Interface for module Pascal

1. Arbres de syntaxe abstraite du Pascal.

```
type typ =
  | Tinteger
  | Tbool

type variable = string list × typ

type constant =
  | Cinteger of int
  | Cbool of bool

type binop =
  | Add | Sub | Mult | Div | Mod | Eq | Noteq | Gt | Ge | Lt | Le | And | Or

type unop =
  | Minus | Not

type expr =
  | Econstant of constant
  | Evar of string
  | Ecall of string × expr list
  | Ebinop of expr × binop × expr
  | Eunop of unop × expr

type instr =
  | Iassign of string × expr
  | Icond of expr × block × block
  | Icall of string × expr list

and block = instr list

type param = variable

type proc_fun =
  | Procedure of string × param list × variable list × block
  | Function of string × param list × typ × variable list × block

type program = string × variable list × proc_fun list × block
```

Module Lexer (Lex)

2. Analyse lexicale.

```
{
  open Lexing
  open Parser
```

```
open Pascal
```

3. Numéro de la ligne courante.

```
let line = ref 1
let get_line () = !line
```

4. Erreurs lexicales.

```
exception Illegal_character
exception Unterminated_comment
```

5. Table des mots clés.

```
let keywords = Hashtbl.create 97
let _ = List.iter
          (fun (s, k) → Hashtbl.add keywords s k)
          [ "if", IF; "then", THEN; "else", ELSE; "program", PROGRAM;
            "procedure", PROCEDURE; "function", FUNCTION;
            "var", VAR; "begin", BEGIN; "end", END;
            "integer", INTEGER; "bool", BOOL;
            "mod", MOD; "and", AND; "or", OR; "not", NOT;
            "true", CONSTANT (Cbool true); "false", CONSTANT (Cbool false)
          ]
let ident_or_keyword s =
  let s = String.lowercase s in
  try Hashtbl.find keywords s with Not_found → IDENT s
}
```

6. Expressions régulières.

```
let letter = ['a'–'z' 'A'–'Z']
let digit = ['0'–'9']
let ident = letter (letter | digit)*
let integer = ['0'–'9']+
let space = [' ' '\t']
```

7. Point d'entrée du lexeur.

```
rule token = parse
  | space+
     { token lexbuf }
  | '\n'
     { incr line; token lexbuf }
  | "(*" | '{'
```

```
  { comment lexbuf; token lexbuf }
| ident
  { ident_or_keyword (lexeme lexbuf) }
| ';'
  { SEMICOLON }
| '.'
  { DOT }
| ','
  { COMMA }
| ':'
  { COLON }
| ":="
  { ASSIGN }
| '+'
  { PLUS }
| '-'
  { MINUS }
| '*'
  { TIMES }
| '/'
  { DIV }
| '='
  { EQ }
| "<>"
  { NOTEQ }
| '<'
  { LT }
| "<="
  { LE }
| '>'
  { GT }
| ">="
  { GE }
| '('
  { LPAREN }
| ')'
  { RPAREN }
| integer
  { CONSTANT (Cinteger (int_of_string (lexeme lexbuf))) }
| eof
  { EOF }
| _
  { raise Illegal_character }
```

8. Commentaires.

```
and comment = parse
  | "*)" | '}'
    { () }
  | '\n'
    { incr line; comment lexbuf }
  | eof
    { raise Unterminated_comment }
  | _
    { comment lexbuf }
```

Module Parser (Yacc)

9. Analyse syntaxique.

```
  open Pascal
```

10. Déclarations des tokens.

```
%token <Pascal.constant> CONSTANT
%token <string> IDENT
%token EOF PROGRAM SEMICOLON DOT VAR COLON COMMA ASSIGN
%token PROCEDURE FUNCTION
%token INTEGER BOOL LPAREN RPAREN BEGIN END LBRAC RBRAC
%token IF THEN ELSE PLUS MINUS TIMES DIV MOD NOT
%token EQ NOTEQ LT LE GT GE AND OR
```

11. Précédences.

```
%nonassoc THEN
%nonassoc ELSE
%left OR
%left AND
%nonassoc NOT
%nonassoc EQ NOTEQ LT LE GT GE
%left PLUS MINUS
%left TIMES DIV MOD
%nonassoc uminus
```

