

Université de Rouen
IUP GMI de Rouen
M1GMI 2006/2007

Compilation

M. PATROU

Projet : **mini ptoc**

LAURENT JILIBERT

Rendu le 26 janvier 2007

Introduction

A propos de ce projet

Dans le cadre du cours de compilation de Master 1 GMI, Il nous a été demandé de développer un langage impératif et de construire autour un compilateur qui partant d'un programme source, génère un programme cible en C. Ce dernier devant être compilable et exécutable.

Je commencerais par détailler le projet, avant de donner le code de l'analyseur lexical et de l'analyseur sémantique. Enfin, je vous montrerais quelques exemples afin de pouvoir conclure sur la réussite ou l'échec de ce projet.

A propos des sources

L'ensemble de la composition a été développé à la main sans reprise de code existant à l'aide de Flex, Bison, C pour le projet et L^AT_EX pour la rédaction du rapport. J'ai essentiellement fait des recherches sur Internet et consulté l'excellent Lex & Yacc d'Oreilly¹.

Comment utiliser le programme

Pour utiliser ce programme tapez simplement :

```
mp2c -f fichier.mp
```

Pour connaître les options tapez :

```
mp2c -help
```

¹Lex & Yacc 2nd edition, John R. Levine, Tony Mason, Doug Brown

1 Détail du projet

Lorsque le sujet nous a été donné, il m'a fallu savoir ce qu'était un langage impératif et je me suis rendu que c'était le cas du Pascal. Je suis donc parti sur un langage qui est très proche du pascal puisqu'il lui empreinte la syntaxe et les mots clés pour la description du langage.

1.1 Description du langage et analyseur lexical

Le premier point à traiter pour implémenter ce langage était de comprendre la syntaxe pascal afin de pouvoir en faire une analyse correcte. Il m'a fallu aussi trouver l'ensemble des mots clé correspondants aux éléments décrits dans le sujet. On obtient donc la liste suivante :

- VAR, CONST : des variables et des constantes
- INTEGER, REAL, BOOLEAN, ARRAY : les types de variables
- +, -, DIV, MOD, ... : les opérateurs arithmétiques
- >, <, =, <>, ... : les opérateurs logiques
- := : l'affectation de valeur
- READ(LN), WRITE(LN) : les accesseurs en lecture et en écriture
- IF, WHILE, FOR, ... : les structures de controles (imbriquables)
- RECORD, TYPE : les pointeurs et les types de structure.
- plus tous les mots de déclaration, les commentaires ...

Une fois que le travail de recensement des mots est effectué, il devient possible de commencer l'écriture de l'analyseur lexical. Afin de proposer un langage ouvert, j'ai choisi de ne pas tenir compte de la casse des mots et il m'a fallu définir pour chaque lettre de l'alphabet une règle lexicale. Par exemple, pour la première lettre de l'alphabet on obtient la règle "A [aA]". Lorsque la création des règles pour chaque lettre, et chaque symbole, a été effectuée, on peut commencer à prévoir les différentes actions qu'elles vont engendrées. Pour la plupart des règles, il ne s'agira que d'envoyer à l'analyseur sémantique un message expliquant le mot reconnu. Mais pour d'autres, il va être nécessaire de copier la valeur scannée. C'est à dire que lorsque je vais lire une affectation "a := 2", je vais demander au scanner de mémoriser la valeur 2 afin de la transmettre à mon lexer de telle sorte que dans la source générée je retrouve "a=2".

Malgré tout, la partie la plus dure à mettre en place dans l'analyse lexical concerne la préparation des commentaires et la mise en place des tabulations. Comme dans tout programme en flex, il est souvent utilisé du code C et c'est donc ce que j'ai utilisé pour "préparer" les différents types de commentaires mais aussi les String passées en paramètres dans l'instruction d'écriture sur la sortie standard. J'ai donc du définir des fonctions permettant de faire du découpage de chaine de caractères.

