Taller 1

Universidad de la Empresa

Licenciatura en Informática

DOCUMENTACION
ENTREGA FINAL

Tutor

Federico Doreste

Integrantes del equipo

Felipe Schneider –
Nicolás Berrogorry –
Washington Chocho –

Diseño de estructuras de datos dinámicas

Introducción: Requerimientos

En primer instancia se requiere una estructura de datos dinámicos en el lenguaje C++ capaz de representar en memoria un programa de CalcuSimple que pueda de ser ejecutado. En este documento argumentamos las elecciones de las estructuras de datos fundamentales.

Struct Programa

Elegimos crear un struct <<Programa>> que contenga todos los datos necesarios para almacenar y ejecutar un programa de CalcuSimple, abstrayendo la funcionalidad para simplificar el <<main>>.

```
typedef struct{
	String nombre;
	ABBVariables variables;
	ListaInstrucciones instrucciones;
} Programa;
```

Las variables del programa se almacenan en un árbol binario de búsqueda y las instrucciones en una lista, ya que así lo especifica la letra del taller.

Struct Variable

Los datos (info) de los nodos del ABB de variables los almacenamos en un tipo estructurado <<Variable>> capaz de contener el nombre de una variable y su valor actual. El valor inicial de cada variable tendrá que ser establecido a cero en la implementación como lo especifica la letra de la tarea.

```
typedef struct{
	String nombre; // Nombre de la variable, propiedad de orden.
	int valor; // Dato o info, no influye en el orden del árbol.
} Variable;
```

Struct Instruccion

Los datos de los nodos de la lista de instrucciones los almacenamos en un struct <<Instruccion>> que contiene qué tipo de instruccion se tiene que ejecutar, sobre qué variable se aplica, y para una instrucción de tipo asignación los datos de la misma:

```
typedef struct{
	TipoInstruccion tipoInstr; // Discr.: LEER, MOSTRAR o ASIGNAR
	union {
		String variable; // tipoInstruccion == LEER o MOSTRAR
		Asignacion asignacion; // tipoInstruccion == ASIGNAR
	};
} Instruccion;
```

La única instrucción que requiere un struct especializado para ser representada es la asignación. Si bien esto se podria haber resuelto en éste módulo, elegimos separarlo para facilitar la codificacion, evitando módulos con procedimientos/funciones muy extensos y complejos.

Caso: Instrucciones de Asignación

Una asignación se compone de un lado izquierdo (variable siendo asignada), y un lado derecho con el valor, la variable o el cálculo al cual asignarla. El tipo de datos <<Asignacion>> puede representar cualquiera de estos casos:

```
typedef struct{
	String variableAsg; // Nombre de la variable a asignar
	FormaParametro formaDer; // Discr.: ENTERO, VARIABLE o FUNCION
	union{
		int enteroDer; // Valor a asignar (formaDer = ENTERO)
		String variableDer; // Variable a asignar (formaDer = VARIABLE)
		LlamadaFuncion llamadaDer; // Funcion '' (formaDer = FUNCION)
	};
} Asignacion;
```

Arriba aparece el struct <<LlamadaFuncion>>; dada la naturaleza compleja de una llamada a función, decidimos utilizar una serie de tipos de datos y discriminantes especializados, facilitando la forma de visualizar el problema y encausando la implementación a favor de la modularización.

Caso: Asignaciones a funciones

En CalcuSimple las instrucciones de asginacion de una variable a una función tienen la siguiente sintaxis:

nombrevariable = FUNC param1 param2

En esta instrucción, param1 y param2 pueden ser o valores enteros o nombres de variables, lo que abre un abanico de complejidad que resolvemos discriminando entre las cuatro posibles dupas de parametros (2 parámetros * las 2 formas posibles en las que aparecen):

```
typedef struct{
      FuncionPredefinida funcion; // Discr.: SUM, RES, MUL, DIV
      TipoDupla tipoDupla; // Discr.: ENTERO_ENTERO, ENTERO_VARIABLE,
                           //           VARIABLE_ENTERO, VARIABLE_VARIABLE
      union{
            DuplaEnteroEntero enteroEntero; // tipoDupla =  ENTERO_ENTERO
            DuplaEnteroVariable enteroVariable; // '' =  ENTERO_VARIABLE
            DuplaVariableEntero variableEntero; // '' =  VARIABLE_ENTERO
            DuplaVariableVariable variableVariable; // '' = VARIABLE_VARIABLE
      };
} LlamadaFuncion;
```

Por definición del lenguaje CalcuSimple, sólo existen 4 funciones predefinidas (SUM, RES, MUL, DIV), y todas reciben una dupla de parámetros; por esta razón se

hizo un enumerado con los mismos y no es necesario incluir esta información en las duplas.

Diseño de algoritmos (Seudocódigo)

Compilación

El ejecutable de CalcuSimple es responsable de la compilación de un programa de este lenguaje a archivos binarios que sean capaces de ser ejecutados posteriormente por el mismo,

La compilación se compone de dos etapas fundamentales:

- Parseo: Se analiza el documento <<*.csim >> y se construye un programa de CalcuSimple en memoria.

- Bajada a disco: Se descargan las estructuras del programa al disco duro en dos archivos, uno que contiene el árbol de variables y otro que contiene la lista de instrucciones, quedando así ambos archivos eficientemente almacenados en forma binaria para volver a cargarlos para su ejecución.

Decidimos que el algoritmo de Parseo es el único que requiere ser diseñado a través de seudocódigo, y en una forma muy detallada. No produciremos seudocódigo de la bajada al disco porque es un tema trivial dado el contenido del curso de Programación 2 y su obligatorio final.

Módulo <<Programa>>

```
Procedimiento ParsearPrograma(archivoFuente, nombre, cError, numLineaError, prog):
    Establecer codigoError a SIN_ERROR;
    Declarar linea de tipo string;
    Crear string vacío con linea;
    Declarar listapal del tipo listapalabras;
    Establecer linea a LeerSiguienteLinea(archivofuente, linea);
    Establecer palabras a SepararPalabras(LeerSiguienteLinea(archivoFuente));

    Declarar nombre de tipo String;
    Según DarCantidadPalabras(palabras) Hacer {
        Caso 0 y Caso 1: {
            Establecer codigoError a FALTAN_PALABRAS_EN_LINEA_PROGRAMA;
        }
        Caso 2: {
            Establecer pal a DarPrimerPalabra(palabras);
            Si pal es "PROGRAMA" Entonces {
                Establecer pal a DarSiguientePalabra(pal);
                Si EsAlfabeticaString(pal) es TRUE Entonces {
                     Si pal es igual a nombre Entonces {
                           Establecer nombrePrograma a pal;}
                } De otro modo {
                    Establecer codigoError a NOMBRE_PROGRAMA_INVALIDO;
                }
            } De otro modo {
                Establecer codigoError a NO_ES_RESERVADA_PROGRAMA;
            }
        }
        De otro modo: {
            Establecer codigoError a DEMASIADAS_PALABRAS_EN_LINEA_PROGRAMA;
        }
    }
    Declarar variables de tipo ABBVariables;
    Si codigoError es SIN_ERROR Entonces {
        Establecer codigoError, variables a
                                  ParsearABBVariables(SepararPalabras(
                                          LeerSiguienteLinea(archivoFuente)));
    }

    Declarar instrucciones de tipo ListaInstrucciones;
    Si codigoError es SIN_ERROR Entonces {
        Establecer codigoError, instrucciones a ParsearListaInstrucciones(
                                                    archivoFuente, variables );
    }

    Devolver codigoError, programa;
```

Módulo <<ABBVariables>>

```
Procedimiento ParsearABBVariables(palabras) de módulo ABBVariables:
    Establecer codigoError a SIN_ERROR;
    Establecer variables a nuevo ABBVariables;
    Si ContarPalabras(palabras) > 0 Entonces {
        Establecer pal a DarPrimerPalabra(palabras);
        Si pal es "VARIABLES" Entonces {
            Establecer pal a DarSiguientePalabra(pal);
            Mientras pal no sea NULL y codigoError sea SIN_ERROR Hacer {
                Si EsAlfabeticaString(pal) es TRUE Entonces {
                    Si ExisteVariable(pal, variables) es FALSE Entonces {
                        Insertar nueva Variable en variables con nombre=pal,
                                                                valor=0;
                        Establecer pal a DarSiguientePalabra(pal);
                    } De otro modo {
                        Establecer codigoError a VARIABLE_YA_EXISTE;
                    }
                } De otro modo {
                    Establecer codigoError a NOMBRE_VARIABLE_INVALIDO;
                }
            }
        } De otro modo {
            codigoError = NO_ES_RESERVADA_VARIABLES;
        }
    } De otro modo {
        codigoError = FALTAN_PALABRAS_EN_LINEA_VARIABLES;
    }

    Devolver codigoError, variables;
```

