open_in name* where name is the full path. If *open_in* throws a *Sys_error* exception because the system can't open the file in that path, it will try the other paths until either it gets to open it or it runs out of list. If the path is empty, it tries opening the file in the current working directory. Once the file is found, *open_in* returns an *in_channel*.

In case the full list has been traversed and the file hasn't been opened, it shows an error message with a call to *Support.err* finishing the execution.

· **val parseFile : string → Syntax.context → Syntax.command list * Syntax.context = <fun>**

*parseFile* receives the filename the user has specified when running the program. It first calls *openfile* to open said file and get an *in_channel*. For the *Lexer* to be able to scan the file, first *Lexing.create inFile pi* is called. This takes the the filename and the input channel to return a *lexbuf* buffer. Finally, *Parser.toplevel Lexer.main lexbuf* is called. Receiving the *Lexer's main* function and the buffer to parse the file content and returns a *Syntax.context → Syntax.command list * Syntax.context* function. A function that takes a context and returns a list of commands to be run and the context after those commands are executed. In case of an error when parsing, *Exit 1* exception is thrown and the program execution finishes with *error*.

· **val process_command : Syntax.context → Syntax.command → Syntax.context**

*process_command* receives a context and a command and executes the command on that context to receive a new context. For *Eval* type commands, it calls *Core.eval* to evaluate it and return a term where no more rules apply. It then calls *printtm_Aterm* to print this new resulting term and returns the same context.

For *Bind* commands, the binding is first evaluated calling *Core.evalbinding*, this returns the binding with its term evaluated. After that, the identifier list is updated and the new binding added to the context.

· **val process_file : string → Syntax.context → Syntax.context = <fun>**

*process_file* takes a filename to process and a context. It first adds the filename to *importedFiles* list which currently has no use. Next, it calls *parseFile f ctx* to parse said file. It then defines the internal function *g* that from an input context and a command, processes said command on that context calling *process_command*. Finally, it applies this function to each command in the list and returns the context.

· **val main : unit → unit = <fun>**

*main* is the "main" function in *Main*. It first calls *parseArgs* to parse the execution arguments given by the user. . If there is a filename return value, it starts *process_file* to parse the file. If the return value is "{toplevel}", indicating the user hasn't specified any filename, it starts the toplevel interactive command by calling *top_level emptycontext*, which receives a *Syntax.emptycontext* and starts parsing the lines inputted by the user.

# 2 – Toplevel.ml

**Summary:**

This module manages the toplevel command line. It allows us to input lambda-calculus expressions from the terminal, skipping the file parsing part. To run this interactive mode, just run the program without any arguments.

**Variables:**

· **exit_program:** list of strings containing the directives that close the program.

· **Identifiers**: list of user-defined identifiers representing bound variables.

**Functions:**

· **val update_identifiers : '_a → '_b → unit = <fun>**

*update_identifiers*receives a symbol and the binding type associated to a symbol. Inside its internal recursive function, it checks if the symbol has already been bound. If it is, it updates the binding type, if it isn't, it just adds it to the end of the list.

· **val parseLine : string → Syntax.context → Syntax.command list * Syntax.context = <fun>**

*parseLine* receives a string line the user has introduced in the terminal. For the line to be scanned by the lexer, *Lexing.from_string line* is called. This creates a *lexbuf* to give to *Parser.toplevel Lexer.main lexbuf*. This returns a function that takes a context and returns the list of commands that were in that line and the context that results from its execution. In case it runs into any errors when parsing, *Parsing.Parse_error* would be caught and the resulting command list will be empty.

**· val process_command : Syntax.context → Syntax.command → Syntax.context = <fun>**

*process_command* receives a context and a command and executes the command on that context to receive a new context. For *Eval* type commands, it calls *Core.eval* to evaluate it and return a term where no more rules apply. It then calls *printtm_Aterm* to print this new resulting term and returns the same context.

For *Bind* commands, the binding is first evaluated calling *Core.evalbinding*, this returns the binding with its term evaluated. After that, the identifier list is updated and the new binding added to the context. This binding is also added to the list of user-defined bindings to be printed when #var is called.

**· val process_line : string → Syntax.context → Syntax.context = <fun>**

*process_line* receives a line to be parsed and a context. It first calls *parseLine* to parse said line and apply the resulting function to the current context. This returns a pair formed by the list of commands in that line and a context which is discarded. It then defines an internal function *g* that calls *process_command* and applies it to each command in the list.

*process_line* also catches any Exit exception preventing the program from closing like in file mode. When it catches one, it just returns the same context that was entered, effectively preserving current state.

**·val print_identifiers : Syntax.context → unit = <fun>**

*print_identifiers* traverses the symbol*binding list and prints each bound variable on screen.

**· val top_level : Syntax.context → 'a = <fun>**

*top_level* is the main function in *Toplevel*. It receives a context, initially *Syntax.emptycontext* and recursively reads lines inserted by the user to process them and generate a new context each time it is called. It first writes the prompt and waits for the user to insert an expression and press Enter. It then reads the input and checks if it is a directive to close the program. If so, it just throws *Exit 0*. Otherwise, it checks if the user has introduced *#var* directive to see the identifiers. Then it checks if the #debug directive is inserted to toggle debug mode.If none of these apply, it calls *process_line* to process the input and generate a new context to be used in the next recursive call.

# 3 - Lexer.mll

**Summary:** Lexer.mll is the lexical analyser. This file automatically generates lexer.ml when compiled with ocamllex. The analyser accounts for line and column numbers to report errors.

It defines a list of reservedWords, which pair a string literal to a function that returns its info as a token. As an example the reserved word “if” is (“if”, fun i → Parser.IF i). These reserved words are stored in a Hashtbl so when any word is parsed, it is first looked up in said hash table.

