DPL

(P3)

Yeray Méndez Romero

yeray.mendez@udc.es

Daniel Rivera López

d.rivera1@udc.es

Index:

Changes to the original code:

In order to build the top level, we needed to make some changes to the original implementation:

1- Changes to ‘parseArgs’:
The error that happened when no input was provided has been changed to return a value ‘Empty’ that will be later used to start the top level.

```
match !inFile with
    None -> err "You must specify an input file"
  | Some(s) -> s
```

```
match !inFile with
    None ->
      print_string("You did not specify an input file\n");
      print_string("Starting iterative mode ...\n");
      "Empty"
 | Some(s) -> s
```

2- Changes to ‘main’:
Now the ‘inFile’ in ‘main’ is matched to detect “Empty” and start the top level.

```
let main () =
  let inFile = parseArgs() in
  let _ = process_file inFile emptycontext in
  ()
```

```
let main () =
  let inFile = parseArgs() in
  match inFile with
  "Empty"->
      shell false emptycontext
  |_->
      let _ = process_file inFile "" emptycontext in
      ()
```

3- New function ‘shell’ added:
This new function starts the top level and process each line wrote by the user.

```
let shell value ctx=
  print_string("\n*****************************************");
  print_string("\nWelcome to Iterative Mode");
  print_string("\n*****************************************\n");
  print_string("\n-----------------------------------------\n");
  print_string("Choose option:\n");
  print_string("Write a expresion to load.\n");
  print_string("Write 'exit' to end iterative mode.\n");
  let rec shell_aux value ctx2=
    let ctx_aux = ref ctx2 in
  if not value  then begin
    print_newline();
    let line = read_line() in
    match line with
    "exit"->
      shell_aux true ctx2
    |line->
    (
    try
      ctx_aux := process_file "" line ctx2;
    with
      e -> ()
      );
      print_string("\n-----------------------------------------\n");
      shell_aux false !ctx_aux
  end
  else
    print_string("Going out of iterative mode....\n");
  in shell_aux value ctx;;
```

4- The function ‘process_file’ now receives a new parameter:
This new parameter ‘line’ represents each line that the user writes on the top level.

```
let process_file f ctx =
  alreadyImported := f :: !alreadyImported;
  let cmds,_ = parseFile f ctx in
```

```
let process_file f line ctx =
  alreadyImported := f :: !alreadyImported;
  let cmds,_ = parseFile f line ctx in
```

5- Changes to ‘parseFile’:
Now receives the new parameter ‘line’ and use it when the ‘inFile’ does not contain anything to build a ‘lexbuf’ with it and send it to the Parser.

```
let parseFile inFile =
  let pi = openfile inFile
  in let lexbuf = Lexer.create inFile pi
  in let result =
    try Parser.toplevel Lexer.main lexbuf with Parsing.Parse_error ->
    error (Lexer.info lexbuf) "Parse error"
in
  Parsing.clear_parser(); close_in pi; result
```

```
let parseFile inFile line= match inFile with
  ""->
    let lexbuf = Lexing.from_string line
    in let result =
    try Parser.toplevel Lexer.main lexbuf with Parsing.Parse_error ->
    error (Lexer.info lexbuf) "Parse error"
    in
      Parsing.clear_parser(); result
  |_->
  let pi = openfile inFile
  in let lexbuf = Lexer.create inFile pi
  in let result =
    try Parser.toplevel Lexer.main lexbuf with Parsing.Parse_error ->
    error (Lexer.info lexbuf) "Parse error"
in
  Parsing.clear_parser(); close_in pi; result
```

The top level:

-User manual:

To start the top level just execute the program with no arguments.

To end the top level write ‘exit’ on the standard input and press Enter.

Inside the top level you can write lambda expressions on the standard input and press Enter to make the program process them.

The reserved words are:

(* Words *)

"if", "then", "else", "true", "false", "lambda"
,"timesfloat”, "succ", "pred", "iszero", "let", "in".

Aside the usual ones, here is an explanation on their use:

- “lambda”: defines the start of a function.

- “timesfloat”: function that multiplies two float numbers. *1

- “succ”: function that calculates the next number of the given one. *1

- “pred”: function that returns the previous number of the given one. *1

- “iszero”: function that checks if the value given is zero or not.

- “let” and “in”: allow to define bindings in only an expression.

    Example: let a = 1 in a; output: 1

    *1 : These functions can be called with anything that the parser allows, but if you do that the program won’t be able to evaluate them and will return exactly the input.