1.2 Analyseur sémantique

Maintenant qu'on a à peu près fini l'analyseur lexical, on peut se pencher sur la partie sémantique. La première chose qu'il faut faire est de construire une grammaire qui reconnaisse l'ensemble des notes définis dans l'analyseur lexical en évitant les conflits, les réductions. De plus, il nous faut aussi définir une structure bison qui nous permet de faire la liaison entre les valeurs sauvegardées lors du scannage du source et la source bison. Pour ce faire on définit une "union" se compose de chaine de caractères, d'entiers, de réels.

Viens ensuite le problème de la création de la grammaire. Pour ce faire, j'ai utilisé une table de conversion des principales instructions du langage Pascal vers le langage C² afin de savoir ce que devait réellement faire les instructions Pascal ainsi que des schémas de traduction³ qui m'a permis d'écrire ma grammaire de base correctement.

²http://turbo-pascal.developpez.com/tutoriel/conversion_pascal_c

³<http://ina.eivd.ch/publications/coursPascal/Annexes/AnnexeG.htm>

Une fois que le squelette de l'analyseur a été posé, j'ai pu commencer à mettre en place les différentes règles de translation vers du langage C. Le principe est le même que pour la calculatrice ; pour éviter d'avoir une notation inversé il faut traiter le plus tôt. Par exemple, pour avoir une conditionnelle avec if parfaitement structurée, il n'est pas possible de traiter toutes les informations en même temps car sinon on se retrouve en notation inversée avec le contenu de la condition. Il faudra donc le noter de la façon suivante :

"IF code boolean_expr THEN code if_body" est la bonne notation.

La dernière partie à implémenter afin de respecter l'énoncé doit se tourner vers le fichier de sortie. C'est-à-dire qu'il va falloir mettre tout les traitements effectués dans un fichier. Pour cette dernière opération, j'ai pris en compte le nom du programme contenu dans le fichier passé en paramètre de l'application car c'est ce qui correspond au main en C. Ce fichier devra posséder l'extension "*.mp" ou "*.MP" pour pouvoir être converti, sinon un message d'erreur sera retourné.

1.3 Parties non-traitées

Il n'y a qu'une seule grosse partie que je n'ai pas réussi à traiter et elle concerne l'indentation du code généré. En effet, bien qu'il soit aisé de récupérer le nombre de tabulations courantes, il m'est paru bien compliqué de les redistribuer au bon endroit. Cela n'affecte que la lisibilité du code car la source générée est compilable et exécutable.

2 Les codes sources

2.1 L'analyseur lexical : pascal.l

```
%{
    #include <stdio.h>
    #include "pascal.h"
    int line_no = 1;

    char *prepare(char *str);
    char *prepareFirComment(char *str);
    char *prepareSecComment(char *str);
}%}

A [aA]
B [bB]
C [cC]
D [dD]
E [eE]
F [fF]
G [gG]
H [hH]
I [iI]
J [jJ]
K [kK]
L [lL]
M [mM]
N [nN]
O [oO]
P [pP]
Q [qQ]
R [rR]
S [sS]
T [tT]
U [uU]
V [vV]
W [wW]
X [xX]
Y [yY]
Z [zZ]

NQUOTE [^']
C1 [^{}]
C2 [^\*|\*[^\\]]
%%

{T}{Y}{P}{E} return(TYPE);
{V}{A}{R} return(VAR);
{C}{O}{N}{S}{T} return(CONST);
{A}{R}{R}{A}{Y} return(ARRAY);
{B}{O}{O}{L}{E}{A}{N} return(BOOLEAN);
{I}{N}{T}{E}{G}{E}{R} return(INTEGER);
{R}{E}{A}{L}{D}{E}{C} return(REALDEC);
```

```

{0}{F} return(OF);
[0-9]+ {yyval.ival = atoi(yytext);return(INT);}
[0-9]+ "." [0-9]+ {yyval.fval = atof(yytext);return(REAL);}
'({NQUOTE}|'')+ '{strcpy(yyval.sval, prepare(yytext));return(String);}
":" return(COLON);
"," return(COMMA);
"[" return(LBRAC);
"(" return(LPAREN);
"]" return(RBRAC);
")" return(RPAREN);
".." return(DOTDOT);

"=" return(EQUAL);
">=" return(GE);
">" return(GT);
"<=" return(LE);
"<>" return(NOTEQUAL);
"<" return(LT);
{N}{I}{L} return(NIL);
{N}{O}{T} return(NOT);
{O}{R} return(OR);
{A}{N}{D} return(AND);
{T}{R}{U}{E} return(TRUE);
{F}{A}{L}{S}{E} return(FALSE);

"- " return(MINUS);
"+" return(PLUS);
"/" return(SLASH);
"*" return(STAR);
{D}{I}{V} return(DIV);
{M}{O}{D} return(MOD);
{I}{N}{C} return(INC);
{D}{E}{C} return(DEC);

":=" return(ASSIGNMENT);

{W}{R}{I}{T}{E} return(WRITE);
{W}{R}{I}{T}{E}{L}{N} return(WRITELN);
{R}{E}{A}{D} return(READ);
{R}{E}{A}{D}{L}{N} return(READLN);

{C}{A}{S}{E} return(CASE);
{D}{O} return(DO);
{E}{L}{S}{E} return(ELSE);
{F}{O}{R} return(FOR);
{I}{F} return(IF);
{R}{E}{P}{E}{A}{T} return(REPEAT);
{T}{H}{E}{N} return(THEN);
{T}{O} return(TO);