It works just as any other lexical analyser does, it parses and matches strings by patterns. When it finds a comment, it enters comment rules and these support nested comments and call errors when these are not nested properly. When it finds a literal string, it parses it using a buffer, supporting /### format and escape characters. It supports both integers and floats as well as explicit line numbering and filename references for error reporting.

Whenever a word is parsed, lexer calls createID. This function tries to match the string with any of the reserved words to return its token value. If it can’t, it will return it as a Parser.UCID or Parser.LCID depending if it is upper or lowercase, its location info and its string value.

### Variables:

· **val lineno : int ref:** tracks the line number through the execution, it is used in error reporting to give the location of the token that produced the error.

· **val depth : int ref:** tracks the depth of nested comments.

· **val start : int ref:** tracks the location (column) of the first character in a line, it is used to calculate column value in error reporting.

· **val filename : string ref:** stores the location of a filename being parsed.

· **val startlex : Support.Error.info ref:** stores information about where lexbuf was located whenever the parser enters a comment or string literal. This allows the program to mark where an unclosed comment or string literal starts, instead of reporting the error at the eof, where it is identified.

· **val stringBuffer : bytes ref:** is a buffer that stores a string when it is being parsed as a literal (as in “between double quotes”).

· **val stringEnd : int ref:** *stringEnd* tracks where the string contained in stringBuffer ends, it is reset with resetStr, incremented with addStr and used in getStr.

## Functions:

**· val createID : Support.error.info →string → Parser.token = <fun>**

*createID* takes a string and tries to find it in the hashtable of reserved words. If it can match it, it will return the reserved word's token. If it can't, it will create either a *Parser.UCID* or a *Parser.LCID* depending on if the string is upper or lowercase, with its location information and its string value.

**· val create : string →in_channel → Lexing.lexbuf = <fun>**

*create* takes a string representing a filename and an input channel. If the filename starts with an explicit reference to the current directory, then global variable *filename* will point to said filename. If it doesn't, this function will assume it is in the current working directory and concatenate this to the filename before storing it in 'filename'. It then calls Lexing.from_channel which returns a lexer buffer when given the input channel.

**· val newline : Lexing.lexbuf →unit = <fun>**

*newline* increases the linenumber counter and updates the starting column to the first character of the new line.

**· val info : Lexing.lexbuf →Support.Error.info = <fun>**

*info* calls *support* function *createInfo*, which creates an *FI* structure containing the filename, line and column where an error was found from the file and line counters, as well as the starting character position and the current starting character on *lexbuf*.

**· val addStr : char →unit = <fun>**

*addStr* is used when parsing literal strings, it takes a character and inserts it in a string buffer. If this buffer is already full, it creates a new buffer twice the length of the old one, copies the old content and continues using this new one. This buffer becomes the full string literal once the lexer finishes reading something between double quotation marks.

**· val resetStr : unit →unit = <fun>**

*resetStr* resets the end of the string buffer used in parsing string literals back to the beginning.

· **val getStr : unit →string = <fun>**

*getStr* returns the content of the string buffer as a string. This is used when a literal string has finished parsing.

· **val extractLineno : string →int →int = <fun>**

*extractLineno* obtains a line number from the text when either a "# [number]" or a "# line [number]" is found using an offset. It basically disregards the "# line" part and converts the number from string to int.

· **val text : Lexing.lexbuf →string = <fun>**

*text* just calls the *Lexing.lexeme* function. This returns the string value of the currently matched string. This is to be used whenever the actual value of a token needs use.

## Rules:

- **val main : Lexing.lexbuf →Parser.token = <fun>**
- **val __ocaml_lex_main_rec : Lexing.lexbuf → int →Parser.token = <fun>**

Main rules define the lexer's default behaviour.

· Whenever it finds a word, it calls createID to match it with a reservedWord or to create a new upper or lowercase ID.

· Whenever it finds an integer, it returns it as a Parser.INTV, parses its string value to int and returns it with its location info.

· Whenever it finds a float, it returns it as a Parser.FLOATV, parses its string value to float and returns it with its location info.

· Whenever it finds a comment start (/*), it sets depth to 1, stores location in startLex and applies comment rules.

· Whenever it finds a comment end (*/), it calls an error as there's no open comment.

· Whenever it finds a double quotation mark ("), it stores location in startle and applies string literal rules.

· Whenever it finds either "# [number]" or "# line [number]", it extracts the number to update lineno and calls getFile rules.

· If it gets to eof, it returns it as a Parser.EOF token with its location information.

· If it finds a tab or formfeed, it ignores it, and /n updates current line and column.

· Any special character accepted is treated as if it were a word.

Anything that doesn't fit in these rules is treated as an error.

- **val comment : Lexing.lexbuf →unit = <fun>**
- **val __ocaml_lex_main_rec : Lexing.lexbuf → int →unit = <fun>**

Comment rules define the lexer's behaviour when it finds a comment start (/*).

· Whenever it finds a comment start it increases depth and keeps parsing using comment rules.

· Whenever it finds a comment end it decreases depth and will keep on using comment rules as long as there's any depth.

· Whenever it finds a newline, it updates current line and column.

· If it gets to eof, it calls an error, as that means some comment isn't properly closed. The error is reported using lexStart, which should mark the beginning of the comment.

Anything that doesn't fit any of these rules is treated as comment content and disregarded.

- **val getFile: Lexing.lexbuf →Parser.token = <fun>**
- **val __ocaml_lex_getFile_rec : Lexing.lexbuf →int →Parser.token = <fun>**

GetFile has a single rule that ignores whitespaces and expects a double quote (") to call getName rules.

- **val getName: Lexing.lexbuf →Parser.token = <fun>**
- **val __ocaml_lex_getName_rec : Lexing.lexbuf →int →Parser.token = <fun>**

GetName has a single rule that treats anything other than a double quote (") or newline (\n) as a filename and updates 'filename' variable accordingly. It then calls finishName rules.

