Universidad de Murcia

Compiladores

PCEO

Traducción de miniPascal a código ensamblador de MIPS

Autores:
Pablo Miralles González
Jose Antonio Lorencio Abril

Profesor:
Eduardo Martínez Gracia

Convocatoria Junio 2019/20

Índice

1. Introducción

En esta memoria vamos a detallar el trabajo que hemos realizado para implementar el compilador de miniPascal.

Hemos decidido organizar el documento agrupando las distintas etapas de la implementación en apartados, explicando la función y desarrollo de cada una de ellas, así como dando ejemplos del funcionamiento de las mismas. Consideramos que de esta forma la explicación queda muy clara en lo que respecta a las distintas partes que tiene un compilador, y no sacrifica el entendimiento global del mismo.

2. Manual de usuario para la ejecución

Veamos primero la estructura del proyecto que presentamos:

Código 1: Fichero

```
.
|-- Makefile
|-- docker
|   |-- Dockerfile
|   `-- Makefile
|-- src
|   |-- code_gen.c
|   |-- code_gen.h
|   |-- listaCodigo.c
|   |-- listaCodigo.h
|   |-- listaSimbolos.c
|   |-- listaSimbolos.h
|   |-- main.c
|   |-- minipascal.l
|   |-- minipascal.y
|   |-- semantic_analysis.c
|   `-- semantic_analysis.h
`-- unit_testing
    |-- codigo
    |   |-- test_cod1.mp
    |   `-- test_cod1_salida.txt
    |-- funciones
    |   |-- test_f1.mp
    |   |-- test_f10.mp
    |   |-- ...
    |-- lexico
    |   |-- test_lex1.mp
    |   |-- test_lex1_out.txt
    |   |-- ...
    |-- semantico
    |   |-- test_sem1.mp
    |   |-- test_sem1_out.txt
    |   |-- ...
    `-- sintactico
        |-- test_sint1.mp
        `-- test_sint1_out.txt
```

Tenemos los siguientes elementos a destacar:

1. **Makefile.** Símplemente compila el proyecto, y con *make clean* elimina todo salvo lo que hay en la estructura que presentamos. Estas son las únicas instrucciones precisas para probar nuestro programa. El ejecutable final aparece en el directorio raíz, con el nombre *minipascal*.

2. **docker.** Directorio que contiene una imagen de docker. Símplemente usar *make* para instalarla, *make run* para iniciarla, y dentro tenemos la versión de los laboratorios de ubuntu, así cómo flex, bison y spim.

3. **src.** Contiene el código de nuestro proyecto.

4. **unit_testing.** Contiene todos los casos de prueba del aula virtual.

3. Análisis léxico.

Para empezar con el compilador en sí, vamos a centrar nuestra atención en el análisis léxico del lenguage, que es el primer paso en el proceso de compilación de un programa. Una vez queda definido el léxico que el lenguage va a utilizar, esto es, los caracteres, palabras y conjuntos de palabras admitidos en el lenguage, debemos implementar, de alguna forma, un analizador que, dado un programa en miniPascal, verifique que es léxicamente correcto. Para este propósito, usamos la herramienta Flex, que nos proporciona una forma sencilla de implementar un analizador como el explicado. Lo único que tenemos que hacer es, mediante expresiones regulares, indicarle a Flex qué es léxicamente aceptable, y esta herramienta se encarga de generar un programa que hace la tarea que queremos.

Pasamos a ver cómo implementamos estas normas léxicas.

3.1. Implementación.

A continuación vemos el código del programa flex que ejercerá la función de analizador léxico.

Código 2: Fichero snippets/minipascal.l

```
%{
    #include "minipascal.tab.h"
    #include <math.h>
    int comment_start = 0; // guardo donde comienza el comentario
                           // para mostrar el error
    int errores_lexicos = 0;
%}
digito              [0-9]
letra               [a-zA-Z]
entero              {digito}+
idinit              [a-zA-Z_]
panico              [^ \n\t\ra-zA-Z0-9_();:.,+/*-]

/* Opción para disponer de números de línea */
%option yylineno
/*%option debug*/

/* Condición de contexto */
%x comentario

%%
[ \n\t]+                                    ;
"//"(.*)[\n]                                ;
"(*"                                        {
                                                comment_start = yylineno;
                                                BEGIN(comentario);
                                            }
<comentario>"*)"                            BEGIN(INITIAL);
<comentario>(.|\n)                          ;
<comentario><<EOF>>                         {
                                                printf("Error en la linea %d, comentario sin cerrar\n",
    comment_start);
                                                errores_lexicos++;
                                                return 0;
                                            }
program                                     return PROGRAM;
function                                    return FUNCTION;
const                                       return CONST;
var                                         return VAR;
integer                                     return INTTYPE;
begin                                       return BEGINN;
end                                         return END;
if                                          return IF;
```

```
then                                    return THEN;
else                                    return ELSE;
while                                   return WHILE;
do                                      return DO;
for                                     return FOR;
to                                      return TO;
write                                   return WRITE;
read                                    return READ;
{idinit}({idinit}|{digito})*            {
                                            yylval.str = strdup(yytext);
                                            if(yyleng > 16){
                                                printf("Error en linea %d, identificador %s demasiado
    largo\n", yylineno, yytext);
                                                errores_lexicos++;
                                            }
                                            return ID;
                                        }
{entero}                                {
                                            yylval.str = strdup(yytext);
                                            int first_nzero = 0;
                                            while(yytext[first_nzero] == '0') first_nzero++;
                                            if(yyleng - first_nzero > 12 || llabs(atoll(yytext)) > pow
    (2,31)){
                                                printf("Error en linea %d, entero %s demasiado grande\n",
    yylineno, yytext);
                                                errores_lexicos++;
                                            }
                                            return INTCONST;
                                        }
"("                                     return LPAREN;
")"                                     return RPAREN;
";"                                     return SEMICOLON;
":"                                     return COLON;
","                                     return COMMA;
"."                                     return POINT;
":="                                    return ASSIGNOP;
"+"                                     return PLUSOP;
"-"                                     return MINUSOP;
"*"                                     return MULTOP;
"/"                                     return DIVOP;
\"(\\.|[^"\n])*\"                       {
                                            yylval.str = strdup(yytext);
                                            return STRING;
                                        }
\"(\\.|[^"\n])*                         {
                                            yylval.str = strdup(yytext);
                                            printf("Error en linea %d, cadena sin cerrar\n",yylineno);
                                            errores_lexicos++;
                                            return STRING;
                                        }
{panico}+                               {
                                            printf("Error en linea %d, secuencia de caracteres incorrectos
     %s\n", yylineno, yytext);
                                            errores_lexicos++;
                                        }
%%
```

Como podemos ver, este fragmento de código es bastante autoexplicativo. Se indican mediante expresiones regulares las composiciones léxicamente correctas. Todo aquello que no se considera válido, lo agrupamos en una ER que llamamos 'panico' y sirve para reconocer errores. Toda aquella estructura que sea reconocida por alguna ER distinta de 'panico' se considera correcta y devuelve su token asociado, que no es más que el emparejamiento entre una construcción léxica y una etiqueta que diferencia esta estructura de las demás. Estos tokens serán después usados por el analizador sintáctico.