Módulo <<ListaInstrucciones>>

```
Procedimiento ParsearListaInstrucciones(archivoFuente, variables):
    Establecer codigoError a SIN_ERROR;
    Establecer instrucciones a nueva ListaInstrucciones;

    Establecer palabras a SepararPalabras(LeerSiguienteLinea(archivoFuente));

    Según DarCantidadPalabras(palabras) Hacer {
        Caso 0: {
            Establecer codigoError a FALTAN_PALABRAS_EN_LINEA_INSTRUCCIONES;
        }
        Caso 1: {
            Establecer pal a DarPrimerPalabra(palabras);

            Si pal es "INSTRUCCIONES" {
                Establecer linea a LeerSiguienteLinea(archivoFuente);
                Mientras linea no sea NULL y codigoError sea SIN_ERROR Hacer {
                    Establecer codigoError, inst a ParsearInstruccion(
                                                    SepararPalabras(linea),
                                                    variables
                                                );
                    Si codigoError es SIN_ERROR Entonces {
                        Insertar inst en instrucciones;
                        Establecer linea a LeerSiguienteLinea(archivoFuente);
                    }
                }
            } De otro modo {
                Establecer codigoError a NO_ES_RESERVADA_INSTRUCCIONES;
            }
        }
        De otro modo: {
            Establecer codigoError a DEMASIADAS_PALABRAS_EN_LINEA_INSTRUCCIONES;
        }
    }

    Devolver codigoError, instrucciones;
```

Módulo <<Instruccion>>

```
Procedimiento ParsearInstruccion(ListaPalabras listaPal, ABBVariables abb,
                        CodigoError &cError, Instruccion &instr)
    Establecer codigoError a SIN_ERROR;
    Declarar instruccion de tipo Instruccion;

    Si DarCantidadPalabras(palabras) >= 2 {
        Establecer codigoError, tipoInstr a ParsearTipoInstruccion(palabras);
        Si codigoError es SIN_ERROR Entonces {
            Según tipoInstr Hacer {
                Caso MOSTRAR y Caso LEER {
                     Establecer variable a
                                 DarStringPalabra(DarSiguientePalabra(
                                                  DarPrimerPalabra(listaPal)));
                     si EsAlfabeticaString(variable){
                        ExisteVariableABBVariables(variable, abb){
                              Establecer instr.tipoInstr a  tipoInstr
                              Estableccer instr.var a variable
                        }sino
                        Establecer codigoError a NO_EXISTE_VARIABLE_INSTRUCCION
                     }sino
                        Establecer codigoError a NOMBRE_VARIABLE_INVALIDO
                    Establecer variable a DarSiguientePalabra(
                                                 DarPrimerPalabra(palabras));
                    Establecer instruccion a nueva Instruccion con
                                                            tipoInstr, variable;
                }
                Caso ASIGNAR: {
                    Establecer codigoError, asign a ParsearAsignacion(palabras);
                    Si codigoError es SIN_ERROR Entonces {
                        Establecer instruccion a nueva Instruccion con
                                                                tipoInstr, asign;
                    }
                }
            }
        }
    } De otro modo {
        Establecer codigoError a FALTAN_PALABRAS_EN_INSTRUCCION;
    }

    Devolver codigoError, instruccion;
```

Módulo <<TipoInstruccion>>

```
Procedimiento ParsearTipoInstruccion(palabras):
    Establecer codigoError a SIN_ERROR;
    Declarar tipoInstruccion de tipo TipoInstruccion;

    Según DarCantidadPalabras(palabras) Hacer {
        Caso 2: {
            Según DarPrimerPalabra(palabras) Hacer {
                Caso "MOSTRAR": {
                    Establecer tipoInstruccion a MOSTRAR;
                }
                Caso "LEER": {
                    Establecer tipoInstruccion a LEER;
                }
                De otro modo: {
                    Establecer codigoError a
                                    NO_ES_LEER_NI_MOSTRAR_TIPO_INSTRUCCION;
                }
            }
        }
        Caso 3 y Caso 5: {
            Establecer pal a DarPrimerPalabra(palabras);
            Si EsValidoNombreVariable(pal) es TRUE Entonces {
                Establecer pal a DarSiguientePalabra(pal);
                Si pal es "=" Entonces {
                    Establecer tipoInstruccion a ASIGNAR;
                } De otro modo {
                    Establecer codigoError a
                                    SE_ESPERABA_SIGNO_IGUAL_TIPO_INSTRUCCION;
                }
            } De otro modo {
                Establecer codigoError a SE_ESPERABA_VARIABLE_TIPO_INSTRUCCION;
            }
        }
        De otro modo: {
            Establecer codigoError CANTIDAD_PALABRAS_INCORRECTA_TIPO_INSTRUCCION;
        }
    }

    Devolver codigoError, tipoInstruccion;
```

Módulo <<Asignacion>>

```
Procedimiento ParsearAsignacion(palabras, variables):
    Precondiciones:
        • Hay o bien 3 o 5 palabras en <<palabras>>.
        • La primer palabra es un nombre de variable válido y la segunda es "=".

    Establecer codigoError a SIN_ERROR;
    Declarar asignacion de tipo Asignacion;

    Establecer variableIzq a DarPrimerPalabra(palabras);
    Si ExisteVariable(variableIzq, variables) Entonces {
        Establecer palabraDer a DarSiguientePalabra(
                                    DarSiguientePalabra(variableIzq));
        Establecer codigoError, formaDer a ParsearFormaParametro(palabraDer);
        Si codigoError es SIN_ERROR Entonces {
            Según formaDer Hacer {
                Caso ENTERO: {
                    Establecer asignacion a nueva Asignacion con variableIzq,
                                                                  formaDer,
                                                                  ParsearEntero(
                                                                      palabraDer);
                }
                Caso VARIABLE: {
                    Si ExisteVariable(palabraDer) es TRUE {
                        Establecer asignacion a nueva Asignacion con variableIzq,
                                                                      formaDer,
                                                                      palabraDer;
                    } De otro modo {
                        Establecer codigoError a
                                        NO_EXISTE_VARIABLE_DER_ASIGNACION;
                    }
                }
                Caso FUNCION: {
                    Establecer tresPalabrasDer de tipo ListaPalabras a palabraDer;
                    Establecer codigoError, llamadaFunc a ParsearLlamadaFuncion(
                                                                   tresPalabrasDer,
                                                                   variables);
                    Si codigoError es SIN_ERROR Entonces {
                        Establecer asignacion a nueva Asignacion con variableIzq,
                                                                      formaDer,
                                                                      llamadaFunc;
                    }
                }
            }
        }
    } De otro modo {
        Establecer codigoError a NO_EXISTE_VARIABLE_IZQ_ASIGNACION;
    }
      Devolver codigoError, asignacion;
```

Módulo <<FormaParametro>>

```
Procedimiento ParsearFormaParametro(palabra):
      Establecer codigoError a SIN_ERROR;
      Declarar formaParametro de tipo FormaParametro;

      Si EsStringNumerica(palabra) es TRUE Entonces {
            Establecer forma a ENTERO;

      } En cambio Si EsValidoNombreVariable(string) es TRUE Entonces {
            Establecer forma a VARIABLE;

      } De otro modo {
        Si EsFuncionPredefinida(palabra) es TRUE {
                Establecer forma a FUNCION;
        } De otro modo {
            Establecer codigoError a NO_ES_FUNCION_PREDEFINIDA_FORMA_PARAMETRO;
        }
      }

      Devolver codigoError, forma;
```

Módulo <<FuncionPredefinida>>

```
Funcion EsFuncionPredefinida(palabra):
      Establecer esPredef a FALSE;

      Si palabra está en (“SUM”, “RES”, “MUL”, “DIV”) Entonces {
            Establecer esPredef a TRUE;
      }

      Devolver esPredef;

Procedimiento ParsearFuncionPredefinida( str, FuncionPredefinida )
{
   establecer FuncionPredefinida a SUM;

    segun (str[0]) {
        caso 'R': igualar FuncionPredefinida a RES;
        caso 'M': igualar FuncionPredefinida a MUL;
        caso 'D': igualar FuncionPredefinida a DIV;
    }
}
```