- **val finishName: Lexing.lexbuf →Parser.token = <fun>**
- **val __ocaml_lex_finishName_rec : Lexing.lexbuf →int →Parser.token = <fun>**

FinishName has a single rule that expects a double quote (“) and calls main rules again.

- **val string: Lexing.lexbuf →Parser.token = <fun>**
- **val __ocaml_lex_string_rec : Lexing.lexbuf →int →Parser.token = <fun>**

String rules are used when a double quote (“) marking the beginning of a string literal is found.

· Whenever a second double quote (“) is found, it returns a Parser.STRINGV with location information stored in startle and the string stored in stringBuffer as value.

· Whenever a backslash is found (\) marking an escaped character, it tries to add the character to stringBuffer using escaped rules.

· Whenever a newline (\n) is found, it adds it to stringBuffer and updates line and column accordingly.

· If it finds eof, it calls an error because there’s some string that hasn’t been closed.

Any other character found is added to the string using addStr.

- **val escaped: Lexing.lexbuf →char = <fun>**
- **val __ocaml_lex_escaped_rec : Lexing.lexbuf →int →char = <fun>**

Escaped rules are used when parsing a literal string and a backslash (\) is found, marking an escaped character.

· Whenever an n is found following the backslash, newline (\n) is returned.

· Whenever a t is found following the backslash, tab (\t) is returned.

· Whenever a backslash (\) is found following the first backslash, backslash (\\) is returned.

· Whenever a double quote (“) is found following the backslash, double quote (\034) is returned.

· Whenever a single quote (‘) is found following the backslash, single quote (\’) is returned.

· Whenever a 3 digit number is found following the backslash, it is treated as an ASCII character. The program checks if it is lower than 255 (otherwise it calls an ‘Illegal character’ error) and converts it into char using Char.chr.

Anything else is treated as an ‘Illegal character’.

# 4 - Parser.mly

**Summary:** Parser.mly defines the Yacc grammar for the parser. When compiled with ocamlyacc, it automatically generates both parser.ml and parser.mli. It defines a set of tokens recognised by the lexical analyser and a set of productions that describe the grammar.

Each token is annotated with the type of data it carries. This mostly translates to Support.Error.info containing the file, line and column where the token was found, but in some cases (namely UCID – uppercase ID, LCID – lowercase ID, INTV – integer value, FLOATV – float value and STRINGV – string literal value, which are not reserved words) it also includes the actual value of the read token, words in the case of UCID and LCID, integer or float value for INTV and FLOATV and string for STRINGV.

The production rules work as any other syntax parser, a starting production is defined, 'toplevel' in this case, and a set of rules whose structure should be matched follow. The program matches the current structure with any valid rule and returns any value assigned to

```
type:
        TOKEN
            {value to return if TOKEN is found}
      | type2 TOKEN2
            {e.g.: $1 + $2.v (sums the value returned by type2 to
            the value of TOKEN2)}
type2:
        TOKEN3
            {$1.v (returns the value of TOKEN3)}
      | TOKEN3 TOKEN4
            {$1.v - $2.v (returns the subtraction of both token's
            values}
```

said rule. For example:

This makes it so that if the program reads a structure like TOKEN3 TOKEN4 TOKEN2 it returns TOKEN3.v – TOKEN4.v + TOKEN2.v.

In this case, the grammar begins by expecting a toplevel, which is defined as a sequence of commands, each terminated by a semicolon (;). When a toplevel is recognised, the program returns a Syntax.context → (Syntax.command list * Syntax.context), a function that takes a naming context and returns a parsed list of commands as well as the new naming context with the names bound in the commands. When parsing each command, new rules are used, and most of these return a function that expects a Syntax.context and call one of Syntax' functions with the command's given arguments.

Yacc works by parsing tokens returned by the lexical analyser, building a stack and trying to match that stack with any of the rules. This means the code sentences (or values returned in this case) are executed when Yacc is able to match the rule, rather than when it detects the first token in a rule. Each rule works by using the values returned by the rules below it, be they simple tokens or more complex rule results. This makes trying to understand the code from top to bottom at first glance kind of weird, and easier to figure out when taking into account what each rule is made of.

## Grammar:

**· toplevel:**

The toplevel is a sequence of commands separated by semicolons.

**EOF**

Following the rules, reading End of File would return a function that takes a context and returns an empty list, but as this empty list is passed backwards, all the other detected commands are appended to it.

**| Command SEMI toplevel**

This defines a list of commands separated by semicolons (;). The program takes the list of commands and context passed by the already-parsed rest of the toplevel and sets the newly-parsed command at the head of the list of commands.

**· Command:**

Each toplevel command can either be a term or a binding.

**Term**

Any parsed term is returned as an Eval evaluation command in the current context.

**| LCIDBinder**

If it is a binding, its name is added to the context and it is returned as a Bind binding.

**· Binder:**

A binder follows a word and binds it in current context.

**SLASH**

A word followed by a slash (/) is treated as a NameBind.

**| EQ Term**

A “word = Term” is returned as a TmAbbBind of said term.

**· Term:**

Terms can either be applications, if-then-else clauses, abstractions or let clauses.

**AppTerm**

An application is returned after being parsed in deeper rules.

**| IF Term THEN Term ELSE Term**

An if-then-else clause is returned as a TmIf with all the parsed inner terms.

**| LAMBDA LCID DOT Term**

An abstraction matching “lambda word . Term” adds said word to the context and returns a TmAbs abstraction with the word and term on the new context.

**| LAMBDA USCORE DOT Term**

An abstraction can also be created while disregarding the variable name using an underscore (_).

**| LET LCID EQ Term IN Term**

A “let word = Term in Term” clause is returned as a TmLet of the terms on current context and the last one added to the context.

**| LET USCORE EQ Term IN Term**

A let clause can also be created with an underscore, disregarding the variable name.

**· AppTerm:**

Application terms can be “pathterms” (projections or atomic terms), nested applications or a multiplication, successor, predecessor or iszero clause.