Quizás merece la pena explicar el análisis de los comentarios multilínea, ya que requieren de la generación de un contexto distinto al que usamos para analizar el resto del programa. Cuando encontramos el inicio de un comentario, como podemos ver, guardamos la línea de comienzo del mismo en la variable 'comment_start' y ejecutamos la sentencia BEGIN(comentario). Esto hace que flex se fije solo en las reglas que comienzan por <comentario>, omitiendo todas las

demás.

También es importante la variable yylval.str, en la que guardamos los lexemas cuando estos son necesarios para el correcto análisis posterior y funcionamiento del futuro programa en miniPascal. Estos lexemas importantes son, como podemos ver, los identificadores, los números enteros y las cadenas.

3.2. Ejemplos de uso.

Código 3: Fichero

```
alumno@compiladores:~$ ./minipascal unit_testing/lexico/test_lex2.mp
Error en linea 7, entero 123456789012 demasiado grande
Error en linea 11, identificador _123456789012345678 demasiado largo
Error en la línea 11: _123456789012345678 no declarado
Error en linea 14, secuencia de caracteres incorrectos #@$%&
Error en linea 17, cadena sin cerrar
Error en la linea 20, comentario sin cerrar
Error en linea 22: syntax error, unexpected $end, expecting RPAREN or COMMA
Errores léxicos: 5
Errores sintácticos: 1
Errores semánticos: 1
```

Una nota a tener en cuenta es que hemos tocado un poco el fichero de prueba, pues la gramática no permitía una lista vacía de parámetros al write, y por o tanto tuvimos que añadir un elemento para que no dejase de analizar tras el tercer error léxico, pues este producía un error semántico.

4. Análisis sintáctico.

Pasamos ahora a la parte del análisis sintáctico. La funcionalidad del analizador sintáctico es, básicamente comprobar que las construcciones lexicamente correctas del programa se correspondan con estructuras que tienen sentido en el lenguaje que hemos definido.

Para llevar a cabo esta tarea, tenemos que definir la gramática del lenguaje, que indica precisamente qué estructuras son válidas y cuáles no.

Tras definir esta gramática, debemos implementar un programa que sea capaz de entender las reglas que hemos establecido y decidir si un programa miniPascal de entrada las cumple o no. Aquí damos uso de la herramienta Bison, que nos facilita mucho este trabajo, pues reduce nuestra tarea a indicar las reglas de derivación que consideramos válidas y a establecer las acciones oportundas para cada regla.

A continuación podemos ver cómo queda nuestro código de bison.

4.1. Implementación.

En el siguiente snippet de código vemos el código completo de bison usado para el analizador sintáctico.