```

```
{U}{N}{T}{I}{L} return(UNTIL);
{W}{H}{I}{L}{E} return(WHILE);
```

```
"->" | "^" return(UPARROW);
{R}{E}{C}{O}{R}{D} return(RECORD);
{W}{I}{T}{H} return(WITH);
```

```
{B}{E}{G}{I}{N} return(PBEGIN);
{E}{N}{D} return(END);
{F}{U}{N}{C}{T}{I}{O}{N} return(FUNCTION);
{P}{R}{O}{C}{E}{D}{U}{R}{E} return(PROCEDURE);
{P}{R}{O}{G}{R}{A}{M} return(PROGRAM);
";" return(SEMICOLON);
"." return(DOT);
```

```
[a-zA-Z]([a-zA-Z0-9])* {strcpy(yylval.sval, yytext);return(IDENTIFIER);}
```

```
"**" return(STARSTAR);
```

```
"*({C2})*" {strcpy(yylval.sval, prepareFirComment(yytext));return(COMMENT);}
"\"({C1})\"" {strcpy(yylval.sval, prepareSecComment(yytext));return(COMMENT);}
```

```
\t
[ \t\f] ;
\n ;
. {printf("' %c' (0%o): le caractère l'est po connu %d\n",yytext[0], yytext[0], line_no);}
```

```
%%
yywrap () {
    return (1);
}
```

```
char *prepare(char *str) {
    char *tmp = (char*)malloc((strlen(str)-2)*sizeof(char));
    int debut = 1;
    memcpy(tmp, (char*)str+debut, strlen(str)-2);
    return tmp;
}
```

```
char *prepareFirComment(char *str) {
    char *tmp = (char*)malloc((strlen(str)-4)*sizeof(char));
    int debut = 2;
    memcpy(tmp, (char*)str+debut, strlen(str)-4);
    return tmp;
}
```

```
char *prepareSecComment(char *str) {
    char *tmp = (char*)malloc((strlen(str)-2)*sizeof(char));
    int debut = 1;
    memcpy(tmp, (char*)str+debut, strlen(str)-2);
```

```

    return tmp;
}