**PathTerm**

PathTerms are parsed deeper and simply returned.

**| AppTerm PathTerm**

An application of an application is also parsed deeper and returned as a TmApp.

**| TIMESFLOAT PathTerm PathTerm**

Float multiplications are returned as TmTimesfloat command of the parsed terms.

**| SUCC PathTerm**

A successor of a PathTerm becomes a TmSucc of said term.

**| PRED PathTerm**

A predecessor of a PathTerm becomes a TmPred of said term.

**| ISZERO PathTerm**

And iszero returns a TmIsZero of the term.

**· PathTerm:**

PathTerms can either be projections or atomic terms.

**PathTerm DOT LCID**

A projection “Term.word” is returned as a TmProj. The dot’s location is used in the info.

**| PathTerm DOT INTV**

A projection using an integer index is also returned as a TmProj.

**| ATerm**

An atomic term is parsed deeper and returned as-is.

**· ATerm:**

Atomic terms are the ones that never require extra parentheses. They can be a Term between parentheses, TRUE, FALSE, a lowercase word, fields between curly brackets, a float, a string literal or an integer.

**LPAREN Term RPAREN**

Parentheses mark precedence, and once a Term has been parsed in-between them, they can be disregarded. The result of Term is returned.

**| TRUE**

True is returned as a TmTrue with its location information.

**| FALSE**

False is returned as a TmFalse with its location information.

**| LCURLY Fields RCURLY**

A list of fields surrounded by curly brackets {} is parsed deeper as list then returned as a TmRecord record.

**| FLOATV**

Floats become TmFloat with location info and value.

**| STRINGV**

String literals are returned as TmString with location info and content.

**| INTV**

And integers can either be TmZero or a succession of TmSuccs of TmZero. This is calculated and returned.

**· Fields:**

A list of fields found between curly brackets can either be empty or nonempty.

"Nothing" marks an empty list of fields, which returns an empty list.

**| NEFields**

A nonempty list of fields is parsed deeper and returned.

**· NEFields:**

A nonempty list of fields separated by commas.

**Field**

The “last” or “only” field of a list is parsed deeper and returned as a list member.

**| Field COMMA NEFields**

Once parsed, the rest of the fields are appended to the list returned by the parsed NEFields.

**· Field:**

Each field can be named or indexed by an integer.

**LCID EQ Term**

A named field is returned with its name.

**| Term**

A simple term is returned with an index.

# 5 – Core.ml

**Summary:** The *core* module defines the core evaluation functions. It contains the rules for recursively evaluating terms until *Syntax* is needed.

## Variables:

· **debug:** This acts as a flag to check if we are in debug mode. It is changed when toplevel detects the #debug directive. When this is true, terms are printed before being evaluated, effectively showing intermediate terms of evaluations.

## Functions:

· **val isnumericval : 'a → Syntax.term → bool = <fun>**

*isnumericval* checks if the term passed as an argument is a number or not. If the term is *TmZero*, it returns true, if it is a *TmSucc* of something, *isnumericval* calls itself recursively until it finds *TmZero*, any other term returns false.

· **val isval : 'a → Syntax.term → bool = <fun>**

*isval* returns true of the term passed as an argument is a value, false if it isn't. If the term is a *TmFalse, TmFloat, TmString* or *TmAbs*, it directly returns true. If it is a number (if *isnumericval* returns true), true is also returned and lastly, if it is a *TmRecord*, it checks if all elements of the record are values. If any of these fail, it returns false.

· **val eval1 : Syntax.context → Syntax.term → Syntax.term = <fun>**

*eval1* evaluates the term passed as an argument on the context one step. This evaluation depends on the term type. See *Syntax* for a more detailed explanation of each term. If no rules apply, a *NoRuleApplies* exception is thrown. The rules are as follows:

→ **TmIf**: of type ***TmIf of Support.Error.info * term1 * term2 * term3***. If *term1* type is *TmTrue*, *term2* is returned. If *term1* type is *TmFalse*, *term3* is returned. If *term1* is a term still to be evaluated, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmIf* is returned with the term *term1* evaluated.

**→TmVar:** of type ***TmVar of Support.Error.info * int1 * int2.*** First, the method *getbinding* is called and the context and the binding position on the context are passed. *int1* stands for said position. Once we get the binding associated, if said binding's type is *TmAbbBind of term,* term *term* is returned. Otherwise, *NoRuleApplies* exception is raised.

**→ TmApp:** of type ***TmApp of Support.Error.info * term1 * term2.*** If *term1* is of type *TmAbs of Support.Error.info * string * term,* and *term2* is a value; this is, when the method *isval* is called and *term2* is passed as argument and *TmTrue* is returned, then *Syntax.termSubstTop v2 t12*is called**.** If *term1* is a value, this is, when *isval* is called and *term1* is passed as argument and true is returned, *eval1* is called recursively and *term2* is passed as argument to be evaluated, and *TmApp* is returned with its term *term2* evaluated.Otherwise, if *term1* is not either an abstraction or a value, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmApp* is returned with its term *term1* evaluated.

**→ TmRecord:** of type ***TmRecord of Support.Error.info * (string * term) list.*** An element whose type is not a value is searched for on the list and it is evaluated. To do so, *eval1* is called recursively and term *term* is passed as argument.The *TmRecord* will be returned with *term* evaluated.If there are no terms to be evaluated, *NoRuleApplies* exception is raised.

**→ TmProj:** of type ***TmProj of Support.Error.info * term * string.*** If *term* is of type *TmRecord of Support.Error.info * (string * term) list,* and all the elements of the record are values; this is, when passing the record to *isval* method as argument *TmTrue* is returned, then the term of the list whose label equals *string* is returned. To do so, *List.assoc* is used (See *List* module in the OCaml library for more info). If *term* is not of type *TmRecord, eval1* is called recursively and *term* is passed as argument to be evaluated, and *TmProj* is returned with the term *term* evaluated.