Módulo <<LlamadaFuncion>>

```
Procedimiento ParsearLlamadaFuncion(palabras, variables):
    Precondición:
       • <<palabras>> contiene tres palabras.

    Establecer codigoError a SIN_ERROR;
    Declarar llamadaFuncion de tipo LlamadaFuncion;

    Establecer funcion a DarPrimerPalabra(palabras);
    Establecer argumentos de tipo ListaPalabras a DarSiguientePalabra(funcion);

    Establecer codigoError, tipoDup a ParsearTipoDupla(argumentos);
    Si codigoError es SIN_ERROR {
        Establecer funcionPredef a ParsearFuncionPredefinida(funcion);
        Según tipoDupla Hacer {
            Caso ENTERO_ENTERO: {
                Establecer duplaEnteroEnt a ParsearDuplaEnteroEntero(argumentos);
                Si DarDerechoDuplaEnteroEntero(duplaEnteroEnt) > 0 Entonces {
                    Establecer llamadaFuncion a nueva LlamadaFuncion con
                                                            funcionPredef,
                                                            tipoDup,
                                                            duplaEnteroEnt;
                } De otro modo {
                    Establecer codigoError a
                                    DIVISION_ENTRE_CERO_DUPLA_ENTERO_ENTERO;
                }
            }
            Caso ENTERO_VARIABLE: {
                Establecer codigoError, duplaEnteroVar a
                                        ParsearDuplaEnteroVariable(argumentos,
                                                                   variables);
                Si codigoError es SIN_ERROR Entonces {
                    Establecer llamadaFuncion a nueva LlamadaFuncion con
                                                            funcionPredef,
                                                            tipoDup,
                                                            duplaEnteroVar;
                }
            }
            Caso VARIABLE_ENTERO: {
                Establecer codigoError, duplaVariableEnt a
                                        ParsearDuplaVariableEntero(argumentos,
                                                                   variables);
                Si codigoError es SIN_ERROR Entonces {
                    Si DarDerechoDuplaVariableEntero(duplaVariableEnt) > 0
                    Entonces {
                        Establecer llamadaFuncion a nueva LlamadaFuncion con
                                                            funcionPredef,
                                                            tipoDup,
                                                            duplaVariableEnt;
```

```
                } De otro modo {
                    Establecer codigoError a
                                DIVISION_ENTRE_CERO_DUPLA_VARIABLE_ENTERO;
                }
            }
        }
        Caso VARIABLE_VARIABLE: {
            Establecer codigoError, duplaVariableVar a
                                ParsearDuplaVariableVariable(argumentos,
                                                             variables);
            Si codigoError es SIN_ERROR Entonces {
                Establecer llamadaFuncion a nueva LlamadaFuncion con
                                                        funcionPredef,
                                                        tipoDup,
                                                        duplaVariableVar;
            }
        }
    }
}

Devolver codigoError, llamadaFuncion;
```

Módulo <<TipoDupla>>

```
Procedimiento ParsearTipoDupla(argumentos):
    Precondición:
       • <<argumentos>> es una ListaPalabras con dos palabras.

    Establecer codigoError a SIN_ERROR;
    Declarar tipoDupla de tipo TipoDupla;
    Declarar arg1esNum de tipo Boolean;
    Declarar arg2esNum de tipo Boolean;

    Si EsStringNumerica(DarPrimerPalabra(argumentos)) es TRUE Entonces {
        Establecer arg1esNum a TRUE;
    } En cambio si EsValidoNombreVariable(argumento1) es TRUE Entonces {
        Establecer arg1esNum a FALSE;
    } De otro modo {
        Establecer codigoError a ARG1_NO_ES_NUM_NI_VAR_TIPO_DUPLA;
    }

    Si codigoError es SIN_ERROR Entonces {
        Si EsStringNumerica(DarPrimerPalabra(argumentos)) es TRUE {
            Establecer arg1esNum a TRUE;
        } En cambio si EsValidoNombreVariable(argumento1) es TRUE{
            Establecer arg1esNum a FALSE;
        } De otro modo {
            Establecer codigoError a ARG2_NO_ES_NUM_NI_VAR_TIPO_DUPLA;
        }
    }

    Si codigoError es SIN_ERROR Entonces {
        Según (arg1esNum, arg2esNum) Hacer {
            Caso (TRUE, TRUE): {
                Establecer tipoDupla a ENTERO_ENTERO;
            }
            Caso (TRUE, FALSE): {
                Establecer tipoDupla a ENTERO_VARIABLE;
            }
            Caso (FALSE, TRUE): {
                Establecer tipoDupla a VARIABLE_ENTERO;
            }
            Caso (FALSE, FALSE): {
                Establecer tipoDupla a ENTERO_ENTERO;
            }
        }
    }

    Devolver codigoError, tipoDupla;
```

Módulo <<DuplaEnteroEntero>>

```
Procedimiento ParsearDuplaEnteroEntero( argumentos, dupla)
{
   declarar pal del tipo  Palabra* e igualarla a  DarPrimerPalabra(argumentos);
    igualar dupla.izq a ParsearEntero(DarStringPalabra(pal));
    igualar dupla.der a ParsearEntero(DarStringPalabra(DarSiguientePalabra(pal)));
}
```

Módulo <<DuplaEnteroVariable>>

```
Procedimiento ParsearDuplaEnteroVariable( argumentos,  variables, codigoError,
dupla)
{
    establecer codigoError a SIN_ERROR;
    declarar pal del tipo Palabra* e igualarla a  DarPrimerPalabra(argumentos);

    si (la funcion
ExisteVariableABBVariables(DarStringPalabra(DarSiguientePalabra(pal)), variables)
devuelve verdadero)
    {
        establecer dupla.ent a ParsearEntero(DarStringPalabra(pal));
        llamar a CrearStringVacio(dupla.var);
        llamar a CopiarString(dupla.var,
DarStringPalabra(DarSiguientePalabra(pal)));
    }
    sino
       establecer codigoError a  NO_EXISTE_VARIABLE_DUPLA_VARIABLE_ENTERO;
}
```

Módulo <<DuplaVariableEntero>>

```
Procedimiento ParsearDuplaVariableEntero( argumentos, codigoError, dupla, abbVar)
{
   establecer codigoError a SIN_ERROR;
    declarar pal del tipo Palabra* e igualarla a DarPrimerPalabra(argumentos);
    si (la funcion ExisteVariableABBVariables(DarStringPalabra(pal),abbVar)
devuelve verdadero) {
       llamar a  CrearStringVacio(dupla.parametroA);
       llamar a  CopiarString(dupla.parametroA,DarStringPalabra(pal));
       establecer dupla.parametroB a
ParsearEntero(DarStringPalabra(DarSiguientePalabra(pal)));
    } sino {
        establecer codigoError a  NO_EXISTE_VARIABLE_DUPLA_VARIABLE_ENTERO;
    }
```

Módulo <<DuplaVariableVariable>>

```
procedimiento ParsearDuplaVariableVariable( argumentos, codigoError,
duplaVariableVariable, abbVar)
{
    establecer codigoError a SIN_ERROR;
    declarar pal del tipo Palabra* e igualarla a DarPrimerPalabra(argumentos);
    si (la funcion ExisteVariableABBVariables(DarStringPalabra(pal),
abbVar)devuelve verdadero)
    {
        llamar a CrearStringVacio(duplaVariableVariable.parametroA);
        llamar a CopiarString(duplaVariableVariable.parametroA,
DarStringPalabra(pal));
        establecer pal a DarSiguientePalabra(argumentos);
        si (la funcion ExisteVariableABBVariables(DarStringPalabra(pal), abbVar)
devuelve verdadero)
        {
            llamar a CrearStringVacio(duplaVariableVariable.parametroB);
            llamar a CopiarString(duplaVariableVariable.parametroB,
DarStringPalabra(pal));
        }
        sino
            establecr codigoError a NO_EXISTE_VAR_DER_DUPLA_VARIABLE_VARIABLE;
    }
    sino
        establecer codigoError a NO_EXISTE_VAR_IZQ_DUPLA_VARIABLE_VARIABLE;
}
```

Ejecución

El ejecutable de CalcuSimple también es capaz de ejecutar los programas que compila. De un modo similar a la compilación, la ejecución se divide en dos partes:

- Carga de las estructuras de datos del programa desde el disco. Tampoco incluimos seudocódio para estos algoritmos por las mismas razones.

- Ejecución del programa, para el cual presentamos los algoritmos siguientes.