Código 4: Fichero snippets/minipascal.y

```
%{
    #include <stdio.h>
    #include <string.h>
    #include "listaSimbolos.h"
    #include "listaCodigo.h"
    #include "semantic_analysis.h"
    #include "code_gen.h"
```

```
    extern int yylex();
    extern int yylineno;

    int errores_sintacticos = 0;
    int errores_semanticos = 0;
    extern int errores_lexicos;
    void yyerror(const char *msg);
    int ok();

    // Lista de símbolos
    Lista l;
    // puntero auxiliar
    PosicionLista aux, aux_fd;
    int in_function = 0;
    int param_count;
%}

%code requires {
    #include "listaCodigo.h"
}

/* Tipos de datos */
%union {
    char *str;
    ListaC codigo;
}

%type <codigo> expression statement read_list print_list print_item compound_statement optional_statements
    statements declarations constants function functions arguments expressions

/* Tokens de la gramática */

%token PROGRAM
%token FUNCTION
%token CONST
%token VAR
%token INTTYPE
%token BEGINN
%token END
%token IF
%token THEN
%token ELSE
%token WHILE
%token DO
%token FOR
%token TO
%token WRITE
%token READ
%token <str> ID
%token <str> INTCONST
%token <str> STRING
%token LPAREN
%token RPAREN
%token SEMICOLON
%token COLON
%token COMMA
%token POINT
%token ASSIGNOP
%token PLUSOP
%token MINUSOP
%token MULTOP
%token DIVOP

%start program

/* genera trazas de depuración */
%define parse.error verbose
%define parse.trace


/* Asociatividad y preferencia */
```

```
%expect 1
%left PLUSOP MINUSOP
%left MULTOP DIVOP


%%
program     : { l = creaLS(); } PROGRAM ID LPAREN RPAREN SEMICOLON functions declarations compound_statement
    POINT {
                if(ok()){
                    imprimirLS(l);
                    ListaC output = program_output($7, $8, $9);
                    imprimirCodigo(output);
                    liberaLC(output);
                }
                liberaLS(l);
           }
            ;
functions   : functions function SEMICOLON { if(ok()) $$ = functions_claus($1, $2);
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($2);
                                        }
                                    }
            | { if(ok()) $$ = functions_lambda(); else $$ = NULL;}
            ;
function    : FUNCTION ID {
                    parse_function_declaration($2);
                    in_function = 1;
                } LPAREN CONST {
                    args_on();
                } identifiers {
                    aux_fd = args_off();
                } COLON type RPAREN COLON type declarations compound_statement {
                    end_function_declaration();
                    if(ok())
                        $$ = function_f(aux_fd, $14, $15);
                    else{
                        $$ = NULL;
                        liberaLC($14);
                        liberaLC($15);
                    }
                    in_function = 0;
                }
            ;
declarations : declarations VAR identifiers COLON type SEMICOLON { if(ok()) $$ = decl_id($1);
                                                                    else{
                                                                        $$ = NULL;
                                                                        liberaLC($1);
                                                                    }
                                                                }
             | declarations CONST constants SEMICOLON { if(ok()) $$ = decl_const($1, $3);
                                                        else{
                                                            $$ = NULL;
                                                            liberaLC($1);
                                                            liberaLC($3);
                                                        }
                                                    }
             | { if(ok()) $$ = decl_lambda(); else $$ = NULL;}
             ;

identifiers : ID { insert_identifier($1, VARIABLE); /* si es argumento lo detecta en el módulo*/ }
            | identifiers COMMA ID { insert_identifier($3, VARIABLE); }
            ;
type        : INTTYPE
            ;
constants   : ID ASSIGNOP expression { aux = insert_identifier($1, CONSTANTE); if(ok()) $$ = const_assign(aux,
     $3);
                                                                                else{
                                                                                    $$ = NULL;
                                                                                    liberaLC($3);
```

```
                                                                                                    }
                                                                                                }
            | constants COMMA ID ASSIGNOP expression { aux = insert_identifier($3, CONSTANTE); if(ok()) $$ =
    const_claus($1, aux, $5);
                                                                                                    else{
                                                                                                        $$ = NULL;
                                                                                                        liberaLC(
    $1);
                                                                                                        liberaLC(
    $5);
                                                                                                    }
                                                                                            }
            ;
compound_statement : BEGINN optional_statements END { if(ok()) $$ = compstat_optstat($2);
                                                        else{
                                                            $$ = NULL;
                                                            liberaLC($2);
                                                        }
                                                    }
                   ;
optional_statements : statements {
                                    if(ok()) $$ = optstat_stats($1);
                                    else{
                                        $$ = NULL;
                                        liberaLC($1);
                                    }
                                }
                    | { if(ok()) $$ = optstat_lambda(); else $$ = NULL; }
                    ;
statements  : statement { if(ok()) $$ = stats_stat($1);
                          else{
                              $$ = NULL;
                              liberaLC($1);
                          }
                        }
            | statements SEMICOLON statement { if(ok()) $$ = stats_claus($1, $3);
                                               else{
                                                   $$ = NULL;
                                                   liberaLC($1);
                                                   liberaLC($3);
                                               }
                                           }
            ;
statement   : ID ASSIGNOP expression{
                                    aux = check_identifier($1, VARIABLE);
                                    if(aux != NULL && ok())
                                        $$ = stat_assign(aux, $3);
                                    else{
                                        $$ = NULL;
                                        liberaLC($3);
                                    }
                                }
            | IF expression THEN statement { if(ok()) $$ = stat_if($2, $4);
                                             else{
                                                 $$ = NULL;
                                                 liberaLC($2);
                                                 liberaLC($4);
                                             }
                                           }
            | IF expression THEN statement ELSE statement { if(ok()) $$ = stat_if_else($2, $4, $6);
                                                            else{
                                                                $$ = NULL;
                                                                liberaLC($2);
                                                                liberaLC($4);
                                                                liberaLC($6);
                                                            }
                                                        }
            | WHILE expression DO statement { if(ok()) $$ = stat_while($2, $4);
                                              else{
                                                  $$ = NULL;
                                                  liberaLC($2);
```

```
                                                liberaLC($4);
                                            }
                                        }
            | FOR ID ASSIGNOP expression TO expression DO statement { aux = check_identifier($2, VARIABLE); if
    (aux != NULL && ok()) $$ = stat_for(aux, $4, $6, $8);

    else{

      $$ = NULL;

      liberaLC($4);

      liberaLC($6);

      liberaLC($8);
                                                                                                        }
                                                                                                        }
            | WRITE LPAREN print_list RPAREN { if(ok()) $$ = stat_write($3);
                                            else{
                                                $$ = NULL;
                                                liberaLC($3);
                                            }
                                        }
            | READ LPAREN read_list RPAREN { if(ok()) $$ = stat_read($3);
                                        else{
                                            $$ = NULL;
                                            liberaLC($3);
                                        }
                                    }
            | compound_statement { if(ok()) $$ = stat_comp($1);
                                else{
                                    $$ = NULL;
                                    liberaLC($1);
                                }
                            }
            ;
print_list  : print_item { if(ok()) $$ = printl_printit($1);
                            else{
                                $$ = NULL;
                                liberaLC($1);
                            }
                        }
            | print_list COMMA print_item { if(ok()) $$ = printl_claus($1, $3);
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($3);
                                        }
                                    }
            ;
print_item  : expression { if(ok()) $$ = printit_exp($1);
                            else{
                                $$ = NULL;
                                liberaLC($1);
                            }
                        }
            | STRING {
                        int str_id = insert_string($1);
                        if(ok()) $$ = printit_str(str_id);
                        else $$ = NULL;
                    }
            ;
read_list   : ID { aux = check_identifier($1, VARIABLE); if(aux != NULL && ok()) $$ = readl_id(aux); else $$ =
     NULL; }
            | read_list COMMA ID { aux = check_identifier($3, VARIABLE); if(aux != NULL && ok()) $$ =
    readl_claus($1, aux);
                                                                        else{
                                                                            $$ = NULL;
                                                                            liberaLC($1);
                                                                        }
                                                                    }
```

```
            ;
expression  : expression PLUSOP expression { if(ok()) $$ = expr_op($1, $3, '+');
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($3);
                                        }
                                    }
            | expression MINUSOP expression { if(ok()) $$ = expr_op($1, $3, '-');
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($3);
                                        }
                                    }
            | expression MULTOP expression { if(ok()) $$ = expr_op($1, $3, '*');
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($3);
                                        }
                                    }
            | expression DIVOP expression { if(ok()) $$ = expr_op($1, $3, '/');
                                        else{
                                            $$ = NULL;
                                            liberaLC($1);
                                            liberaLC($3);
                                        }
                                    }
            | MINUSOP expression { if(ok()) $$ = expr_neg($2);
                                else{
                                    $$ = NULL;
                                    liberaLC($2);
                                }
                            }
            | LPAREN expression RPAREN { if(ok()) $$ = expr_paren($2);
                                        else{
                                            $$ = NULL;
                                            liberaLC($2);
                                        }
                                    }
            | ID { aux = check_identifier($1, VARIABLE | CONSTANTE | ARGUMENTO); if(aux != NULL && ok()) $$ =
    expr_id(aux); else $$ = NULL; }
            | INTCONST { if(ok()) $$ = expr_num($1);
                            else{
                                $$ = NULL;
                            }
                        }
            | ID { parse_function_call($1); } LPAREN arguments RPAREN { end_function_call();
                                                                        if(ok()) aux = buscaLS(l, $1)->sig, $$
     = expr_func(aux, $4);
                                                                        else{
                                                                            $$ = NULL;
                                                                            liberaLC($4);
                                                                        }
                                                                    }
            ;
arguments   : { param_count = 0; } expressions { if(ok()) $$ = args_exprs($2);
                                                else{
                                                    $$ = NULL;
                                                    liberaLC($2);
                                                }
                                            }
            | { if(ok()) $$ = args_lambda(); else $$ = NULL; }
            ;
expressions : expression { add_param();
                            if(ok()) exprs_expr($1, param_count++);
                            else {
                                $$ = NULL;
                                liberaLC($1);
                            }
```

```
                        }
            | expressions COMMA expression { add_param();
                                             if(ok()) exprs_claus($1, $3, param_count++);
                                             else{
                                                 $$ = NULL;
                                                 liberaLC($1);
                                                 liberaLC($3);
                                             }
                                           }
%%

void yyerror(const char *msg){
    printf("Error en linea %d: %s\n", yylineno, msg);
    errores_sintacticos++;
}

int ok() {
    return !(errores_lexicos + errores_sintacticos + errores_semanticos);
}
```