**→ TmTimesfloat:** of type ***TmTimesfloat of Support.Error.info * term1 * term2***. If *term1* and *term2* are both floats, the product is returned. If *term1* is a float but *term2* is not, *eval1* is called recursively and *term2* is passed as argument to be evaluated, and the *TmTimesfloat* is returned with the term *term2* evaluated. Otherwise, if *term1* is not a float, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmTimesfloat* is returned with the term *term1* evaluated.

**→ TmSucc:** of type ***TmSucc of Support.Error.info * term.****eval1* is called recursively and *term* is passed as argument to be evaluated.*TmSucc* is returned with term *term* evaluated.

**→ TmPred:** of type ***TmPred of Support.error.info * term1.*** If *term1* is of type *TmZero*, a *TmZero * Support.Error.info* is returned where *Support.Error.info = UNKNOWN*.If *term1* is of type *TmSucc of Support.Error.Info* term2,* and *term2* is a number, this is, when passed to *isnumericval* as argument *TmTrue* is returned, *term2* is returned.If *term1* is not one of said types, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmPred* is returned with its term *term1* evaluated.

→ **TmIsZero:** of **type *TmIsZero of Support.Error.info * term1***. If *term1* is of type *TmZero,* a *TmTrue of Support.Error.info* where *Support.Error.info = UNKNOWN* is returned. If *term1* is of type *TmSucc of Support.Error.info * term2,* and *term2* is a number, this is, when passed to *isnumericval* as argument *TmTrue* is returned, a *TmFalse of Support.Error.info* where *Support.Error.info = UNKNOWN* is returned.If *term1* is not one of said types, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmIsZero* is returned with its term *term1* evaluated.

→ **TmLet:** of type ***TmLet of Support.Error.info * string * term1 * term2.*** If *term1* is a value; this is, when *isval* is called and *term1* is passed as argument *TmTrue* is returned, then *Syntax.termSubstTop v1 t2* is called.If term1is not a value, *eval1* is called recursively and *term1* is passed as argument to be evaluated, and *TmLet* is returned with its term *term1* evaluated.

· **val eval : Syntax.context → Syntax.term → Syntax.term = <fun>**

*eval* evaluates the term passed as an argument on the given context and returns a new term as result that cannot be evaluated anymore, a value. To do so, *eval* calls *eval1* recursively applying steps until a *NoRuleApplies* exception is caught. When that happens, the last final term is returned.

Here is where debug is checked. If it is true, each iteration that tries to evaluate something that is not a value will first print the term to be evaluated.

· **val evalBinding : Syntax.context → Syntax.binding → Syntax.binding = <fun>**

*evalBinding* evaluates the term that the binding contains. To do so, it calls *eval* and passes the context and term. If the binding is “typed” *binding*, that same binding is returned.

# 6 – Syntax.ml

**Summary:** The *syntax* module defines the Lambda-Calculus syntax that the program recognises, with all the types of terms the program recognises.

## Types:

- **Terms:**

  They are declared as elements of type term. All of them have a Support.Error.info associated with them. They are:

  · **TmTrue of info:** Represents Boolean true.

  · **TmFalse of info:** Represents Boolean false.

  · **TmIf of info * term * term * term:** Represents an If-then-else clause. First term is the condition, second the term following the then and third the one following the else.

  · **TmVar of info * int * int:**Represents a variable. The first integer is the index the variable had in its context, and the second, said context's length.

  · **TmAbstraction of info * string * term:** Represents an abstraction of the symbol named by the string in the term.

  · **TmApp of info * term * term:** Represents an application of terms.

  · **TmRecord of info * (string * term) list:** Represents a record as a list of field names and content

  · **TmProj of info * term * string:** Represents a projection of a field.

  · **TmFloat of info * float:** Represents a floating point number.

  · **TmTimesfloat of info * term * term:** Represents the product of two floats.

  · **TmString of info * string:** Represents a string literal.

  · **TmZero of info:** Represents zero.

  · **TmSucc of info * term:** Represents successor (next, +1).

  · **TmPred of info * term:** Represents predecessor (previous, -1)

  · **TmIsZero of info * term:** Represents an IsZero check.

  · **TmLet of info * string * term * term:** Represents a "let = in" clause.

- **Context:**

  The Lambda-Calculus context is modelled by the type *Context*. This is a *(string * binding) list*, a list of pairs where each element is a string representing the symbol that stands for the binding, and the binding itself.

- **Binding:**

  Bindings can be created between symbols and terms or only to initialise a symbol without it being associated to a term (x/;). A binding between a symbol and a term is typed ***TmAbbBind of term***. A standalone binding of a symbol is typed ***NameBind***.

- **Command:**

  These represent the commands recognised by the program and to be executed. A command can either be an evaluation or a binding. A command representing an evaluation is an ***Eval of info * term***. A command representing a binding is of type ***Bind of info * string * binding***. Where the string is the symbol that stands for the binding.

## Functions:

- ### Context Management:

  · **val ctxlength : 'a list → int = <fun>**

*ctxlength* returns the length of the list passed as an argument. This works with any list, it just assumes it is a context.

· **val addbinding : ('a * 'b) list → 'a → 'b → ('a * 'b) list = <fun>**

*addbinding* is used to add a binding and its symbol to a context.

**· val addname : ('a * binding) list → 'a →  ('a * binding) list = <fun>**

*addname* receives a context and a symbol of type string. It calls *addbinding* to add a binding to the given context with the provided symbol and *NameBind* as a binding.

**· val isnamebound : ('a * 'b) list → 'a → bool = <fun>**

*isnamebound* receives a context and a symbol, it then recursively searches if the symbol has already been used for a binding on the given context. If it has, it returns true.