```

2.2 L'analyseur sémantique : pascal.y

```

%{
    #include<stdio.h>
    #include<stdlib.h>
    #include<string.h>
    #include<stdarg.h>

    /* La structure qui permet de sauvegarder toutes les variables déclarées*/
    struct MyStruct {
        char name[32];
        char type[32];
        struct MyStruct *suivant;
    };

    struct MyStruct *varstruct = NULL;

    /*Permet de se faire des opérations dans la structure*/
    char *varType(char *str);
    int addVar(char *name, char *type);

    /*Variable de masse utilisée dans le travail en profondeur*/
    char assign_var[128];

    /*Le fichier de sortie avec son nom que l'on ouvre avec la dite fonction*/
    FILE *file;
    char filename[32];
    int testExtension(char *str);
    int openFile(const char *name);

    /*Ecriture dans le fichier de sortie et écriture des tabulations*/
    int writeOUT(const char *format, ...);
    int writeDEC();
    int decal=0;
    int state = 1;
}%
%union {
    char sval[128];
    int ival;
    double fval;
}

%token TYPE VAR CONST ARRAY BOOLEAN INTEGER OF DIGSEQ REALNUMBER
%token COLON COMMA LBRAC LPAREN RBRAC RPAREN DOTDOT
%token EQUAL GE GT LE NOTEQUAL LT NIL NOT OR AND UPARROW RECORD WITH
%token MINUS PLUS SLASH STAR DIV MOD ASSIGNMENT WRITELN READLN CASE DO
%token ELSE FOR IF REPEAT THEN TO UNTIL WHILE END PBEGIN INC DEC
%token FUNCTION PROCEDURE PROGRAM SEMICOLON DOT STARSTAR READ WRITE
%token FALSE TRUE REALDEC

%token <sval> IDENTIFIER

```



```

%token <ival> INT
%token <sval> STRING
%token <sval> COMMENT
%token <fval> REAL

%start Input

%%

Input :
    header body
    ;

header :
    title semicolon declare
    ;

title :
    PROGRAM IDENTIFIER {openFile($2);writeOUT("int main() {\n");}
    | PROGRAM IDENTIFIER LPAREN identifier_list RPAREN {openFile($2);writeOUT("int main(int argc,
        char *argv[]) {\n");}
    ;

declare:
    const_def /*type_def*/ var_def
    ;

const_def:
    CONST const_list
    | ;

const_list:
    const_decl const_list
    | const_decl
    ;

const_decl:
    IDENTIFIER EQUAL INT SEMICOLON {writeOUT("#define %s %d\n", $1, $3);}
    | IDENTIFIER EQUAL MINUS INT SEMICOLON {writeOUT("#define %s -%d\n", $1, $4);}
    | IDENTIFIER EQUAL REAL SEMICOLON {writeOUT("#define %s %f\n", $1, $3);}
    | IDENTIFIER EQUAL MINUS REAL SEMICOLON {writeOUT("#define %s -%f\n", $1, $4);}
    | IDENTIFIER EQUAL STRING SEMICOLON {writeOUT("#define %s \"%s\"\n", $1, $3);}
    | IDENTIFIER EQUAL TRUE SEMICOLON {writeOUT("#define %s true\n", $1);}
    | IDENTIFIER EQUAL FALSE SEMICOLON {writeOUT("#define %s true\n", $1);}
    ;

var_def:
    VAR var_list
    | ;

var_list:
    var_decl var_list
    | var_decl
    ;

```

```

var_decl:
    IDENTIFIER COLON var_decl_type semicolon {writeOUT("%s;\n", $1);addVar($1,assign_var);
        strcpy(assign_var,"");}
    ;

var_decl_type:
    INTEGER {writeOUT("int ");strcpy(assign_var,"int");}
    | REALDEC {writeOUT("double ");strcpy(assign_var,"double");}
    | BOOLEAN {writeOUT("boolean ");strcpy(assign_var,"boolean");}
    | ARRAY LBRAC INT DOTDOT INT RBRAC OF INTEGER {writeOUT("int [%d] ",$5);
        strcpy(assign_var,"array");}
    ;

/* BODY OF PROGRAM */
body:
    PBEGIN statements END dot
    ;

/* LES DIFFERENTES TACHES POSSIBLES */
statements:
    statement statements
    | statement
    ;

statement:
    if_stmt
    | while_stmt
    | for_stmt
    | repeat_stmt
    | write_stmt
    | writeln_stmt
    | read_stmt
    | assign_stmt
    | ccrement_stmt
    | comment_stmt
    ;

/* IF */
if_stmt:
    IF {writeOUT("if(");} boolean_expr THEN {writeOUT(")");} if_body
    ;

if_body:
    PBEGIN {writeOUT("{\n");} statements END semicolon {writeOUT("}\n");}
    | {writeOUT("{\n");} statement {writeOUT("}\n");}
    ;

/* EXPRESSION DE TEST */
boolean_expr:
    boolean_expr AND {writeOUT(" && ");} boolean_expr
    | boolean_expr OR {writeOUT(" || ");} boolean_expr
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} EQUAL {writeOUT("== ");}