En este código también se realiza parte del análisis semántico, que explicaremos en el siguiente apartado. Vamos a intentar restringirnos en este punto puramente al análisis sintáctico.

Como puede observarse, las reglas de producción de la gramática quedan claramente plasmadas en este código, y ciertamente es una gramática sencilla de entender.

No hay mucho que explicar en este punto, realmente solo trata de plasmar la gramática en un programa bison y encontrar los errores encontrados en el análisis de un programa.

Para tratar los errores usamos la función yyerror, que simplemente escribe por pantalla que se ha encontrado un error, la línea en la que este ha sido detectado, y un mensaje de error pasado como parámetro. Además, aumenta un contador de errores sintácticos. Esta función es llamada por bison automáticamente cuando detecta una estructura no reconocida por las reglas de producción de la gramática.

4.2. Ejemplos de uso.

Código 5: Fichero

```
alumno@compiladores:~$ ./minipascal unit_testing/sintactico/test_sint2.mp
Error en linea 7: syntax error, unexpected IF, expecting END
Errores léxicos: 0
Errores sintácticos: 1
Errores semánticos: 0
```

5. Análisis semántico.

En esta parte de la práctica requerimos de una tabla de símbolos, que en realidad es una lista enlazada, que ha sido implementado por los profesores de la asignatura. Muy por encima, los errores tratados son redefiniciones de un mismo identificador, y el uso de un identificador que no ha sido definido. De hecho la implementación de esto no tiene casi ningún misterio, si no fuese por la ampliación de las funciones. Para la ampliación de las funciones tenemos los siguientes obstáculos:

1. Se puede utilizar el identificador de la función como una variable más dentro de la función.
2. Se pueden repetir identificadores siempre que el alcance no sea el mismo. Para esto usaremos la idea del profesor de añadir dentro de las funciones el prefijo **<NOMBRE_FUNCIÓN>.** a los identificadores locales de cada función.
3. Una función no debe de tener más de cuatro elementos.

4. Una función debe ser llamada con el número correcto de elementos.

Como vemos, se complica mucho más la cosa con la entrada en juego de las funciones. Veamos ahora algunos detalles de como solucionamos esto.

5.1. Implementación.

Cómo hemos visto en el fichero *minipascal.y*, realmente el análisis sintáctico nos lo hemos llevado al módulo *syntactic_analysis*. Procedo primero a listar el fichero de cabecera y a explicar qué hace cada cosa.

Código 6: Fichero snippets/syntactic_analysis.h

```
#ifndef __SEMANTIC_ANALYSIS__
#define __SEMANTIC_ANALYSIS__
#include "listaSimbolos.h"

extern Lista l;
extern int errores_semanticos;
extern int yylineno;

// for function related analysis
void parse_function_declaration(char* name);
void end_function_declaration();
void parse_function_call(char* name);
void end_function_call();
void add_param();
void args_on();
PosicionLista args_off();

// for general analysis, including functions
PosicionLista insert_identifier(char* name, int type);
int insert_string(char* str);
PosicionLista check_identifier(char* name, int types);

// outputs data
void imprimirLS();

#endif
```

La idea con el módulo era tener una especie de API que nos permitiese olvidarnos de esta parte en el archivo *minipascal.y*. Por los nombres de las funciones se intuye un poco lo que hacen. Por supuesto al final las funciones más importantes son las cuatro últimos, que insertan y checkean los identificadores, e imprimen la sección de datos, que realmente podría estar en la parte de generación de código, pero hemos decidido dejarlo aquí para tener todo lo relacionado con cada tipo de lista separado. Antes tenemos varias funciones y procedimientos que sirven para que sirven para decirle al módulo que estamos en el contexto de una función, y alguna utilidad más para el tema de las funciones.

Las partes principales del código son:

1. El siguiente código se encarga de poner al módulo en el contexto de que está analizando una función. Al comienzo de la regla de producción, se llama a *parse_function_declaration*, que se asegura de que no esté ya definida otra función con el mismo nombre. Siempre pone el booleano a 1 de que estamos en una función. Si todo va bien, crea una nueva lista de símbolos en la que se añadirá la función y los parámetros, y se establece una variable PosicionLista apuntando a la función actual. Posteriormente la llamada a *args_on* indica que comenzamos a contar los argumentos de la función. Tras la llamada a **args_off**, si no se han pasado de 4 parámetros, se concatena la lista nueva a la de símbolos. Si todo ha ido bien se devuelve la PosicionLista que apunta a la función, lo cual usa la generación de código.

Código 7: Fichero snippets/parse_func_declaration.c

```
// start function parsing mode
void parse_function_declaration(char* name) {
```

```
    parsing_func = 1;  // parsing function now
    PosicionLista p = buscaLS(l, name);
    if (p != finalLS(l)) {
        // function redeclared
        throw_semantic_error(name, REDECLARATION);
        return;
    }
    aux = creaLS();
    current_fd = insert_ls(name, FUNCION, 0, aux);
}

// end function parsing mode
void end_function_declaration() {
    parsing_func = 0;
    current_fd = NULL;
}

// start argument parsing
void args_on() {
    parsing_args = 1;
    count_args = 0;
}

// end argument parsing
PosicionLista args_off() {
    parsing_args = 0;
    if (current_fd != NULL) {
        // if more arguments than possible, do not add to list
        if (count_args > 4) {
            throw_semantic_error(current_fd->dato.nombre, TOOMANYARGS);
            liberaLS(aux);
            current_fd = NULL;
            aux = NULL;
            return NULL;
        }
        current_fd->dato.valor = count_args;
        concatenaLS(l, aux);
        aux = NULL;
        return current_fd;
    }
    return NULL;
}
```

2. En el siguiente snippet mostramos un código similar al anterior pero para las llamadas a funciones. Al finalizar el parseo se comprueba que el número de parámetros que se le han pasado es el correcto.