12. Point d'entrée.

```
%type <Pascal.program> program
%start program
```

13. Règles de grammaire.

program ::=
| *PROGRAM IDENT SEMICOLON variables proc_and_funs*
  BEGIN instrs END DOT EOF
  { ($2, $4, $5, $7) }

variables ::=
| /∗ epsilon ∗/ { [] }
| *VAR var_decls SEMICOLON* { $2 }

var_decls ::=
| *var_decl* { [$1] }
| *var_decls SEMICOLON var_decl* { $1 @ [$3] }

var_decl ::=
| *idents COLON typ* { ($1, $3) }

idents ::=
| *IDENT* { [$1] }
| *IDENT COMMA idents* { $1 :: $3 }

typ ::=
| *INTEGER* { *Tinteger* }
| *BOOL* { *Tbool* }

proc_and_funs ::=
| /∗ epsilon ∗/ { [] }
| *proc_and_fun SEMICOLON proc_and_funs* { $1 :: $3 }

proc_and_fun ::=
| *PROCEDURE IDENT LPAREN var_decls RPAREN SEMICOLON*
  variables BEGIN instrs END
  { *Procedure* ($2, $4, $7, $9) }
| *FUNCTION IDENT LPAREN var_decls RPAREN COLON typ SEMICOLON*
  variables BEGIN instrs END
  { *Function* ($2, $4, $7, $9, $11) }

block ::=
| *instr* { [$1] }
| *BEGIN instrs END* { $2 }

instrs ::=
| *instr SEMICOLON* { [$1] }
| *instr SEMICOLON instrs* { $1 :: $3 }

instr ::=
| *IDENT ASSIGN expr* { *Iassign* ($1, $3) }
| *IDENT LPAREN exprs RPAREN* { *Icall* ($1, $3) }
| *IF expr THEN block* { *Icond* ($2, $4, []) }
| *IF expr THEN block ELSE block* { *Icond* ($2, $4, $6) }

exprs ::=
| /* epsilon */ { [] }
| *expr* { [$1] }
| *expr COMMA exprs* { $1 :: $3 }

expr ::=
| *CONSTANT* { *Econstant* $1 }
| *IDENT* { *Evar* $1 }
| *IDENT LPAREN exprs RPAREN* { *Ecall* ($1, $3) }
| *expr PLUS expr* { *Ebinop* ($1, *Add*, $3) }
| *expr MINUS expr* { *Ebinop* ($1, *Sub*, $3) }
| *expr TIMES expr* { *Ebinop* ($1, *Mult*, $3) }
| *expr DIV expr* { *Ebinop* ($1, *Div*, $3) }
| *expr MOD expr* { *Ebinop* ($1, *Mod*, $3) }
| *expr EQ expr* { *Ebinop* ($1, *Eq*, $3) }
| *expr NOTEQ expr* { *Ebinop* ($1, *Noteq*, $3) }
| *expr LT expr* { *Ebinop* ($1, *Lt*, $3) }
| *expr LE expr* { *Ebinop* ($1, *Le*, $3) }
| *expr GT expr* { *Ebinop* ($1, *Gt*, $3) }
| *expr GE expr* { *Ebinop* ($1, *Ge*, $3) }
| *expr AND expr* { *Ebinop* ($1, *And*, $3) }
| *expr OR expr* { *Ebinop* ($1, *Or*, $3) }
| *LPAREN expr RPAREN* { $2 }
| *NOT expr* { *Eunop* (*Not*, $2) }
| *MINUS expr* %prec *uminus* { *Eunop* (*Minus*, $2) }

Interface for module Compile

14. Le formatteur sur lequel on écrit le code C.

```
val fmt : Format.formatter ref
```

15. La fonction de compilation.

```
val program : Pascal.program → unit
```

Module Compile

16. Production du code C.

```
open Format
open Pascal

let fmt = ref std_formatter
let print s = fprintf !fmt s
```

17. Types.

```
let string_of_typ = function
  | Tinteger → "int"
  | Tbool → "int"
```

18. Liste d'identificateurs, séparés par des virgules.

```
let rec idents = function
  | [] → assert false
  | [x] → print "%s" x
  | x :: l → print "%s,␣" x; idents l
```

19. Déclarations de variables.

```
let vars bl =
  List.iter
    (fun (vl, t) →
       print "%s␣" (string_of_typ t); idents vl;
       print ";@\n")
    bl
```

20. Paramètres de fonctions ou procédures.

```
let rec param t = function
  | [] → assert false
  | [x] → print "%s␣%s" (string_of_typ t) x
  | x :: l → print "%s␣%s,␣" (string_of_typ t) x; param t l

let rec params = function
  | [] →
      ()
  | (vl, t) :: l →
```

```
    param t vl;
    if l ≠ [] then print ", ";
    params l
```

21. Opérateurs binaires et unaires.

```
let string_of_binop = function
  | Add → "+"
  | Sub → "-"
  | Mult → "*"
  | Div → "/"
  | Mod → "%"
  | Eq → "=="
  | Noteq → "!="
  | Gt → ">"
  | Ge → ">="
  | Lt → "<"
  | Le → "<="
  | And → "&&"
  | Or → "||"

let string_of_unop = function
  | Minus → "-"
  | Not → "!"
```