Módulo <<Programa>>

```
Procedimiento AbrirArchivosPrograma(direccion,  nombre,  fopenMode, archivoVars,
&archivoInst)
{
    Declarar String nomArchivoVars, nomArchivoInst;
    Modificar nomArchivoVars con CrearStringVacio();
    Modificar nombArchivoInst con CrearStringVacio();
    Si direccion es diferente a “”
      hacer
    {
        Copiar direccion a nomArchivoVars;
        Concatenar el string “/” a nomArchivoVars;
    }
    Concatenar el String nombre a nomArchivoVars;
    Copiar en el String nomArchivoInst el string nomArchivoVars;
    Concatenar el String nomArchivoVars y ".vars";
    Concatenar el String nomArchivoInst y ".inst";
    establecer archivoVars a fopen(nomArchivoVars, fopenMode);
    establecer archivoInst a fopen(nomArchivoInst, fopenMode);
    Liberar el String nomArchivoVars;
    Liberar el String nomArchivoInst;

Procedimiento CargarPrograma( dire,  nombre, cError, prog)
{
    establecer cError a SIN_ERROR;
    Crear String Vacio prog.nombre;
    Copiar el String nombre en prog.nombre;
    declarar archivoVars como puntero a FILE;
    declarar archivoInst como puntero a FILE;
    Abrir Archivos Programa(dire, prog.nombre, "rb", archivoVars, archivoInst);
    si (archivoVars no es nulo)
    {
        Crear el ABBVariables prog.variables;
        Cargar el ABBVariables prog.variables desde archivoVars;
        si (archivoInst no es nulo){
            Cargar la ListaInstrucciones prog.instrucciones desde archivoInst;
            establecer archivoInst a fclose(archivoInst);
        }
        sino
        {
           establecer cError a NO_EXISTEN_ARCHIVOS_COMPILADOS;
        }
        establecer archivoVars a fclose(archivoVars);
    }
    sino
    {
       establecer cError a NO_EXISTEN_ARCHIVOS_COMPILADOS;
    }
}
```

Módulo <<ABBVariables>>

```
Procedimiento InsertarNuevaABBVariable( nombre, cError, abb)
{
    Establecer  cError  SIN_ERROR;
    Si (abb es nulo) {
         Establecer abb a nuevo NodoABBVariables;
        Crear Variable abb->var con nombre;
        establecer abb -> izq a NULL;
        establecer abb -> der a NULL;
    } sino {
        Si (Es Igual el String nombre y abb->var){
            establecer cError a VARIABLE_YA_EXISTE_ABB_VARIABLES;
        }Sino
            si (Es Menor String nombre que abb->var){
            Llamar a InsertarNuevaABBVariable(nombre, cError, abb->izq);
        } sino {
           llamar a InsertarNuevaABBVariable(nombre, cError, abb->der);
        }
    }
}

Funcion DarVariableABBVariables( nombre, ABBVariables )
{
    si(abb es nulo ) {
        devolver nulo;
    }sino{
        si(Es Igual String nombre que abb->var){
            devolver el puntero de abb->var;
        } sino {
            si(Es Menor el String nombre que abb->var){
                devolver el puntero abb->izq;
            } sino {
                devolver el puntero abb->der;
            }
        }
    }
}
```

Módulo <<ListaInstrucciones>>

```
Procedimiento InsertarComienzoListaInstrucciones(lista, instruccion)
{
    declarar nueva del tipo ListaInstrucciones y etablecerla como  new
NodoListaInstrucciones;
    establecer nueva->instruccion a inst;
    si(lista es  NULL){
        establecer nueva->sig a NULL;
    } sino {
       establecer nueva->sig a lista;
    }
    establecer lista a nueva;
}

Procedimiento EjecutrarListaInstrucciones(abb, cError, numLineaError,
                                  listaInstr)
{
   establecer cError a SIN_ERROR;
    establecer  numLineaError a 4;
    mientras (listaInstr sea NULL y cError sea SIN_ERROR)
    {
        llamar a EjecutarInstruccion(listaInstr->instruccion, abb, cError);

        si(cError es SIN_ERROR){
            establecer listaInstr a listaInstr->sig;
            aumentar numLineaError un uno;
        }
    }
}


procedimiento GuardarListaInstrucciones( instrucciones, archivo)
{
    si(instrucciones no es NULL)
    {
        llamar a GuardarInstruccion(instrucciones->instruccion, archivo);
        llamar a GuardarListaInstrucciones(instrucciones->sig, archivo);
    }
}


Procedimiento CargarListaInstrucciones(instrucciones,  archivo)
{
    establecer instrucciones a NULL;
    declarar instrLeida del tipo Instruccion ;
    llamar a CargarInstruccion(instrLeida, archivo);
    mientras(no sea el final de archivo)
    {
        llamar a InsertarComienzoListaInstrucciones(instrucciones, instrLeida);
        llamar a CargarInstruccion(instrLeida, archivo);
```

```
    }
}


Procedimiento LiberarListaInstrucciones(listaInst)
{
    si(listaInst no es NULL)
    {
        llamar a LiberarListaInstrucciones(listaInst->sig);
        llamar a LiberarInstruccion(listaInst->instruccion);
    }
}
```

Módulo <<Instruccion>>

```
Funcion DarTipoInstruccion(instruccion)
{
    devolver instr.tipoInstr;
}


Procedimiento GuardarInstruccion(instruccion, archivo)
{
   llamar a GuardarTipoInstruccion(instr.tipoInstr, archivo);
    segun(instr.tipoInstr)
    {
        caso LEER:
        caso MOSTRAR:
            llamar a GuardarString(instr.var, archivo);

        caso ASIGNAR:
            llamar a GuardarAsignacion(instr.asignacion, archivo);

    }
}


Procedimiento CargarInstruccion(Instruccion, archivo)
{
    Llamar a CargarTipoInstruccion(instr.tipoInstr, archivo);
    Segun(instr.tipoInstr)
    {
        caso LEER:
        caso MOSTRAR:
            Llamar a CrearStringVacio(instr.var);
            llamar a CargarString(instr.var, archivo);

        caso ASIGNAR:
            llamar a CargarAsignacion(instr.asignacion, archivo);

    }
}
```

```
Procedimiento EjecutarInstruccion(Instruccion , ABBVariables , CodigoError)
{
    establecer cError a SIN_ERROR;
    segun (instr.tipoInstr)
    {
    caso LEER:
    {
        Declarar inputAux del tipo String ;
        llamar a CrearStringVacio(inputAux);
        imprimir ("  Ingrese \"");
        Declarar var del tipo Variable* e igualarla a
      DarVariableABBVariables(instr.var, abb);
        llamar a MostrarString(var->nombre);
        imprimir("\":\n  > ");
        Llamar a LeerString(inputAux);
        Mientras(inputAux no es un string numerico)){
            imprimir("* Ingrese un numero entero!\n");
            imprimir("  > ");
            llamar a LeerString(inputAux);
        }
        declarar intAux del tipo entero e igualarlo a ParsearEntero(inputAux);
        llamar a CambiarValorVariable(intAux, var);
        llamar a LiberarString(inputAux);

    }
    caso MOSTRAR:
        {
            imprimir ("  Se muestra \"");
            declarar var del tipo Variable* e igualarla a
DarVariableABBVariables(instr.var, abb);
            llamar a MostrarString(var->nombre);
            imprimir("\": %i\n", var->valor);
        };
    caso ASIGNAR:
        {
            declarar varAsg del tipo Variable* e igualarlo a
DarVariableABBVariables(
                                DarNombreVariableAsignacion(instr.asignacion),
abb);
            segun (DarFormaParametroAsignacion(instr.asignacion))
            {
            caso ENTERO:
                llamar a
CambiarValorVariable(DarEnteroDerAsignacion(instr.asignacion), varAsg);
                break;
            caso VARIABLE:
                {
                    declarar varDer del tipo Variable* e igualarlo a
DarVariableABBVariables(
```

```
DarVariableDerAsignacion(instr.asignacion), abb);
                    CambiarValorVariable(DarValorVariable(*varDer), varAsg);
                }
            caso FUNCION:
                {
                   declarar resu del tipo entero;
                    llamar a CalcularResultadoFuncion(
                        DarLlamadaFuncionDerAsignacion(instr.asignacion), abb,
resu, cError);
                    si(cError es SIN_ERROR)
                    {
                        llamar a CambiarValorVariable(resu, varAsg);
                    }
                }
            }
        }
    }
}


Procedimiento LiberarInstruccion(Instruccion &instr)
{
    Si(instr.tipoInstr es igual a  LEER o instr.tipoInstr es igual a  MOSTRAR) {
        llamar a LiberarString(instr.var);
    } sino {
       llamar a LiberarAsignacion(instr.asignacion);
    }
}
```