Código 8: Fichero snippets/parse_func_call.c

```
// This method searches for FUNCTION in ls
PosicionLista get_function(char* arg) {
    PosicionLista p = buscaLS(l, arg);
    if (p == finalLS(l)) {
        throw_semantic_error(arg, NOTDECLARED);
        return NULL;
    }
    if (p->sig->dato.tipo != FUNCION) {
        throw_semantic_error(arg, NOTAFUNCTION);
        return NULL;
    }
    return p->sig;
}

void parse_function_call(char* name) {
    PosicionLista p = get_function(name);
    if (p != NULL) current_fc = p;
    count_args = 0;
}

void add_param() { count_args++; }
```

```
void end_function_call() {
    if (current_fc != NULL)
        if (count_args != current_fc->dato.valor)
            throw_semantic_error(current_fc->dato.nombre, WRONGPARAMNUMBER);
    current_fc = NULL;
}
```

3. Mostramos ahora las funciones para insertar tanto identificadores como cadenas. El de las cadenas no tiene mucho contenido, quizás sea interesante que devuelve el contador de cadena para el generador de código. Veamos la inserción de identificadores. La mayor parte del código es la parte de análisis dentro de una función. Si estamos analizando una función, pero la variable que apunta a la función actual está a NULL, ha habido algún error al comienzo, y no se ha insertado la función, con lo que no queremos analizar más. Si va bien la cosa aún, se comprueba si es el nombre de la función, lo cual no permitimos, y si no le ponemos ya el prefijo de a función. Por último, el caso especial de que estemos mirando argumentos, en cuyo caso se comprueba que no se redeclare, y se inserta en la lista auxiliar de la función actual. Por último se inserta en todos los casos igual.

Código 9: Fichero snippets/insert.c

```
PosicionLista insert_identifier(char* name, int type) {
    char* original_name = name;
    PosicionLista ret;
    if (parsing_func) {
        if (current_fd == NULL) return NULL;
        if (parsing_args) count_args++;
        if (strcmp(current_fd->dato.nombre, name) == 0) {
            // nombre del identificador coincide con nombre de función
            throw_semantic_error(name, FUNCARGNAME);
            return NULL;
        }
        sprintf(buffer, "%s.%s", current_fd->dato.nombre, name);
        name = buffer;
        if (parsing_args) {
            PosicionLista p = buscaLS(l, name);
            if (p != finalLS(l) ||
                (ret = insert_ls(name, ARGUMENTO, count_args - 1, aux)) ==
                    NULL) {
                throw_semantic_error(original_name, REDECLARATION);
                return NULL;
            }
            return ret;
        }
    }
    if ((ret = insert(name, type, 0)) == NULL) {
        throw_semantic_error(original_name, REDECLARATION);
        return NULL;
    }
    return ret;
}

int insert_string(char* str) {
    PosicionLista p = buscaLS(l, str);
    if (p == finalLS(l)) {
        Simbolo aux;
        aux.nombre = strdup(str);
        aux.valor = str_counter++;
        aux.tipo = CADENA;
        insertaLS(l, finalLS(l), aux);
        return aux.valor;
    } else
        return recuperaLS(l, p).valor;
}
```

4. Sigue un esquema la función de chequeo de identificadores. Si estamos en una función pero no se ha insertado no analizamos. Si se usa el identificador de la función actual, se devuelve su posición en la lista. Si no se añade el prefijo y se checkea de manera standard.

Código 10: Fichero snippets/check.c

```
// This method searches for VARIABLE, CONSTANT or ARGUMENT in LS
PosicionLista check_identifier(char* name, int types) {
    char* original_name = name;
    // If in function, add prefix
    if (parsing_func) {
        if (current_fd == NULL) return NULL;
        if (strcmp(current_fd->dato.nombre, name) == 0) {
            // están accediendo a lo que guarda la función
            // no hay ningún error
            return current_fd;
        }
        sprintf(buffer, "%s.%s", current_fd->dato.nombre, name);
        name = buffer;
    }
    // Search for identifier
    PosicionLista p = buscaLS(l, name);
    if (p == finalLS(l)) {
        // If we didn't find it, semantic error, not declared
        throw_semantic_error(name, NOTDECLARED);
        return NULL;
    } else {
        Simbolo sim = recuperaLS(l, p);
        if ((types & sim.tipo) == 0) {
            throw_semantic_error(name, WRONGTYPE);
            return NULL;
        }
    }
    return p->sig;
}
```

5.2. Estructuras de datos.

La única estructura de datos usada es la lista de símbolos implementada por los profesores. Lo único que hemos añadido es la operación de concatenación sin copia, que tiene orden constante.

5.3. Ejemplos de uso.

Código 11: Fichero

```
alumno@compiladores:~$ for i in 3 4 5 6 7 8; do echo "unit_testing/funciones/test_f$i.mp" | xargs ./minipascal
     ; echo ; done
Error en la línea 3: y es el nombre de la función
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 1

Error en la línea 4: y es el nombre de la función
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 1

Error en la línea 6: y.a es constante
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 1

Error en la línea 4: a redeclarado
Error en la línea 14: b redeclarado
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 2

Error en la línea 3: y tiene más de 4 argumentos
```

```
Error en la línea 9: y no declarado
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 2

Error en la línea 9: y llamada con número incorrecto de parámetros
Error en la línea 10: b no declarado
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 2
```

Código 12: Fichero

```
alumno@compiladores:~$ for i in 1 2 3 4; do echo "unit_testing/semantico/test_sem$i.mp" | xargs ./minipascal;
    echo; done
##################
# Seccion de datos
  .data
$str1:
  .asciiz "Test 1\n"
$str2:
  .asciiz "Fin test1\n"
_a:
  .word 0
_b:
  .word 0
_c:
  .word 0

###################
# Seccion de codigo
  .text
  .globl main
main:
  li $t0,3
  sw $t0,_b
  li $t0,0
  sw $t0,_c
  li $v0,4
  la $a0,$str1
  syscall
  li $t0,1
  lw $t1,_b
  add $t2,$t0,$t1
  sw $t2,_a
  li $v0,5
  syscall
  sw $v0,_a
  li $v0,4
  la $a0,$str2
  syscall
  li $v0,10
  syscall

Error en la línea 6: a redeclarado
Error en la línea 7: a redeclarado
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 2

Error en la línea 7: x no declarado
Error en la línea 8: y no declarado
Error en la línea 9: z no declarado
Errores léxicos: 0
Errores sintácticos: 0
Errores semánticos: 3

Error en la línea 7: b es constante
Error en la línea 8: b es constante
Errores léxicos: 0
Errores sintácticos: 0
```

```
Errores semánticos: 2
```

6. Generación de código.

Llegamos ahora a la última fase del proceso de compilación: la generación de código MIPS ensamblador.