(* Symbols *)

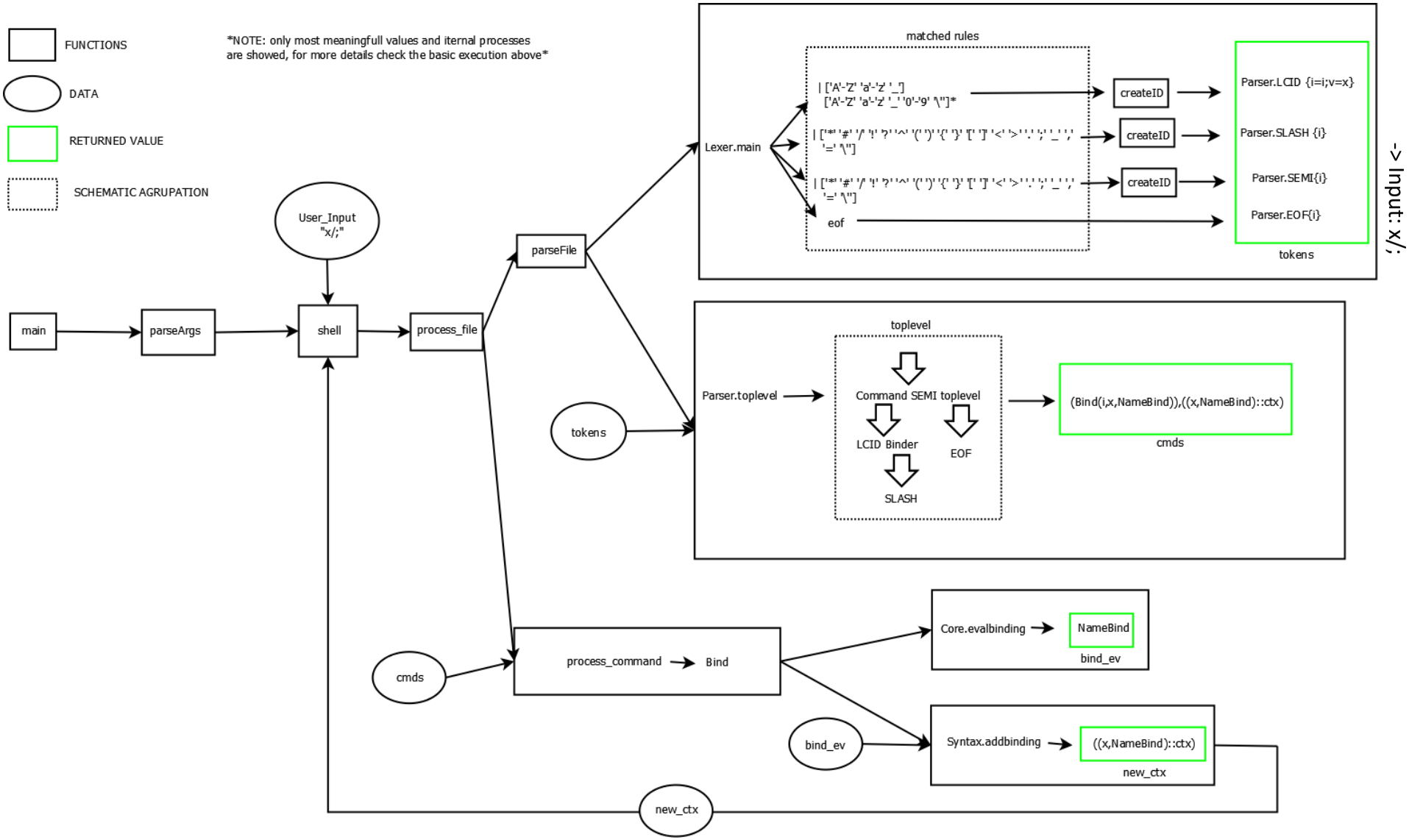
There is a lot of reserved symbols too, but here will only be shown the basic ones, for more details check the files “lexer.mll” and for all the possible ways to combine them check “parser.mly”.

- “;”: each command must end with one.
- “/”: used when declaring a variable without value
- All usual lambda calculous symbols will work as expected.