Módulo <<LlamadaFuncion>>

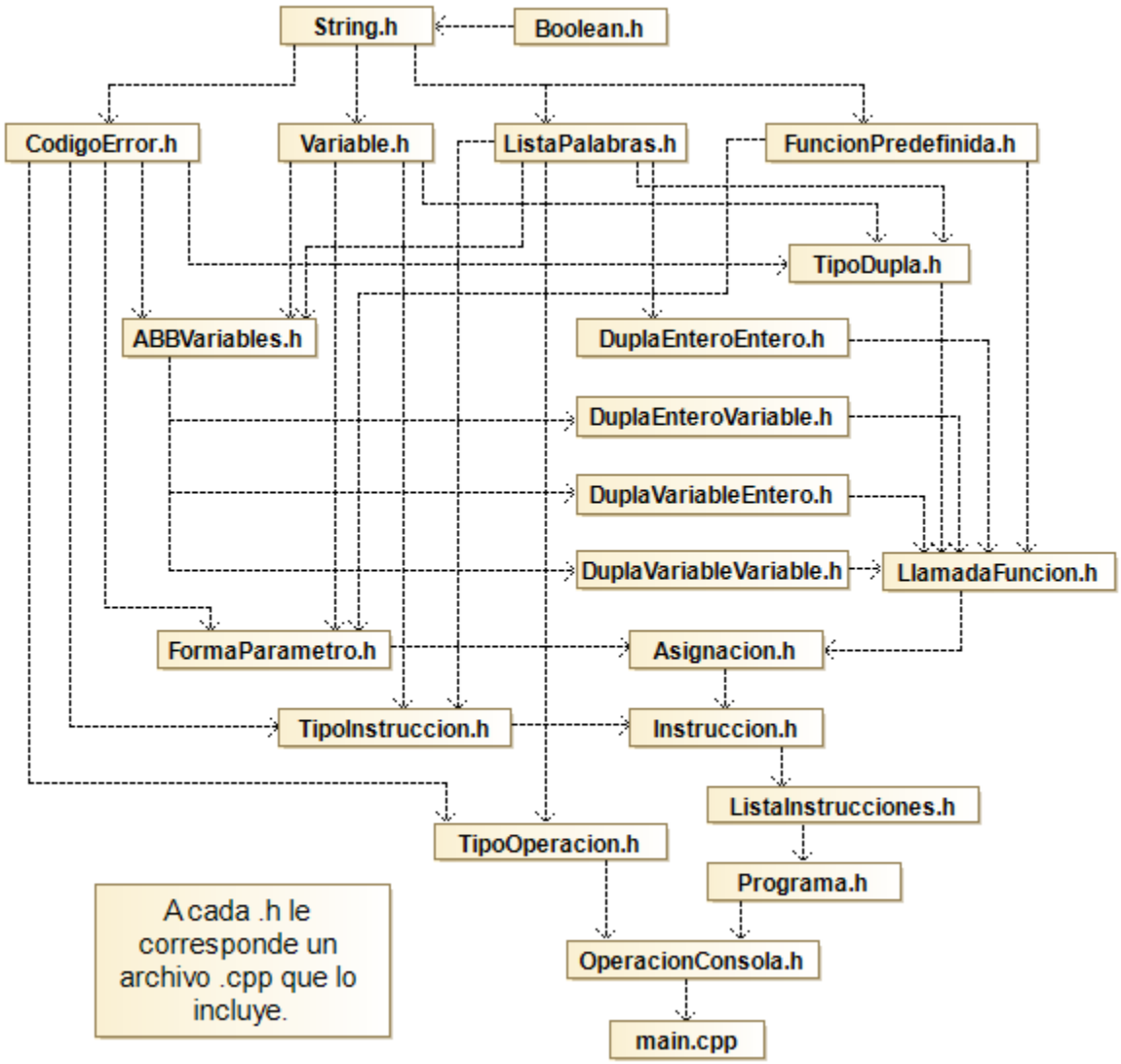
```
Procedimiento CalcularResultadoFuncion( llamadaF,  abbvars,  resu,  &cError)
{
    establecer cError a SIN_ERROR;
    declarar valIzq, valDer del tipo entero
    segun(llamadaF.tipoDupla)
    {
        caso ENTERO_ENTERO:
           igualar valIzq a DarIzqDuplaEnteroEntero(llamadaF.enteroEntero);
           igualar valDer a DarDerDuplaEnteroEntero(llamadaF.enteroEntero);

        caso ENTERO_VARIABLE:
            {
                igualar a valIzq a
DarEnteroDuplaEnteroVariable(llamadaF.enteroVariable);
                declarar varDer del tipo Variable* e igualarla a
DarVariableABBVariables(
                    DarVariableDuplaEnteroVariable(llamadaF.enteroVariable),
abbvars);
```

```
                igualar valDer a DarValorVariable(*varDer);
            }
        caso VARIABLE_ENTERO:
            {
                declarar varIzq del tipo Variable* e igualarla a
DarVariableABBVariables(
                    DarVariableDuplaVariableEntero(llamadaF.variableEntero),
abbvars);
                igualar valIzq a DarValorVariable(*varIzq);
                igualar valDer a
DarEnteroDuplaVariableEntero(llamadaF.variableEntero);
            }
        caso VARIABLE_VARIABLE:
            {
                declarar varIzq del tipo Variable* e igualarla a
DarVariableABBVariables(
                    DarIzqDuplaVariableVariable(llamadaF.variableVariable),
abbvars);
                igualar valIzq = DarValorVariable(*varIzq);
                declarar varDer del tipo Variable* igualarla a
DarVariableABBVariables(
                    DarDerDuplaVariableVariable(llamadaF.variableVariable),
abbvars);
                igualar valDer a DarValorVariable(*varDer);
            }
    }

    segun(llamadaF.funcion)
    {
        caso SUM: igualar resu a valDer + valIzq; break;
        caso RES: igualar resu a valIzq - valDer; break;
        caso MUL: igualar resu a valDer * valIzq; break;
        caso DIV:
            si(valDer es igual a  0)
                establecer cError a DIVISION_ENTRE_CERO;
            sino
                igualar resu a valIzq / valDer;

    }
}


procedimiento GuardarLlamadaFuncion( llamadaF,  archivo)
{
    llamar a GuardarFuncionPredefinida(llamadaF.funcion, archivo);
    llamar a GuardarTipoDupla(llamadaF.tipoDupla, archivo);
    segun (llamadaF.tipoDupla)
    {
    caso ENTERO_ENTERO:
        llamar a GuardarDuplaEnteroEntero(llamadaF.enteroEntero, archivo);

    caso ENTERO_VARIABLE:
```

```
        llamar a GuardarDuplaEnteroVariable(llamadaF.enteroVariable, archivo);

    caso VARIABLE_ENTERO:
        llamar a GuardarDuplaVariableEntero(llamadaF.variableEntero, archivo);

    caso VARIABLE_VARIABLE:
        llamar a GuardarDuplaVariableVariable(llamadaF.variableVariable, archivo);

    }
}


caso CargarLlamadaFuncion(LlamadaFuncion &llamadaF, FILE * archivo)
{
    llamar a CargarFuncionPredefinida(llamadaF.funcion, archivo);
    llamar a CargarTipoDupla(llamadaF.tipoDupla, archivo);
    segun (llamadaF.tipoDupla)
    {
        caso ENTERO_ENTERO:
            llamar a CargarDuplaEnteroEntero(llamadaF.enteroEntero, archivo);

        caso ENTERO_VARIABLE:
            llamar a CargarDuplaEnteroVariable(llamadaF.enteroVariable, archivo);

        caso VARIABLE_ENTERO:
            llamar a CargarDuplaVariableEntero(llamadaF.variableEntero, archivo);

        caso VARIABLE_VARIABLE:
            CargarDuplaVariableVariable(llamadaF.variableVariable, archivo);

    }
}


procedimiento LiberarLlamadaFuncion(llamadaF)
{
    segun(llamadaF.tipoDupla){
        caso ENTERO_ENTERO:
        caso ENTERO_VARIABLE: llamar a
LiberarDuplaEnteroVariable(llamadaF.enteroVariable);
        caso VARIABLE_ENTERO: llamar a
LiberarDuplaVariableEntero(llamadaF.variableEntero);
        caso VARIABLE_VARIABLE: llamar a
LiberarDuplaVariableVariable(llamadaF.variableVariable);
    }
}
```