**· val pickfreshname : (string * binding) list → string → (string * binding) list * string = <fun>**

*pickfreshname* receives a context and a symbol. It first checks if the symbol is already bound. If it is, it concatenates an apostrophe (‘) at the end of the symbol and tries again. If the symbol isn’t bound, this function returns a pair of elements where the first one is the new context with the new element added and the second one is the added symbol.

**· val index2name : Support.Error.info → ('a * 'b) list → int → 'a = <fun>**

*index2name* attempts to return the symbol at the given position of the context. To do this, it calls *List.nth*. If a *Failure* exception is caught, it calls error with a message.

**· val name2index : Support.Error.info → (string * 'a) list → string → int = <fun>**

*name2index* attempts to return the index where the given symbol is located in the context. If it cannot find it, it calls error with a corresponding message.

**· val getbinding : Support.Error.info → ('a * binding) list → int → binding = <fun>**

*getbinding* returns the binding of passed context on passed index position. If it finds it, it performs a shift with its position + 1. If it cannot find it, it calls error with a corresponding message.

- **Shifting:**

· **val termShift : int→term→term = <fun>**

*termShift* is used when needing to shift terms around, it takes an integer and a term and calls *termShiftAbove int 0 term*.

· **val termShiftAbove : int→ int →term→term = <fun>**

*termShift*Above defines an onvar function to describe what to do when finding a variable. Then it calls tmmap with said function, its second integer and the term.

· **val tmmap : (Support.Error.info →int→ int → int→ term) → int → term → term = <fun>**

*tmmap* defines behaviour when shifting each of the term "types". To do this, it creates a recursive function *walk* that iterates one step each time. Depending on the term:

· **TmTrue:**it just returns the term.
· **TmFalse:**It just returns the term.
· **TmIf:**It walks inside the condition, the "then" term and the "else" term.
· **TmVar:**It calls the onvar function and shifts the variable's index and length.
· **TmAbstraction:**it walks into the term while increasing the shifting counter by 1.
· **TmApp:**It walks into both terms.
· **TmRecord:** It applies a *List.map* to the list of fields with a function that walks into each field's term.
· **TmProj:**It walks inside the projection's term.
· **TmFloat:**It just returns the term.
· **TmTimesfloat:**It walks into each term.
· **TmString:** It just returns the term.
· **TmZero:**it just returns the term.
· **TmSucc:**It walks into the successor content.
· **TmPred:**It walks into the predecessor content.
· **TmIsZero:**It walks inside the IsZero term.
· **TmLet:** It walks inside the first term and increases counter before walking into the second term.

· **val bindingshift : int→ binding → binding =<fun>**

*bindingshift* just checks what kind of binding it receives, if it is a *NameBind*, it returns it as is, if it is a *TmAbbBind*, it calls termShift on its content.

- **Substitution:**

· **val termSubstTop : term → term → term =<fun>**

*termSubstTop* calls three functions to perform the sifting needed for the substitution. Taking two terms ***S*** and ***T***, it returns ***termShift***(-1) (***termSubst*** 0 (**termShift** 1 ***S***) ***T***). Shifting first over ***S***, then performing the substitution and then shifting back again over the result.

· **val termSubst : int → term → term → term =<fun>**

*termSubst* is similar to termShiftAbove in that it defines an onvar function and calls tmmap with it. In this case the onvar function calls *termShift* if the variable symbol is in the shifting position.

- **Extracting file info:**

· **val tmInfo: term → Support.Error.info =<fun>**

*tmInfo* simply returns the "fileinfo" (filename, line and column) of a given term.

- **Printing:**

Here the program defines a set of directives and functions to allow OCaml to prettyprint each of the term types. These mostly consist of recursive functions that match a term ‘type’ and go deeper until they find a value they can print.

As an interesting fact, there was an error in the original implementation’s printing functions. When printing a float number product, the original program printed twice the second float and never the first one. It would print timesfloat 2.0 3.0 as timesfloat 3. 3. .

# 7 – Support.ml

**Summary:** The *Support* module is used to define error information storage and printing. It consists of two modules: *Error* and *Pervasive*. The first one defines all error behaviour and structure. The second one just defines two abbreviations for *Error.info* and *Printf.printf*, namely *info* and *pr*.

**Types:**

· **Exit of int**: Exception associated to an int to exit the program. Zero represents that the program exited with no errors. Any other small int means an error has occurred.

· **info**: It represents a term's "file position". Its value can be either *FI of string * int1 * int2* or *UNKNOWN*. In the first case, *string* is the filename, *int1* is the line number and *int2* is the character's position within the line.

· **'a withinfo**: A convenient datatype for "value with file info". It is used in *Lexer* and *Parser* for tokens that also have an inherent value.

· **dummyinfo**: Placeholder value for *UNKNOWN info* types.

**Functions:**

· **val createInfo : string → int → int → info = <fun>**

*createInfo* behaves as an *info* constructor. It receives a filename, column and character position and creates an according *info*.

· **val errf : (unit → unit) → 'a** =**<fun>**

*errf* executes the function passed as an argument to print an error message and also follows some pretty-printing directives. It the raises *Exit 1*.

· **val printInfo : info → unit = <fun>**

If the *info*'s passed argument value is not *UNKNOWN*, the name of the file, line and character position are printed. Otherwise "<Unknown file and line>:" is printed.

· **val errfAt : info → (unit → unit) → 'a = <fun>**

*errfAt* is the same as *errf* but it also calls *printInfo* to print the information.

· **val err : string → 'a= <fun>**

*err* calls *errf* to print an error message which is represented by the string passed as argument.

· **val error : info → string → 'a = <fun>**

This is the same as *errfAt* but in addition, it prints an error message represented by the string passed as an argument.

· **val error1 : string → unit = <fun>**

It prints a warning message represented by the string passed as argument but, unlike the error functions, it does not fail afterwards.