```

```

        boolean_term
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} GE {writeOUT(" >= ");}
        boolean_term
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} GT {writeOUT(" > ");}
        boolean_term
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} LE {writeOUT(" <= ");}
        boolean_term
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} NOTEQUAL
        {writeOUT(" != ");} boolean_term
    | expr {writeOUT("%s ", assign_var);strcpy(assign_var, "");} LT {writeOUT(" < ");}
        boolean_term
    ;

boolean_term:
    TRUE {writeOUT("true ");}
    | FALSE {writeOUT("false ");}
    | IDENTIFIER {writeOUT("%s ", $1);}
    | INT {writeOUT("%d ", $1);}
    | REAL {writeOUT("%f ", $1);}
    ;

/* WHILE */
while_stmt:
    WHILE {writeOUT("while(");} boolean_expr DO {writeOUT(")");} while_body
    ;

while_body:
    PBEGIN {writeOUT("{\n");} statements END semicolon {writeOUT("}\n");}
    | {writeOUT("{\n");} statement {writeOUT("}\n");}

/* FOR */
for_stmt:
    FOR IDENTIFIER ASSIGNMENT INT TO INT DO {
        writeOUT("for(%s = %d ; %s <= %d ; %s++) ", $2, $4, $2, $6, $2);} for_body
    | FOR IDENTIFIER ASSIGNMENT REAL TO REAL DO {
        writeOUT("for(%s = %f ; %s <= %f ; %s++) ", $2, $4, $2, $6, $2);} for_body
    ;

for_body:
    PBEGIN {writeOUT("{\n");} statements END SEMICOLON {writeOUT("}\n");}
    | {writeOUT("{\n");} statement {writeOUT("}\n");}

/* ASSIGNATION DE VARIABLES */
assign_stmt:
    IDENTIFIER ASSIGNMENT expr SEMICOLON {writeOUT("%s = %s;\n", $1, assign_var);
        strcpy(assign_var, "");}
    ;

expr:
    expr PLUS {strcat(assign_var, "+ ");} term
    | expr MINUS {strcat(assign_var, "- ");} term
    | term
    ;

```

```

term:
    term STAR {strcat(assign_var,"* ");} factor
    | term SLASH {strcat(assign_var,"/ ");} factor
    | term MOD {strcat(assign_var,"% ");} factor
    | term DIV {strcat(assign_var,"/ ");} factor
    | factor
    ;

factor:
    LPAREN {strcat(assign_var,"(");} expr RPAREN {strcat(assign_var,")");}
    | TRUE {strcat(assign_var,"true");}
    | FALSE {strcat(assign_var,"false");}
    | IDENTIFIER {sprintf(assign_var, "%s%s ", assign_var, $1);}
    | INT {sprintf(assign_var, "%s%d ", assign_var, $1);}
    | REAL {sprintf(assign_var, "%s%f ", assign_var, $1);}
    ;

/* INCREMENTATION & DECREMENTATION*/
crement_stmt:
    DEC LPAREN IDENTIFIER RPAREN semicolon {writeOUT("%s--;\n", $3);}
    | DEC LPAREN IDENTIFIER COMMA IDENTIFIER RPAREN semicolon {writeOUT("%s -= %s ;\n", $3, $5);}
    | INC LPAREN IDENTIFIER RPAREN semicolon {writeOUT("%s++;\n", $3);}
    | INC LPAREN IDENTIFIER COMMA IDENTIFIER RPAREN semicolon {writeOUT("%s += %s ;\n", $3, $5);}
    ;

/* WRITE */
write_stmt:
    WRITE LPAREN {writeOUT("printf(\"");} write_expr RPAREN semicolon
    ;

write_expr:
    IDENTIFIER {writeOUT("%s\", %s);\n", varType($1), $1);}
    | STRING COMMA IDENTIFIER {writeOUT("%s %s\", %s);\n", $1, varType($3), $3);}
    | STRING {writeOUT("%s\");\n", $1);}
    ;

/* WRITELN */
writeln_stmt:
    WRITELN LPAREN writeln_expr RPAREN semicolon
    ;

writeln_expr:
    IDENTIFIER {writeOUT("printf(\"%s\\n\", %s);\n", varType($1), $1);}
    | STRING COMMA IDENTIFIER {writeOUT("printf(\"%s %s\\n\", %s);\n", $1, varType($3), $3);}
    | STRING {writeOUT("printf(\"%s\\n\");\n", $1);}
    ;

/* READ & READLN */
read_stmt:
    READ LPAREN IDENTIFIER RPAREN SEMICOLON {writeOUT("scanf(\"%s\",&%s);\n", varType($3), $3);}
    | READLN LPAREN IDENTIFIER RPAREN SEMICOLON
        {writeOUT("scanf(\"%s\\n\",&%s);\n", varType($3), $3);}
    ;