Definición de encabezados a partir de las necesidades de los algoritmos

Diagrama de inclusiones



String.h

```
#ifndef STRING_H
#define STRING_H

#include "Boolean.h"

const int STRMAX = 200;
typedef char* String;

/*
Detalle: En varios módulos será necesario escribir cadenas constantes al programa, en forma de literales de string de c++. Estas literales con forma "abcd1234" son guardadas en un arreglo constante de char de forma interna por el compilador (comportamiento estandarizado), y por esto el programa estaría mal formado si asignaramos una literal a una cadena de tipo String. Entonces, es necesario un tipo de cadena constante que será utilizado en todos los parámetros de las operaciones que no sean pasados por referencia, es decir, que se pasen con el fin de solo lectura.
*/
typedef const char* ConstString;


/*
Precondición: El String NO fue creado ni cargado previamente.
Detalle: Crea un String con el fin de cadena como único caracter.
*/
void CrearStringVacio(String &str);


/*
    Precondición: EsStringNumerica(<<str>>) devuelve TRUE.
    Detalle: A partir de un ConstString parsea y devuelve por copia un entero.
*/

int ParsearEntero(ConstString str);
```

```
/* Precondición: El String fue creado.
Detalle: Devuelve el largo del String.
*/
int LargoString(ConstString str);


/*
Precondición: <<destino>> fue creado.
Detalle: Copia el <<origen>> en <<destino>>. Se pierde el valor previo de
<<destino>>.
*/
void CopiarString (String &destino, ConstString origen);


/*
    Precondiciones:
        * <<archivo>> es recibido en modo read.
        * <<archivo>> es de texto y utiliza CRLF como caracteres de fin de
línea.
    Detalle: Lee la siguiente linea y la carga en el string recibido por
referencia
*/
void LeerSiguienteLinea(FILE* archivo, String &str);



/*
Precondición: <<str>> fue creado.
Detalle: Carga <<str>> por referencia desde el teclado.
*/
void LeerString(String &str);


/*
Precondiciones: <<s1>> fue creado.
Detalle: Concatena el contenido de <<s2>> por referencia al final de
<<s1>>.
*/
void ConcatenarString (String &s1, ConstString s2);
```

```
/*
Precondición: Intercambia los contenidos de <<s1>> y <<s2>> por referencia.
Detalle: <<s1>> y <<s2>> fueron creados.
*/
void IntercambiarString (String &s1, String &s2);


/*
Detalle: Muestra un String por pantalla.
*/
void MostrarString (ConstString str);


/* Funcion que determina si s1 es alfabéticamente menor que s2 */
Boolean EsMenorString (ConstString s1, ConstString s2);


/*
Precondición: Ambas String fueron cargadas.
Detalle: Determina si <<s1>> y <<s2>> son iguales.
*/
Boolean EsIgualString(ConstString str1, ConstString str2);


/*
    Detalle: Devuelve TRUE si todos los caracteres del string se encuentran
en {'0', '1', '2', '3', '4', '5',
    '6', '7', '8', '9'} y FALSE en caso contrario.
*/
Boolean EsNumericaString(ConstString str);

/*
    Detalle: Devuelve TRUE si todos los caracteres del string son letras
minúsculas o mayúsculas.
*/
Boolean EsAlfabeticaString(ConstString str);
```

```
/*
Precondiciones:
* El String fue creado o cargado.
* El archivo está abierto en modo "wb".
Detalle: Guarda el String al final del archivo.
*/
void GuardarString(ConstString str, FILE* archivo);


/*
Detalle: Libera la memoria de <<str>> dejándola inusable.
*/
void LiberarString(String &str); // TODO: CUAL DE LOS DOS???



/*
Procedimiento modificación de datos de entrada.
Carga un String desde el archivo.
Precondiciones:
* El String NO fue creado ni cargado previamente.
* El archivo está abierto en modo "rb".
* El archivo contiene un String.
*/
void CargarString(String &str, FILE* archivo);


/*
    Precondición: <<destino>> fue creado.
    Detalle: Carga <<destino>> desde origen, quitando el fullpath y dejando
solo el nombre del archivo.
*/
void DarNombreArchivoSinBarras(String &destino, ConstString origen);


#endif
```

CodigoError.h

```
#ifndef CODIGO_ERROR_H
#define CODIGO_ERROR_H

#include "String.h"

typedef enum{
/* Sin error se define a cero para habilitar la comparación booleana
a FALSE en caso de que no haya un error */
SIN_ERROR = 0,

/* Los siguientes items son los errores extraídos del algoritmo de
parseo del proceso de compilación.
*/
NO_EXISTE_ARCHIVO, //Se esperaba archivo nombre.csim pero no se encontró
NO_EXISTEN_ARCHIVOS_COMPILADOS, //Es esperaba archivo .vars o .inst y no
se encontró
FALTAN_PALABRAS_EN_LINEA_PROGRAMA,
NOMBRE_PROGRAMA_INVALIDO,
NO_ES_RESERVADA_PROGRAMA,
NO_ES_IGUAL_NOMBRE_ARCHIVO_NOMBRE_PROGRAMA,
DEMASIADAS_PALABRAS_EN_LINEA_PROGRAMA,
VARIABLE_YA_EXISTE_ABB_VARIABLES,
NOMBRE_VARIABLE_INVALIDO,
NO_ES_RESERVADA_VARIABLES,
FALTAN_PALABRAS_EN_LINEA_VARIABLES,
FALTA_RESERVADA_LINEA_INSTRUCCIONES,
NO_ES_RESERVADA_INSTRUCCIONES,
DEMASIADAS_PALABRAS_EN_LINEA_INSTRUCCIONES,
FALTAN_PALABRAS_EN_INSTRUCCION,
NO_EXISTE_VARIABLE_INSTRUCCION,
NO_ES_LEER_NI_MOSTRAR_TIPO_INSTRUCCION,
SE_ESPERABA_SIGNO_IGUAL_TIPO_INSTRUCCION,
CANTIDAD_PALABRAS_INCORRECTA_TIPO_INSTRUCCION,
NO_EXISTE_VARIABLE_DER_ASIGNACION,
NO_EXISTE_VARIABLE_IZQ_ASIGNACION,
NO_ES_FORMA_PARAMETRO_VALIDA,
```

```
DIVISION_ENTRE_CERO_DUPLA_ENTERO_ENTERO,
DIVISION_ENTRE_CERO_DUPLA_VARIABLE_ENTERO,
ARG1_NO_ES_NUM_NI_VAR_TIPO_DUPLA,
ARG2_NO_ES_NUM_NI_VAR_TIPO_DUPLA,
NO_EXISTE_VARIABLE_DUPLA_ENTERO_VARIABLE,
NO_EXISTE_VARIABLE_DUPLA_VARIABLE_ENTERO,
NO_EXISTE_VAR_DER_DUPLA_VARIABLE_VARIABLE,
NO_EXISTE_VAR_IZQ_DUPLA_VARIABLE_VARIABLE,

  /* Los siguientes son errores de tipo operacion de la CLI */
  FALTA_NOMBRE_PROGRAMA_TIPO_OPERACION, // Falta la palabra del nombre
de programa
  OPERACION_DESCONOCIDA_TIPO_OPERACION, // No se reconoción el tipo de
operación
  CANTIDAD_PALABRAS_INCORRECTA_TIPO_OPERACION, // Cero palabras o más
de 3 palabras en el comando de consola


/* Los siguientes items son los errores extraídos del algoritmo de
ejecución.
*/
VARIABLE_NO_DEFINIDA,
DIVISION_ENTRE_CERO
} CodigoError;


/*
Detalle: Devuelve el string asignado a un código de error.
*/
ConstString DarStringCodigoError(CodigoError codigoError);



#endif
```

FuncionPredefinida.h

```
#ifndef FUNCION_PREDEFINIDA_H
#define FUNCION_PREDEFINIDA_H


#include "String.h" // Provee Boolean


typedef enum {
SUM, // Suma
RES, // Resta
MUL, // Multiplicación
DIV // División
} FuncionPredefinida;


/*
Detalle: Devuelve TRUE si <<str>> pertenece a {"SUM", "RES", "MUL", "DIV"}
y FALSE en caso contrario.
*/
Boolean EsFuncionPredefinida(ConstString str);


/*
Precondición: <<str>> pertenece a {"SUM", "RES", "MUL", "DIV"}.
Detalle: Devuevle por referencia una FuncionPredefinida dada una string de
función predefinida.
*/
void ParsearFuncionPredefinida(ConstString str, FuncionPredefinida &funcion);

/*
    Precondicion: <<archivo>> fue abierto en modo lectura.
    Detalle: Guarda la función predefinida en el archivo.
*/
void GuardarFuncionPredefinida(FuncionPredefinida func, FILE * archivo);
```

```
/*
    Precondicion: <<archivo>> fue abierto en modo lectura.
    Detalle: Carga la función predefinida desde el archivo.
*/
void CargarFuncionPredefinida(FuncionPredefinida &func, FILE * archivo);

#endif
```