Una vez sabemos que el programa es totalmente correcto, tanto léxica, sintáctica y semánticamente, debemos traducir el código miniPascal a un programa en código ensamblador MIPS, para poder ser ejecutado. Nótese, no obstante, que esta generación de código se hace, en realidad, paralelamente al análisis sintáctico y semántico, aunque deja de realizarse en el momento en el que se detecta algún error.

El esquema de funcionamiento es el siguiente:

1. Se traduce siguiendo el análisis ascendente de la gramática de miniPascal. Por lo tanto, si lo vemos como un árbol de derivación, cuando estamos analizando un nodo, ya tenemos generado el código de los nodos inferiores.
2. El hecho de que tengamos las listas de código de los hijos implica que no hace falta que sepamos lo que hacen, sino dónde guardan el resultado. Esto se almacena en la lista de código.
3. Hay que llevar particular cuidado con la liberación de registros conforme ya no se van necesitando.

Vamos a ver más detalladamente cómo llevamos a cabo este proceso.

6.1. Implementación.

Para llevar a cabo este trabajo hemos programado un modulo 'codegen', que hace una función de API similar a la de 'semantyc_analysis.c', explicada en el apartado anterior.

En el siguiente snippet vemos las funciones que tiene este módulo.

Código 13: Fichero snippets/codegen.h

```
#ifndef __CODE_GEN__
#define __CODE_GEN__

#include "listaCodigo.h"
#include "listaSimbolos.h"
extern int in_function;

// expressions -> ...
ListaC exprs_expr(ListaC arg, int param_count);
ListaC exprs_claus(ListaC arg1, ListaC arg2, int param_count);

// expression -> ...

ListaC expr_num(char* arg);
ListaC expr_id(PosicionLista p);
ListaC expr_op(ListaC arg1, ListaC arg2, char op);
ListaC expr_paren(ListaC arg);
ListaC expr_neg(ListaC arg);
ListaC expr_func(PosicionLista p, ListaC arg);

// statement -> ...

ListaC stat_assign(PosicionLista arg1, ListaC arg2);
ListaC stat_write(ListaC arg);
ListaC stat_read(ListaC arg);
ListaC stat_if(ListaC arg1, ListaC arg2);
ListaC stat_if_else(ListaC arg1, ListaC arg2, ListaC arg3);
```

```
ListaC stat_while(ListaC arg1, ListaC arg2);
ListaC stat_comp(ListaC arg);
ListaC stat_for(PosicionLista arg1, ListaC arg2, ListaC arg3, ListaC arg4);

// statements -> ...

ListaC stats_stat(ListaC arg);
ListaC stats_claus(ListaC arg1, ListaC arg2);

// optional_statements -> ...

ListaC optstat_lambda();
ListaC optstat_stats(ListaC arg);

// compound_statement -> ...

ListaC compstat_optstat(ListaC arg);

// print_* -> ...

ListaC printit_exp(ListaC arg);
ListaC printit_str(int arg);
ListaC printl_printit(ListaC arg);
ListaC printl_claus(ListaC arg1, ListaC arg2);

// read_list -> ...

ListaC readl_id(PosicionLista arg);
ListaC readl_claus(ListaC arg1, PosicionLista arg2);

// constants -> ...

ListaC const_assign(PosicionLista arg1, ListaC arg2);
ListaC const_claus(ListaC arg1, PosicionLista arg2, ListaC arg3);

// declarations -> ...

ListaC decl_id(ListaC arg);
ListaC decl_const(ListaC arg1, ListaC arg2);
ListaC decl_lambda();

// arguments -> ...

ListaC args_lambda();
ListaC args_exprs(ListaC arg);

// program -> ...

ListaC program_output(ListaC funcs, ListaC decl, ListaC comp_stat);

// function -> ...

ListaC function_f(PosicionLista p, ListaC decl, ListaC comp_stat);
ListaC functions_claus(ListaC arg1, ListaC arg2);
ListaC functions_lambda();

void imprimirCodigo(ListaC codigo);

#endif
```

Como podemos ver, hay realmente una función por cada regla de producción que requiere generar código. Esto hace que repitamos a veces código y tengamos funciones muy triviales, pero también permite que se oculte perfectamente la implementación, y que no haya que escribir nada en de generación de código en *minipascal.y*. Por lo tanto, nos centraremos solo en funciones particularmente difíciles, y detalles que son importantes para el correcto funcionamiento del programa.

A continuación vemos uno de los casos más sencillos, que corresponde a un nodo hoja por así decirlo. Como vemos está un poco cambiado respecto a la versión del vídeo, lo cual se debe a la mejora de las funciones. Nos pasan la posición en la lista de símbolos asociada al ID, y si se trata de una función, es que estamos tratando el valor de retorno de una función,

con lo que devolvemos una lista vacía, con el registro de respuesta *$v0*. En caso de argumento devolvemos lo mismo con el *$ai* correspondiente. El caso general es igual que en el vídeo.

Código 14: Fichero snippets/expr_id.c

```
ListaC expr_id(PosicionLista arg) {
    ListaC ret = creaLC();
    switch (arg->dato.tipo) {
        case FUNCION:
            guardaResLC(ret, "$v0");
            break;
        case ARGUMENTO:
            guardaResLC(ret, get_argument(arg->dato.valor));
            break;
        default:
            set_oper(obtener_reg(), "lw", concatena("_", arg->dato.nombre),
                     NULL);
            insertaLC(ret, finalLC(ret), oper);
            guardaResLC(ret, oper.res);
    }
    return ret;
}
```

Veamos ahora algún otro detalle. En las operaciones aritméticas ahora siempre creamos un registro nuevo para almacenar el resultado, liberando los otros dos. Esto lo hacemos para no alterar el valor de los argumentos en caso de que este sea el primer operando.