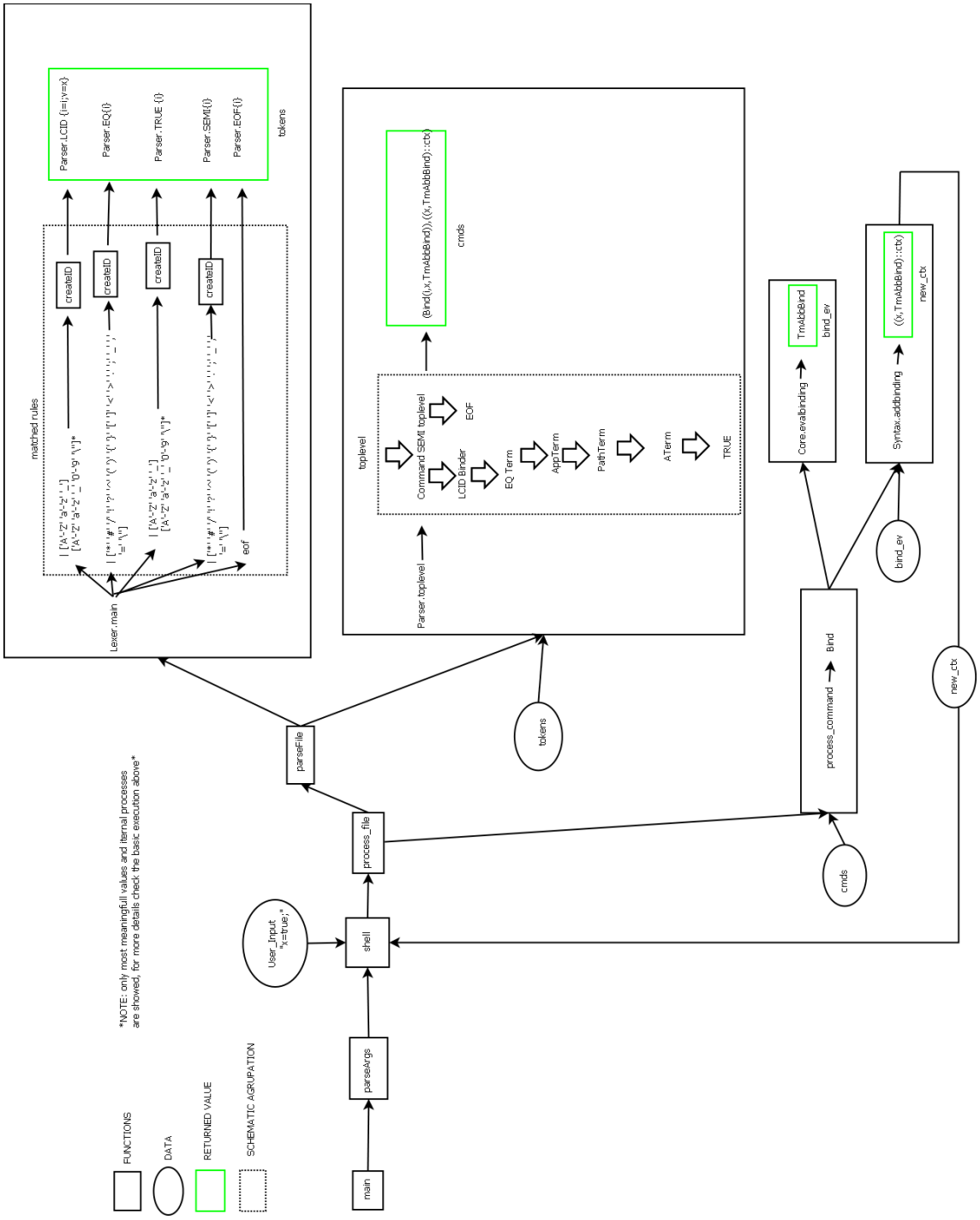
-Execution examples:

-> Basic execution:

The program starts on the “main” function and calls “parseArgs”, because the program has been launched without parameters “parseArgs” returns ‘Empty’ witch indicates to the “main” function to start the top level by calling the function “shell”. The function “shell” receives the user expression from the standard input and calls “process_file” with the user expression, “process_file” calls to “parseFile” where the “Lexer” is called and the user input is converted into tokens that return to “parseFile” to be the input in the “Parser” call. In the “Parser” the tokens will be compared with the grammar and a list of commands in addition to a context list will be returned. At this point the program flow returns to “process_file” and calls “process_command” with each command that returns from “parseFile”. Inside “process_command” Binding and Evaluation commands are evaluated, the result from the evaluation is printed to the user and a new context is returned all the way to the “shell” function where will be stored for its use with the next user input.


FUNCTIONS
DATA
RETURNED VALUE
SCHEMATIC AGRUPATION
NOTE: only most meaningfull values and iternal processes are showed, for more details check the basic execution above
User_Input "x/;"
main
parseArgs
shell
process_file
parseFile
Lexer.main
matched rules
createID
createID
createID
eof
Parser.LCID {i=i;v=x}
Parser.SLASH {i}
Parser.SEMI{i}
Parser.EOF{i}
tokens
-> Input: x/;
tokens
Parser.toplevel
toplevel
Command SEMI toplevel
LCID Binder
EOF
SLASH
(Bind(i,x,NameBind)),((x,NameBind)::ctx)
cmds
cmds
process_command → Bind
Core.evalbinding
NameBind
bind_ev
bind_ev
Syntax.addbinding
((x,NameBind)::ctx)
new_ctx
new_ctx

-> Input: x = true;


main
parseArgs
shell
User_Input "x=true;"
process_file
parseFile
tokens
Lexer.main
matched rules
createID
createID
createID
createID
eof
Parser.LCID {l=i;v=x}
Parser.EQ{}
Parser.TRUE {}
Parser.SEMI{}
Parser.EOF{}
tokens
Parser.toplevel
toplevel
Command.SEMI toplevel
EOF
LCID Binder
EQ Term
AppTerm
PathTerm
ATerm
TRUE
(Bind(i,x,TmAbbBind)),((x,TmAbbBind)::ctx)
cmds
process_command
Bind
cmds
Core.evalbinding
TmAbbBind
bind_ev
Syntax.addbinding
((x,TmAbbBind)::ctx)
new_ctx
bind_ev
new_ctx
FUNCTIONS
DATA
RETURNED VALUE
SCHEMATIC AGRUPATION
NOTE: only most meaningfull values and iternal processes are showed, for more details check the basic execution above

-> Input: x; (after executing x/;)