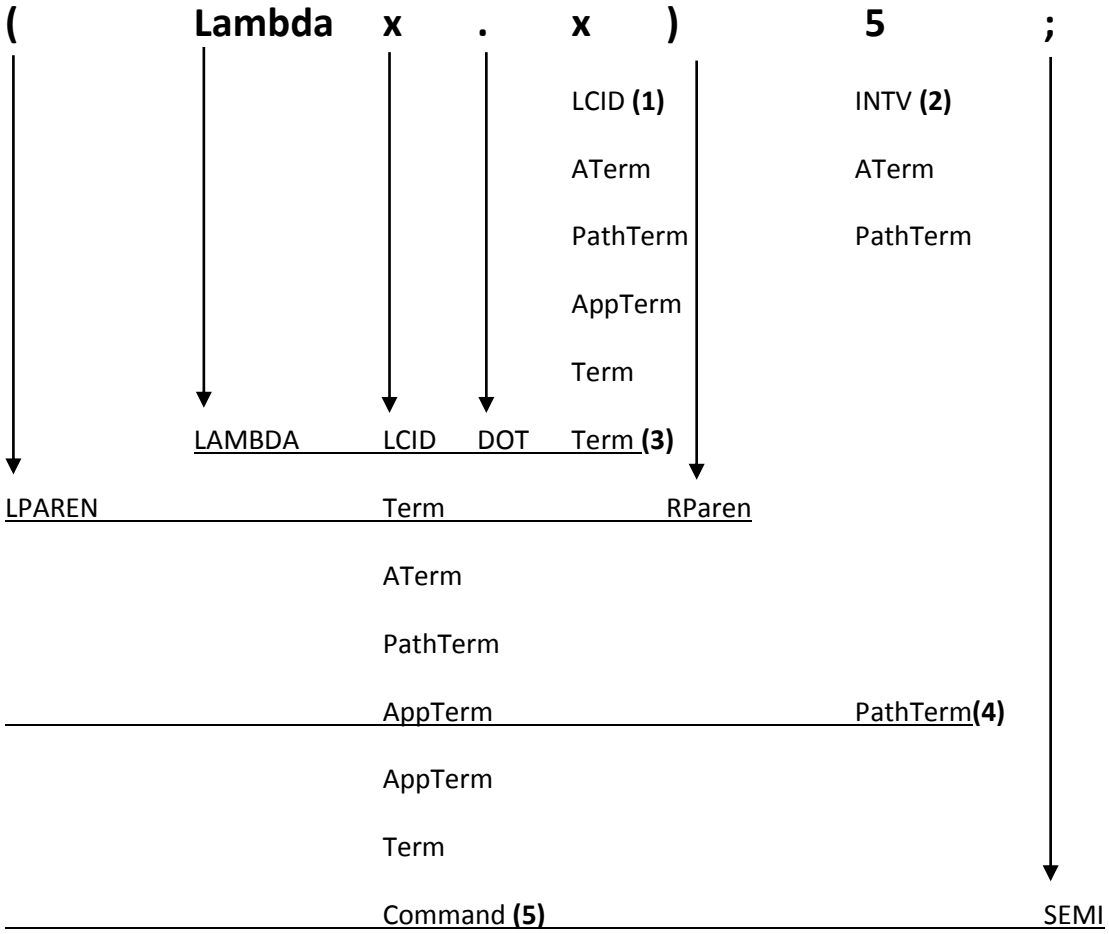
· **val error1At : info → string → unit = <fun>**

This is the same as error1 but it calls *printInfo* to print the location information.

# Practical example:

Here we explain how a command is processed internally by the program from the parsing phase until it is ready to print.

**Parsing**




This is the complete parsing tree of our command, parenthesised numbers indicate something happens in those steps, the rest are simple returns.

**1** – First, a lowercase unreserved word is recognised as an atomic term (variable). Thus, it is returned as a function that accepts a context and returns a TmVar with the word's information, index in said context and length of the context.

**2** – “5” Is matched as an atomic term (integer). So it is returned as a function that takes a context and returns TmSucc(TmSucc(TmSucc(TmSucc(TmSucc(TmZero()))))) (or simply 5) with its information.

**3** – Lambda x . x Is recognised as an abstraction. This will return another function that takes a context and returns something. This function will take the context and add the “first” x to it, creating “ctx1”. Then it returns a TmAbs with the location information, the “first” x’s name and the “Term’s” (our “last” x) application to this ctx1. So it applies **(1)**’s returned function to a context with the added variable.

This can be understood as TmAbs(**Lambda’s info**, **x1’s name**, TmVar(**x2’s info**, **x2’s index ctx1, ctx1’s length**)). Or rather, a function that takes a context and returns all that.

**4** – When the whole thing is recognised as an application, another “take a context return something complicated” function is created. This function will take the context and apply the returned functions from both the abstraction and the number five to it. Theformer will get the context in which to insert the “x” and create the TmVar while the latter will just return the TmSucc succession.

In summary, this returns a

```
ctx → TmApp(TmAbs’ info,
                    TmAbs( Lambda’s info, x1’s name,
                                        TmVar(x2’s info, x2’s index ctx1, ctx1’s length)
                    ),
                    TmSucc(
                            TmSucc(
                                    TmSucc(
                                            TmSucc(
                                                    TmSucc(
                                                            TmSucc(
                                                                    TmZero()
                                                            )
                                                    )
                                            )
                                    )
                            )
                    )
            )
```

**5** – When this term is recognised as a command, it is returned as a function that takes a context and returns an Eval command with its information and passes said context to its function. This function is added to the list of command-functions on toplevel.

# Evaluating

Once our command-function has been added to the list and returned to main.ml, it is processed using the eval function of core.ml. This eval function basically applies the eval1 function until no rule applies to it. When eval1 receives a TmApp containing an abstraction and a value (which matches our case), it calls syntax.ml's termSubstTop with two arguments. The first one is the value (5's TmSuccs) and the second one is TmAbs' TmVar, the one that was created in the first step of our parsing: TmVar(**x2's info**, **x2's index in ctx1, ctx1's length**). Remember, x2 is the second x in λx.x and ctx1 was a context to which x was added. We'll call the TmSucc chain "5" and TmVar TmVar(…).