Código 15: Fichero snippets/expr_op.c

```
ListaC expr_op(ListaC arg1, ListaC arg2, char op) {
    concatenaLC(arg1, arg2);
    char* operation;
    switch (op) {
        case '+':
            operation = "add";
            break;
        case '-':
            operation = "sub";
            break;
        case '*':
            operation = "mul";
            break;
        case '/':
            operation = "div";
            break;
    }
    set_oper(obtener_reg(), operation, recuperaResLC(arg1),
             recuperaResLC(arg2));
    insertaLC(arg1, finalLC(arg1), oper);
    liberaLC(arg2);
    guardaResLC(arg1, oper.res);
    liberarReg(oper.arg1);
    liberarReg(oper.arg2);
    return arg1;
}
```

Otro pequeño cambio con la introducción de las funciones es el siguiente:

Código 16: Fichero snippets/save_syscall_regs.c

```
ListaC save_syscall_regs(ListaC arg, int save) {
    if (save & REGV) {
        set_oper("$s1", "move", "$v0", NULL);
        insertaLC(arg, inicioLC(arg), oper);
        set_oper("$v0", "move", "$s1", NULL);
        insertaLC(arg, finalLC(arg), oper);
    }
    if (save & REGA) {
        set_oper("$s0", "move", "$a0", NULL);
```

```
        insertaLC(arg, inicioLC(arg), oper);
        set_oper("$a0", "move", "$s0", NULL);
        insertaLC(arg, finalLC(arg), oper);
    }
    return arg;
}

ListaC printit_exp(ListaC arg) {
    set_oper("$v0", "li", WRITESYSCALLVAR, NULL);
    insertaLC(arg, finalLC(arg), oper);
    set_oper("$a0", "move", recuperaResLC(arg), NULL);
    insertaLC(arg, finalLC(arg), oper);
    set_oper(NULL, "syscall", NULL, NULL);
    insertaLC(arg, finalLC(arg), oper);
    liberarReg(recuperaResLC(arg));
    int save = in_function ? (REGV | REGA) : 0;
    return save_syscall_regs(arg, save);
}

ListaC printit_str(int arg) {
    ListaC res = creaLC();
    set_oper("$v0", "li", WRITESYSCALLSTR, NULL);
    insertaLC(res, finalLC(res), oper);
    set_oper("$a0", "la", get_str_tag(arg), NULL);
    insertaLC(res, finalLC(res), oper);
    set_oper(NULL, "syscall", NULL, NULL);
    insertaLC(res, finalLC(res), oper);
    int save = in_function ? (REGV | REGA) : 0;
    return save_syscall_regs(res, save);
}

ListaC readl_id(PosicionLista arg) {
    ListaC res = creaLC();
    set_oper("$v0", "li", READSYSCALL, NULL);
    insertaLC(res, finalLC(res), oper);
    set_oper(NULL, "syscall", NULL, NULL);
    insertaLC(res, finalLC(res), oper);
    if (arg->dato.tipo == FUNCION) {
        return res;
    }
    set_oper("$v0", "sw", concatena("_", arg->dato.nombre), NULL);
    insertaLC(res, finalLC(res), oper);
    return (in_function ? save_syscall_regs(res, REGV) : res);
}
```

Tenemos unas constantes REGV y REGA que valen 1 y 2 respectivamente, dos bits disjuntos. De esta manera con un entero el puedo decir a la función *save_ syscall_ regs* que registros debe guardar en *$s0* y *$s1*. Esta función será llamada entonces para lecturas y escrituras en funciones. Será en *minipascal.y* donde tengamos una variable que indique si estamos en una función, que usará este módulo (importándola con extern).

Vamos a pasar ahora a la generación de código de las mejoras. El resto de funciones son bastante sencillas, símplemente es concatenar listas e insertar las operacines correctas, y son fácilmente legibles en el código fuente.

La primera mejora es el bucle for:

Código 17: Fichero snippets/for.c

```
ListaC stat_for(PosicionLista arg1, ListaC arg2, ListaC arg3, ListaC arg4) {
    ListaC ret, advance_iteration, aux; // ret contains response
    char *tag_start = new_tag(), *tag_end = new_tag();
    ret = stat_assign(arg1, arg2); // id := expr1
    // concatenate iteration advance to body of iteration
    advance_iteration =
        stat_assign(arg1, expr_op(expr_id(arg1), expr_num("1"), '+'));
    concatenaLC(arg4, advance_iteration);
    liberaLC(advance_iteration);
    // beginning tag
    set_oper(tag_start, "tag", NULL, NULL);
    insertaLC(ret, finalLC(ret), oper);
```

```
    // comparison and jump
    concatenaLC(ret, arg3); // value of expr2
    aux = expr_id(arg1); // get value of variable
    concatenaLC(ret, aux);
    set_oper(recuperaResLC(aux), "bgt", recuperaResLC(arg3), tag_end); // compare these values
    insertaLC(ret, finalLC(ret), oper);
    // liberate registers and lists
    liberarReg(recuperaResLC(aux));
    liberarReg(recuperaResLC(arg3));
    liberaLC(arg3);
    liberaLC(aux);
    // body of loop
    concatenaLC(ret, arg4);
    liberaLC(arg4);
    // jump to beginning
    set_oper(tag_start, "b", NULL, NULL);
    insertaLC(ret, finalLC(ret), oper);
    // end tag
    set_oper(tag_end, "tag", NULL, NULL);
    insertaLC(ret, finalLC(ret), oper);
    return ret;
}
```

Comenzamos con una asignación a la variable. A continuación le concateno al cuerpo del bucle el avance en la iteración, esto es, sumar 1 a la variable. El esquema quedaría:

1. Asignación a la variable.
2. Etiqueta de comienzo de bucle 'start'
3. Comparación y salto condicional con *bg* a la etiqueta final 'end'.
4. Cuerpo del bucle con el avance concatenado.
5. Salto incondicional a la etiqueta 'start'.
6. Etiqueta 'end'.

Para la generación del código de una función tenemos ya que meter bastante código.