22. Expressions.

```
let rec expr = function
  | Econstant (Cinteger n) →
      print "%d" n
  | Econstant (Cbool b) →
      print "%b" b
  | Evar x →
      print "%s" x
  | Ecall (x, el) →
      print "%s(" x; exprs el; print ")"
  | Ebinop (e1, op, e2) →
      print "(";
      expr e1; print " %s " (string_of_binop op);
      expr e2; print ")"
  | Eunop (op, e) →
      print "(%s" (string_of_unop op);
      expr e; print ")"

and exprs = function
  | [] → ()
  | [e] → expr e
```

```
  | e :: el → expr e; print ",␣"; exprs el
```

23. Instructions.

```
let rec instr f = function
  | Iassign (x, e) when Some x = f →
      print "_%s␣=␣" x; expr e; print ";"
  | Iassign (x, e) →
      print "%s␣=␣" x; expr e; print ";"
  | Icond (e, st, []) →
      print "if␣("; expr e;
      print ")␣{@\n␣␣@["; statement f st;
      print "@]@\n};"
  | Icond (e, st, st') →
      print "if␣("; expr e;
      print ")␣{@\n␣␣@["; statement f st;
      print "@]@\n}␣else␣{@\n␣␣@["; statement f st';
      print "@]@\n};"
  | Icall (x, [e]) when x = "writeln" →
      print "printf(\"%%d\\n\",␣"; expr e; print ");"
  | Icall (x, el) →
      print "%s(" x; exprs el; print ");"
and statement f = function
  | [] → ()
  | [i] → instr f i
  | i :: il → instr f i; print "@\n"; statement f il
```

24. Déclarations de fonctions et procédures.

```
let proc_and_fun = function
  | Procedure (p, pa, vl, st) →
      print "void␣%s(" p;
      params pa;
      print ")␣{@\n␣␣@[";
      vars vl;
      print "@\n";
      statement None st;
      print "@]@\n}@\n@\n"
  | Function (f, pa, t, vl, st) →
      print "%s␣%s(" (string_of_typ t) f;
      params pa;
      print ")␣{@\n␣␣@[";
      print "%s␣_%s;@\n" (string_of_typ t) f;
      vars vl;
      print "@\n";
      statement (Some f) st;
```

```
    print "@\nreturn␣-%s;" f;
    print "@]@\n}@\n@\n"
let proc_and_funs = List.iter proc_and_fun
```

25. Programme.

```
let program (id, vl, pl, st) =
  print "/*␣program␣%s␣*/@\n@\n" id;
  print "#define␣true␣1@\n";
  print "#define␣false␣0@\n@\n";
  vars vl;
  print "@\n";
  proc_and_funs pl;
  print "int␣main()␣{@\n␣␣@[";
  statement None st;
  print "@\nexit(0);@]@\n}@\n";
  pp_print_flush !fmt ()
```

Module Main

26. Programme principal.

```
open Printf
```

27. Utilisation.

```
let usage () =
  eprintf "usage:␣p2c␣file.pas\n";
  exit 1
```

28. Fichier à compiler.

```
let file =
  if Array.length Sys.argv = 1 then usage ();
  let f = Sys.argv.(1) in
  if ¬ (Filename.check_suffix f ".pas") then usage ();
  if ¬ (Sys.file_exists f) then begin
    eprintf "p2c:␣%s:␣no␣such␣file\n" f; exit 1
  end;
  f
```

29. Affichage des erreurs.

```
let report_error e =
  eprintf "%s:␣%d:␣" file (Lexer.get_line ());
  match e with
```

```
  | Lexer.Illegal_character ->
      eprintf "illegal␣character\n"
  | Lexer.Unterminated_comment ->
      eprintf "unterminated␣comment\n"
  | Parsing.Parse_error ->
      eprintf "syntax␣error\n"
  | _ ->
      eprintf "anomaly:␣%s\n" (Printexc.to_string e)
```

30. Ouverture du fichier, analyse syntaxique puis compilation.

```
let main () =
  try
    let c = open_in file in
    let lb = Lexing.from_channel c in
    let p = Parser.program Lexer.token lb in
    close_in c;
    let cfile = (Filename.chop_suffix file ".pas") ^ ".c" in
    let c = open_out cfile in
    Compile.fmt := Format.formatter_of_out_channel c;
    Compile.program p;
    close_out c
  with e ->
    report_error e; exit 1

let _ = Printexc.catch main ()
```

Index