What termSubstTop does is kind of tricky, as it calls:

**termShift** (-1) (**termSubst** 0 (**termShift** 1 **"5"**) **TmVar(…)**)

· **termShift d t:** termShift just calls **termShiftAboved** 0 **t**.

· **termShiftAbove d c t**: This creates a function **fun fi c x n** that checks if x is bigger than c, if it is, it returns a TmVar where the position of the term in context has been increased by d, as well as the length of the context increased by d. If it is not, it returns a TmVar where only the context's length has been increased by d. We'll call this function **onvar**. After creating it, termShiftAbove calls **tmmaponvar c t**.

· **tmmap onvar c t:** This function defines a recursive inner function called **walk c t**. What walk does is check the different types of term types **t** can be. We'll focus on our first case, 5. Our five is a TmSucc succession, when walk finds that, it keeps "walking" into the inner successors until it finds TmZero, which it returns as itself, effectively returning the TmSucc as is. tmmap has a behaviour defined for each term type to find, so walk could find something different inside the successors, and behave differently.

Now let's check how all this joins together starting by (**termShift** 1 "5").

**termShift 1 "5"** →**termShiftAbove 1 0 "5"** → **tmmap (onvar with d = 1) 0 "5"** →
**walk 0 TmSucc(...)** → **TmSucc(fi, walk 0 TmSucc(…))** – 4 times-> **TmSucc(…TmZero())**.

Our termShift has become "5" again, this leaves us with:

**termShift** (-1) (**termSubst** 0 **"5"TmVar(…)**)

· **termSubst j s t**: This function is similar to termShiftAbove, in that it creates an **onvar**function and calls tmmap with it. termSubst's **onvar fi c x n**checks if x equals j + c. If it does, it calls **termShift c s**, if it doesn't, it returns a **TmVar(fi, x n)**. This onvar function is passed to tmmap with 0 and t.

Let's see how **termSubst** behaves with our inputs:

**termSubst 0 "5" TmVar(…)** → **tmmap (onvar with j = 0 and s = TmVar(…)) 0 "5"**→
**walk 0 TmSucc(...)** → **TmSucc(fi, walk 0 TmSucc(…))** – 4 times-> **TmSucc(…TmZero())**

Which returns "5" again. Leaving us with

**termShift -1 "5"** →**termShiftAbove -1 0 "5"** → **tmmap (onvar with d = -1) 0 "5"** →
**walk 0 TmSucc(...)** → **TmSucc(fi, walk 0 TmSucc(…))** – 4 times-> **TmSucc(…TmZero())**.

This "5" will get returned to core.ml's eval1. As stated before, eval1 will keep recursively applying rules until it can't do it anymore. So with a TmSucc succession, it will get in the inner TmSuccs until it finds a TmZero. As there is no rule for TmZero, it will raise a NoRuleApplies exception. This rule is caught in eval, returning the "5", which is received by main.ml where it is printed as a result.

Had something more complex been used, more of the rules of evaluation could have been showcased. Let's see another example:

**TmIf(**info, **TmTrue(_)**, **TmLet(**
info,**"**word", **TmZero(_)**, **TmisZero(TmVar(**"word"**))**
**),**
TmSucc(_,TmZero(_))
**)**

Something like "if true then let word = 0 in iszero word else 1". (Take into account, each TmVar contains information on the word and location in context, and that context includes the word, as per parser rules).

When reaching eval1's recursion, **TmIf**(_, **TmTrue**, t2, t3) gets matched to a true if, and so t2, our **TmLet** is returned to the recursion. A **TmLet** containing info, word, value (**TmZero** in our case, term (**isZero(TmVar(**"word"**))** in our case); calls **termSubstTop value term**.

**termSubstTop value term** →**termShift** -1 (**termSubst** 0 (**termShift** 1 **value**) **term**)

**termShift** 1 **TmZero(_)** will just become **TmZero(_)** as seen before, leaving us with:

**termShift** -1 (**termSubst** 0 **TmZeroterm**)

Next, let's look at **termSubst**:

**termSubst** 0 **TmZeroTmIsZero(TmVar(_**, **word's index in context**, **context length))**→
**tmmap(onvar with j = 0 and s = TmZero(_))** 0 **TmIsZero(**…) →
**walk** 0 **TmIsZero(**…**)** →**TmIsZero(_**, **walk** 0 **TmVar(_**, **index**, **length))** →
**TmIsZero(_**, **onvar** _ 0 **index length**).

As stated before, **termSubst**'s **onvar fi c x n**calls another termShift with **TmZero**, returning itself to **TmIsZero**, leaving us with

**termShift** -1 **TmIsZero(**_,**TmZero(**dummyinfo**))** →
**termShiftAbove** -1 0 **TmIsZero(**_,**TmZero(**dummyinfo**))** →
**tmmap (onvar with d = -1 and c = 0) 0 TmIsZero(**_,**TmZero(**dummyinfo**))** →
**walk 0 TmIsZero(**_,**TmZero(**dummyinfo**))**→
**TmIsZero(**_, **walk 0 TmZero(**dummyinfo**))** →
**TmIsZero(**_, **TmZero(**dummyinfo**))**

This **TmIsZero** goes back into **eval1** where it is matched and returned as a **TmTrue** to be printed on main.ml.

# Final considerations:

There is still a lot of functionality to add to this program that couldn't be done because of time constraints.

Some other ideas we considered were:

Allowing the user to input more than one line per command, to ease input format and not force the user to define complex commands in a single depth.

A better printing system that shows terms in a clearer or more readable way.

An option to load a file from the toplevel command line to pre-define functions and then interact with them.

Apart from that, the implementation works mostly as intended. It is really sub-optimal for recursive calculations (factorial(7) takes seconds to process), but the results are correct.