```

```

/* REPEAT */
repeat_stmt:
    REPEAT {writeOUT("do{\n");} statements UNTIL {writeOUT("}while(");} boolean_expr SEMICOLON
        {writeOUT(");\n");}
    ;

/* COMMENTAIRE */
comment_stmt:
    COMMENT {writeOUT("/%s*\n",$1);}
    ;

/* DIVERS BAZAR */
identifier_list :
    identifier_list comma identifier
    | identifier
    ;

identifier :
    IDENTIFIER
    ;

semicolon :
    SEMICOLON
    ;

comma :
    COMMA
    ;

dot:
    DOT {writeOUT("return 1;\n");fclose(file);}
    ;

%%
int yyerror(char *s) {
    state = -1;
    printf("%s\n",s);
}

int writeOUT(const char *format, ...) {
    int valeur;
    writeDEC();

    va_list ap;

    va_start(ap, format);
    valeur = vfprintf(file, format, ap);
    va_end(ap);

    return valeur;
}

int writeDEC() {
    int i=0;

```

```

        for(i ; i<dec1-1 ; i++) {
            fprintf(file, "\t");
        }

        return 1;
    }

int openFile(const char *name) {
    sprintf(filename,"%s.c", name);
    file = fopen(filename,"w+");

    if(file == NULL) {
        return -1;
    }

    writeOUT("#include<stdio.h>\n");
    writeOUT("#include<stdlib.h>\n");
    writeOUT("#include<string.h>\n\n");
    writeOUT("typedef enum {false, true} boolean;\n\n");

    return 1;
}

char *varType(char *str) {
    struct MyStruct *actuel = varstruct;

    while( strcmp(actuel->name,str) && actuel->suivant != NULL ) {
        actuel = actuel->suivant;
    }

    if(!strcmp(actuel->type,"int")) {
        return "%d";
    } else if(!strcmp(actuel->type,"double")) {
        return "%f";
    } else if(!strcmp(actuel->type,"boolean")) {
        return "%d";
    } else if(!strcmp(actuel->type,"array")) {
        return "%d";
    } else {
        return "%s";
    }
}

int addVar(char *name, char *type) {
    struct MyStruct *tmp = (struct MyStruct*)malloc(sizeof(struct MyStruct));

    if( varstruct != NULL ) {
        strcpy(tmp->name, name);
        strcpy(tmp->type, type);
        tmp->suivant = varstruct;

        varstruct = tmp;
    } else {