ListaPalabras.h

```
#ifndef LISTA_PALABRAS_H
#define LISTA_PALABRAS_H


#include "String.h"


typedef struct nodoPalabra{
  nodoPalabra *sig;
  String palabra;
} Palabra;

typedef Palabra* ListaPalabras;


/*
Precondición: <<palabras>> no está creada.
Detalle: Genera una lista de palabras a partir de un string con grupos de caracteres separados por
espacios en blanco. Los espacios en blanco son descartados y cada grupo se guarda en una Palabra.
*/
void SepararPalabras(ConstString linea, ListaPalabras &lista);


/*
    Precondición: <<listaPal>> fue creada.
    Detalle: Devuelve la cantidad de palabras en la lista.
*/
int ContarPalabras(ListaPalabras listaPal);

/*
    Precondición: <<listaPal>> tiene al menos una palabra.
    Detalle: Devuelve la primer palabra de la lista.
*/
Palabra* DarPrimerPalabra(ListaPalabras listaPal);
```

```
/*
    Precondición: <<palabra>> tiene al menos una Palabra siguiente en su
lista.
    Detalle: Devuelve la siguiente Palabra.
*/
Palabra* DarSiguientePalabra(Palabra* palabra);


/*
    Precondición: <<palabra>> fue creada.
    Detalle: Devuelve en modo solo lectura la cadena de <<palabra>>.
*/
ConstString DarStringPalabra(Palabra* palabra);

/*
Precondición: <<lista>> fue cargada.
Detalle: Elimina todas las palabras de la lista.
*/
void LiberarListaPalabras(ListaPalabras &palabras);



#endif
```

Variable.h

```
#ifndef VARIABLE_H
#define VARIABLE_H


#include "String.h" // Provee Boolaean


typedef struct{
String nombre;
int valor;
} Variable;


/*
Precondición: <<variable>> NO fue creada.
Detalle: Crea <<variable>> con nombre <<nombre>> y valor cero, y la
devuelve por referencia.
*/
void CrearVariable(ConstString nombre, Variable &variable);


/*
Precondición: <<variable>> fue creada.
Detalle: Devuelve el nombre <<variable>> en modo solo lectura. No realiza
una copia.
*/
ConstString DarNombreVariable(Variable variable);


/*
Precondición: <<variable>> fue creada.
Detalle: Devuelve el valor actual de <<variable>>.
*/
int DarValorVariable(Variable variable);
```

```
/*
Precondición: <<variable>> fue creada.
Detalle: Cambia el valor de <<variable>>.
*/
void CambiarValorVariable(int nuevoValor, Variable &variable);


/*
    Precondicion: <<archivo>> esta abierto en modo "wb".
    Detalle: Guarda la variable en el disco.
*/
void GuardarVariable(FILE* archivo, Variable var);


/*
    Precondicion: <<archivo>> esta abierto en modo "rb".
    Detalle: Carga una variable desde el disco.
*/
void CargarVariable(FILE* archivo, Variable &var);


/*
Precondición: <<variable>> fue creada.
Detalle: Libera la memoria dinámica asociada a <<variable>>. Queda en un
estado inutilizable.
*/
void LiberarVariable(Variable &variable);


#endif
```

ABBVariables.h

```
#ifndef ABB_VARIABLES_H
#define ABB_VARIABLES_H

#include "CodigoError.h" // Provee ConstString y Boolean
#include "ListaPalabras.h"
#include "Variable.h"


typedef struct nodoABB{ // Nombre interno para autoreferencia
nodoABB* izq;
nodoABB* der;
  Variable var;
} NodoABBVariables; // Nodo


typedef NodoABBVariables* ABBVariables; // Raiz


/*
    Precondiciones:
        <<palabras>> fue creada.
        <<abb> NO fue creado.
    Detalle: Verifica que la primer palabra de <<palabras>> sea "VARIABLES"
y las siguientes sean nombres
             válidos de variables, y crea <<abb>> a partir de éstas. Todas
las variables son inicializa-
             das a cero. Devuelve el código de error correspondiente.
*/
void ParsearABBVariables(ListaPalabras palabras, CodigoError &codigoError,
ABBVariables &variables);



/*
    Precondición: <<abb>> NO fue creado.
    Detalle: Crea un ABBVariables vacío.
```

```
*/
void CrearABBVariables(ABBVariables &abb);

/*
    Precondiciones:
        * <<nombre>> es alfabética.
        * <<abb>> fue creado.
    Detalle: Inserta de forma ordenada una nueva variable en el árbol.
Devuelve por referencia el código de error correspondiente.
*/
void InsertarNuevaABBVariable(ConstString nombre, CodigoError &cError, ABBVariables &abb);


/*
    Precondiciones:
        * <<abb>> fue creado.
        * Existe una variable de nombre <<nombre>> en <<abb>>.
    Detalle: Devuelve la variable desde <<abb>> que tenga el nombre
<<nombre>>.
*/
Variable* DarVariableABBVariables(ConstString nombre, ABBVariables abb);

/*
 Precondicion: <<abb>> fue creado.
    Detalle: Devuelve TRUE si la variable existe en el árbol.
*/
Boolean ExisteVariableABBVariables(ConstString nombre, ABBVariables variables);


/*
    Precondiciones:
        * <<archivo>> está abierto en modo "wb".
        * <<abb>> fue creado.
    Detalle: Guarda <<abb>> en el <<archivo>>.
*/
void GuardarABBVariables(ABBVariables abb, FILE *archivo);

/*
Precondiciones:
* <<variables>> NO fue creado.
* <<archivo>> está abierto en modo "rb".
```

```
Detalle: Carga <<variables>> desde <<archivo>>.
*/
void CargarABBVariables(FILE* archivo, ABBVariables &variables);


/*
Precondición: <<variables>> fue creado.
Detalle: Libera la memoria dinamica asociada a <<variables>>.
*/
void LiberarABBVariables(ABBVariables &variables);



#endif
```

TipoDupla.h

```
#ifndef TIPO_DUPLA_H
#define TIPO_DUPLA_H


#include "ListaPalabras.h"
#include "CodigoError.h"
#include "Variable.h"


typedef enum{
ENTERO_ENTERO,
ENTERO_VARIABLE,
VARIABLE_ENTERO,
VARIABLE_VARIABLE
} TipoDupla;


/*
Precondición: <<argumentos>> tiene dos palabras.
Detalle: Identifica que tipo de dupla se requiere para almacenar los
argumentos y lo devuelve por
referencia junto con un CodigoError.
*/
void ParsearTipoDupla(ListaPalabras argumentos, CodigoError &codigoError,
TipoDupla &tipoDupla);


/*
    Precondiciones:
        * <<tipoD>> fue creado.
        * <<archivo>> fue abierto en modo "wb".
    Detalle: Guarda el TipoDupla en el archivo.
*/
void GuardarTipoDupla(TipoDupla tipoD, FILE *archivo);
```

```
/*
    Precondiciones:
        * <<tipoD>> fue creado.
        * <<archivo>> fue abierto en modo "wb".
    Detalle: Carga un TipoDupla desde el archivo.
*/
void CargarTipoDupla(TipoDupla &tipoD, FILE *archivo);

#endif
```

DuplaEnteroEntero.h

```
#ifndef DUPLA_ENTERO_ENTERO_H
#define DUPLA_ENTERO_ENTERO_H


#include "../ListaPalabras.h"


typedef struct{
int izq;
int der;
} DuplaEnteroEntero;


/*
Precondición: <<argumentos>> tiene dos palabras de numeros enteros.
Detalle: A partir de una ListaPalabras con dos palabras que contienen un
numero entero cada una,
devuelve por copia una DuplaEnteroEntero.
*/
DuplaEnteroEntero ParsearDuplaEnteroEntero(ListaPalabras argumentos);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve el entero izquierdo de la dupla.
*/
int DarIzqDuplaEnteroEntero(DuplaEnteroEntero dupla);


/*
  Precondición: <<dupla>> fue creada.
    Detalle: Devuelve el entero derecho de la dupla.
*/
int  DarDerDuplaEnteroEntero(DuplaEnteroEntero dupla);
```

```
/*
    Precondiciones:
        * <<archivo>> fue abierto en modo escritura de bytes "wb" o "ab".
        * <<dupla>> NO fue creada.
    Detalle: Carga una DuplaEnteroEntero desde el disco.
*/
void GuardarDuplaEnteroEntero(DuplaEnteroEntero dupla, FILE* archivo);


/*
    Precondiciones:
        * <<archivo>> fue abierto en modo lectura "rb" y contiene los datos
binarios de una DuplaEnteroEntero.
        * <<dupla>> NO fue creada.
    Detalle: Carga una DuplaEnteroEntero desde el disco.
*/
void CargarDuplaEnteroEntero(DuplaEnteroEntero &dupla, FILE* archivo);


#endif
```