Código 18: Fichero snippets/funciones.c

```
ListaC function_f(PosicionLista p, ListaC decl, ListaC comp_stat) {
    concatenaLC(decl, comp_stat);
    liberaLC(comp_stat);
    return envolve(decl, concatena("_", p->dato.nombre));
}

ListaC envolve(ListaC arg, char* name) {
    char reg_aux[NREG] = {0};
    PosicionListaC p = inicioLC(arg);
    int offset, i;
    while (p != finalLC(arg)) {
        Operacion o = recuperaLC(arg, p);
        mark(o.res, reg_aux);
        /*mark(o.arg1, reg_aux);*/
        /*mark(o.arg2, reg_aux);*/
        p = siguienteLC(arg, p);
    }
    // beginning
    set_oper("$v0", "move", "$zero", NULL);
    insertaLC(arg, inicioLC(arg), oper);
    set_oper("$ra", "sw", "0($sp)", NULL);
    insertaLC(arg, inicioLC(arg), oper);
    for (offset = 4, i = NREG - 1; i >= 0; i--)
        if (reg_aux[i] != 0) {
            set_oper(int_to_reg(i), "sw", get_sp_offset(offset), NULL);
```

```
            insertaLC(arg, inicioLC(arg), oper);
            offset += 4;
        }
    set_oper("$sp", "addiu", "$sp", my_itoa(-offset));
    insertaLC(arg, inicioLC(arg), oper);
    set_oper(name, "tag", NULL, NULL);
    insertaLC(arg, inicioLC(arg), oper);
    // end
    set_oper("$ra", "lw", "0($sp)", NULL);
    insertaLC(arg, finalLC(arg), oper);
    for (offset = 4, i = NREG - 1; i >= 0; i--)
        if (reg_aux[i] != 0) {
            set_oper(int_to_reg(i), "lw", get_sp_offset(offset), NULL);
            insertaLC(arg, finalLC(arg), oper);
            offset += 4;
            reg_aux[i] = 0;
        }
    set_oper("$sp", "addiu", "$sp", my_itoa(offset));
    insertaLC(arg, finalLC(arg), oper);
    set_oper("$ra", "jr", NULL, NULL);
    insertaLC(arg, finalLC(arg), oper);
    return arg;
}
```

El mayor contenido lo tiene la función *envolve*, que mira en la lista de código los registros en los que se escribe, y a continución crea el comienzo y el final de la función, apilando y desapilando los registros necesarios, así como el registro que guarda la dirección de retorno.

Por último veamos las llamadas a funciones:

Código 19: Fichero snippets/expr_func.c

```
ListaC exprs_expr(ListaC arg, int param_count) {
    set_oper(get_argument(param_count), "move", recuperaResLC(arg), NULL);
    liberarReg(recuperaResLC(arg));
    insertaLC(arg, finalLC(arg), oper);
    return arg;
}

ListaC expr_func(PosicionLista p, ListaC arg) {
    set_oper(concatena("_", p->dato.nombre), "jal", NULL, NULL);
    insertaLC(arg, finalLC(arg), oper);
    char* res = obtener_reg();
    set_oper(res, "move", "$v0", NULL);
    insertaLC(arg, finalLC(arg), oper);
    guardaResLC(arg, res);
    return arg;
}
```

Tampoco tiene una gran complicación. Simplemente en las derivaciones de expressions aprovechamos para cargar el resultado en los *$ai* correspondientes.

Estos son los detalles más importantes de la implementación de la generación de código. Cómo hemos dicho, las funciones en general son bastante legibles, y no merece la pena pasar por ellas una a una en esta memoria.

6.2. Ejemplos de uso.

Código 20: Fichero

```
alumno@compiladores:~$ ./minipascal unit_testing/funciones/test_f11.mp
##################
# Seccion de datos
  .data
$str1:
  .asciiz "b == "
```

```
$str2:
  .asciiz "\n"
_siguiente.a:
  .word 0
_siguiente.c:
  .word 0
_b:
  .word 0

##################
# Seccion de codigo
  .text
  .globl main
_siguiente:
  addiu $sp,$sp,-16
  sw $t0,12($sp)
  sw $t1,8($sp)
  sw $t2,4($sp)
  sw $ra,0($sp)
  move $v0,$zero
  beqz $a0,$l2
  li $t0,1
  sub $t1,$a0,$t0
  move $a0,$t1
  jal _siguiente
  move $t0,$v0
  li $t1,1
  add $t2,$t0,$t1
  move $v0,$t2
  b $l1
$l2:
  li $t0,1
  move $v0,$t0
$l1:
  lw $ra,0($sp)
  lw $t2,4($sp)
  lw $t1,8($sp)
  lw $t0,12($sp)
  addiu $sp,$sp,16
  jr $ra
main:
  li $t0,5
  move $a0,$t0
  jal _siguiente
  move $t0,$v0
  sw $t0,_b
$l3:
  lw $t0,_b
  beqz $t0,$l4
  li $v0,4
  la $a0,$str1
  syscall
  lw $t1,_b
  li $v0,1
  move $a0,$t1
  syscall
  li $v0,4
  la $a0,$str2
  syscall
  lw $t1,_b
  li $t2,1
  sub $t3,$t1,$t2
  sw $t3,_b
  b $l3
$l4:
  li $t0,1
  sw $t0,_b
$l5:
  li $t1,10
  lw $t0,_b
  bgt $t0,$t1,$l6
```

```
  li $v0,4
  la $a0,$str1
  syscall
  lw $t2,_b
  li $v0,1
  move $a0,$t2
  syscall
  li $v0,4
  la $a0,$str2
  syscall
  lw $t2,_b
  li $t0,1
  add $t3,$t2,$t0
  sw $t3,_b
  b $l5
$l6:
  li $v0,10
  syscall
```

7. Conclusiones acerca de las prácticas de la asignatura

Respecto al tiempo estimado de trabajo, calculamos que hemos trabajado alrededor de 20 horas en este proyecto, fuera de las horas de clase.

Este ha sido un proyecto muy completo, que, además, complementa perfectamente los conceptos aprendidos en las clases de teoría. Al fin y al cabo, es fácil entender qué es una gramática y qué procesos se siguen para su análisis, pero no es tan visible su importancia para nuestro campo de estudio. Este proyecto nos ha ayudado a ver el sentido práctico de todos conceptos teóricos que hemos trabajado. Así como sus limitaciones y problemas asociados.

Por otro lado, aunque nuestro compilador sea muy simple, consideramos que es suficiente para comprender todo lo que hay detrás de cada lenguaje de programación, cada compilador, cada intérprete, y cada programa.