```

```

        varstruct = (struct MyStruct*)malloc(32 * sizeof(struct MyStruct));
        strcpy(varstruct->name, name);
        strcpy(varstruct->type, type);
        varstruct->suivant = NULL;
    }
}

int testExtension(char *chaine) {
    char *result = (char*)malloc(3*sizeof(char));
    int debut = strlen(chaine) - 3;
    memcpy (result, (char *)chaine+debut, strlen(chaine));

    if(strcasecmp(result, ".mp") == 0)
        return 1;

    return -1;
}

extern FILE *yyin;

int main(int argc, char *argv[]) {
    char infile[128];
    state = 1;
    if(argc < 3) {
        if(argc == 2) {
            if( strcmp(argv[1], "-help") == 0 ) {
                printf("Usage:  %s [-f FILE] | [OPTIONS] \n",argv[0]);
                printf("Options : \n");
                printf("-f      : specifie un fichier special-pascal test.mp\n");
                printf("-help   : affiche ce message d'aide\n");
                printf("-t      : affiche des infos complémentaires\n");
                printf("-v      : affiche les informations sur l'application\n\n");

                return -1;
            } else if( strcmp(argv[1], "-v") == 0 ) {
                printf("MyP2C a été réalisé, lors d'un projet de compilation de Master 1 à l'IUP");
                printf(" GMI de Rouen, par Laurent Jilibert (2007).\n");

                return -1;
            }
        }
        printf("Usage:  %s [-f FILE] | [OPTIONS] \n",argv[0]);
        printf("Pour en savoir davantage, faites: « mp2c -help »\n");
        return -1;
    } else if( strcmp(argv[1], "-f") == 0 ){
        if(testExtension(argv[2]) == -1) {
            printf("Conversion cancelled\n");
            printf("Please use file with *.mp extension\n");
            return -1;
        }
        sprintf(infile,"%s",argv[2]);
        yyin = fopen(infile,"r");
        yyparse();
        fclose(yyin);
    }
}

```

```

        if( state == 1 ) {
            printf("Conversion succeed\n");
            printf("Input : %s\n",infile);
            printf("Output: %s\n",filename);
        } else {
            printf("Conversion failed\n");
            remove(filename);
        }
    } else {
        printf("Usage:  %s [-f FILE] | [OPTIONS] \n",argv[0]);
        printf("Pour en savoir davantage, faites: « mp2c -help »\n");
        return -1;
    }
}

```


3 Quelques exemples

Avant de conclure, je vais présenter un exemple de code complexe qu'il est possible de passer au mini-compilateur mp2c. L'exemple est volontairement incohérent au niveau des déclarations de variables. Il s'agit simplement de démontrer que le compilateur fonctionne dans le seul but de convertir un format vers un autre en supposant que la source est correcte.

3.1 La source mini-pascal

```
program test;
const
  k = 2;
  c = 'variable c';
  d = true;
var
  b : boolean;
  i : integer;
  s : array [1..10] of integer;
  e : real;
begin
  write(i);
  a := 5 + (3 * (6 / 9));
  for i := 0 TO 10 do
  begin
    writeln(e);
    while i <> 0 do readln(b);
  end;

  while i=3.3 and i<= 4 do
  begin
    writeln(e);
  end;

  repeat
    writeln(i);
    write('toto');
    for i := 0 TO 10 do
    begin
      writeln(i);
    end;
  until i < 5;

  if i=6 then
  begin
    if d=true then read(i);
  end;
end.
```

3.2 Le code généré⁴

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

⁴J'ai indenté le code généré pour le rendre plus lisible et comparable à la source

```

typedef enum {false, true} boolean;

int main() {
#define b 2
#define c "variable c"
#define d true
    boolean b;
    int i;
    int [10] s;
    double e;

    printf("%d", i);
    a = 5 + (3 * (6 / 9 ));
    for(i = 0 ; i <= 10 ; i++) {
        printf("%f\n", e);
        while(i != 0 ){
            scanf("%d\n",&b);
        }
    }

    while(i == 3.300000 && i <= 4 ){
        printf("%f\n", e);
    }

    do{
        printf("%d\n", i);
        printf("toto");
        for(i = 0 ; i <= 10 ; i++) {
            printf("%d\n", i);
        }
    }while(i < 5 );

    if(i == 6 ){
        if(d == true ){
            scanf("%d",&i);
        }
    }
    return 1;
}

```

Conclusion

Dans l'ensemble, le programme marche assez bien puisqu'il respecte en général la spécification énoncée dans le sujet. Cependant, il reste améliorable dans bien des points, notamment dans celui de l'indentation et de la lisibilité du code. Avec un peu plus de temps, il m'aurait été possible de réparer ce problème et de pousser plus loin dans le développement de fonctions supplémentaires. Il aurait été possible, je pense, de rajouter les notions de procédure et de fonction, mais pour ce faire il aurait fallu changer tout le début de la grammaire.