DuplaEnteroVariable.h

```
#ifndef DUPLA_ENTERO_VARIABLE_H
#define DUPLA_ENTERO_VARIABLE_H


#include "../ABBVariables.h" // Provee ListaPalabras


typedef struct{
int ent;
String var;
} DuplaEnteroVariable;

/*
Precondición: <<argumentos>> tiene dos palabras, la primera es un numero
entero y la segunda es nombre de
variable válido.
Detalle: A partir de los argumentos crea y devuelve por referencia una
DuplaEnteroVariable, y también el
código de error correspondiente a la operación. Verifica que la variable
perteneza al árbol de
variables.
*/
void ParsearDuplaEnteroVariable(ListaPalabras argumentos, ABBVariables
variables, CodigoError &codigoError, DuplaEnteroVariable
&duplaEnteroVariable);

/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve el entero de la dupla.
*/
int DarEnteroDuplaEnteroVariable ( DuplaEnteroVariable dupla);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve la variable de la dupla.
```

```
*/
ConstString DarVariableDuplaEnteroVariable(DuplaEnteroVariable dupla);


/*
    Precondiciones:
        * <<dupla>> no fue creada.
        * <<archivo>> fue abierto en modo "wb".
    Detalle: Carga una DuplaEnteroVariable desde el disco.
*/
void CargarDuplaEnteroVariable (DuplaEnteroVariable &dupla, FILE* archivo);


/*
    Precondiciones:
        * <<dupla>> no fue creada.
        * <<archivo>> fue abierto en modo "rb".
    Detalle: Guarda una DuplaEnteroVariable en el disco.
*/
void GuardarDuplaEnteroVariable (DuplaEnteroVariable dupla, FILE* archivo);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Libera la memoria dinámica asociada a la dupla.
*/
void LiberarDuplaEnteroVariable(DuplaEnteroVariable &dupla);


#endif
```

DuplaVariableEntero.h

```
#ifndef DUPLA_VARIABLE_ENTERO_H
#define DUPLA_VARIABLE_ENTERO_H


#include "../ABBVariables.h" // Provee ListaPalabras


typedef struct{
String parametroA;
int parametroB;
} DuplaVariableEntero;

/*
Precondición: <<argumentos>> tiene dos palabras, la primera es un nombre de
variable válido y la segunda
es un numero entero.
Detalle: A partir de los argumentos crea y devuelve por referencia una
DuplaVariableEntero, y también
el código de error correspondiente a la operación.
*/
void ParsearDuplaVariableEntero(ListaPalabras argumentos, CodigoError
&codigoError,
DuplaVariableEntero &duplaVariableEntero, ABBVariables abbVar);

/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve el entero de la dupla.
*/
int DarEnteroDuplaVariableEntero ( DuplaVariableEntero dupla);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve la variable de la dupla.
*/
ConstString DarVariableDuplaVariableEntero (DuplaVariableEntero dupla);
```

```
/*
    Precondiciones:
        * <<dupla>> no fue creada.
        * <<archivo>> fue abierto en modo "wb".
    Detalle: Carga una DuplaVariableEntero desde el disco.
*/
void CargarDuplaVariableEntero (DuplaVariableEntero &dupla, FILE* archivo);

/*
    Precondiciones:
        * <<dupla>> no fue creada.
        * <<archivo>> fue abierto en modo "rb".
    Detalle: Guarda una DuplaVariableEntero en el disco.
*/
void GuardarDuplaVariableEntero (DuplaVariableEntero dupla, FILE* archivo);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Libera la memoria dinámica asociada a la dupla.
*/
void LiberarDuplaVariableEntero(DuplaVariableEntero &dupla);



#endif
```

DuplaVariableVariable.h

```
#ifndef DUPLA_VARIABLE_VARIABLE_H
#define DUPLA_VARIABLE_VARIABLE_H


#include "../ABBVariables.h" // Provee ListaPalabras


typedef struct{
String parametroA;
String parametroB;
} DuplaVariableVariable;

/*
Precondición: <<argumentos>> tiene dos palabras, la primera es un nombre de
variable válido y la segunda
es un numero entero.
Detalle: A partir de los argumentos crea y devuelve por referencia una
DuplaVariableEntero, y también
el código de error correspondiente a la operación.
*/
void ParsearDuplaVariableEntero(ListaPalabras argumentos, CodigoError
&codigoError,
DuplaVariableEntero &duplaVariableEntero, ABBVariables abbVar);

/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve la variable derecha de la dupla.
*/
ConstString DarDerDuplaVariableVariable (DuplaVariableVariable dupla);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Devuelve la variable izquierda de la dupla.
```

```
*/
ConstString DarIzqDuplaVariableVariable (DuplaVariableVariable dupla);


/*
  Precondiciones:
       * <<dupla>> no fue creada.
       * <<archivo>> fue abierto en modo "wb".
    Detalle: Carga una DuplaVariableVariable desde el disco.
*/
void CargarDuplaVariableVariable (DuplaVariableVariable &dupla, FILE* archivo);


/*
    Precondiciones:
       * <<dupla>> no fue creada.
       * <<archivo>> fue abierto en modo "rb".
    Detalle: Guarda una DuplaVariableVariable en el disco.
*/
void GuardarDuplaVariableVariable (DuplaVariableVariable dupla, FILE* archivo);


/*
    Precondición: <<dupla>> fue creada.
    Detalle: Libera la memoria dinámica asociada a la dupla.
*/
void LiberarDuplaVariableVariable(DuplaVariableVariable &dupla);


#endif
```

LlamadaFuncion.h

```
#ifndef LLAMADA_FUNCION_H
#define LLAMADA_FUNCION_H

#include "FuncionPredefinida.h"
#include "TipoDupla.h" // Provee ListaPalabras
#include "duplas/DuplaEnteroEntero.h" // Provee ABBVariables
#include "duplas/DuplaEnteroVariable.h"
#include "duplas/DuplaVariableEntero.h"
#include "duplas/DuplaVariableVariable.h"
#include "ABBVariables.h"

typedef struct{
FuncionPredefinida funcion;
TipoDupla tipoDupla; // Discriminante

union{
DuplaEnteroEntero enteroEntero; // tipoDupla = ENTERO_ENTERO
DuplaEnteroVariable enteroVariable; // tipoDupla = ENTERO_VARIABLE
DuplaVariableEntero variableEntero; // tipoDupla = VARIABLE_ENTERO
DuplaVariableVariable variableVariable; // tipoDupla = VARIABLE_VARIABLE
};
} LlamadaFuncion;

/*
Precondiciones:
* <<palabras>> contiene tres palabras.
* <<variables>> fue creado.
Detalle: A partir de <<palabras>> crea <<llamadaFuncion>>, creando la dupla
de parámetros correspondiente.
*/
void ParsearLlamadaFuncion(ListaPalabras palabras, ABBVariables variables,
CodigoError codigoError, LlamadaFuncion &llamadaFuncion);

/*
Precondicion: <<llamadaFuncion>> fue creada.
Detalle: Libera la memoria dimámica asociada con la llamada.
```

```
*/
void LiberarLlamadaFuncion(LlamadaFuncion &llamadaFuncion);


/*
    Precondición: <<llamadaF>> fue creada.
    Detalle: Devuelve la función predefinida de la llamada.
*/
FuncionPredefinida DarFuncionPredefinidaLlamadaFuncion(LlamadaFuncion llamadaF);

/*
    Precondición: <<llamadaF>> fue creada.
    Detalle: Devuelve el tipo de dupla de la llamada.
*/
TipoDupla DarTipoDuplaLlamadaFuncion(LlamadaFuncion llamadaF);


/*
    Precondición: <<llamadaF>> fue creada.
    Detalle: Devuelve el resultado de una llamada a funcion
*/
void CalcularResultadoFuncion(LlamadaFuncion llamadaF, ABBVariables vars,  int &resu,
CodigoError &cError);


/*
    Precondición: <<llamadaF>> fue creada.
    Detalle: <<archivo>> fue abierto en modo "wb".
*/
void GuardarLlamadaFuncion(LlamadaFuncion llamadaF, FILE * archivo);


/*
    Precondición: <<llamadaF>> fue creada.
    Detalle: <<archivo>> fue abierto en modo "rb".
*/
void CargarLlamadaFuncion(LlamadaFuncion &llamadaF, FILE * archivo);

#endif
```

FormaParametro.h

```
#ifndef FORMA_PARAMETRO_H
#define FORMA_PARAMETRO_H

#include "CodigoError.h" // Provee ConstString
#include "FuncionPredefinida.h"
#include "Variable.h"

typedef enum{
ENTERO = 0,
VARIABLE = 1,
FUNCION = 2
} FormaParametro;
/*
Precondición: <<str>> es la primer palabra del lado derecho de una
Asignacion.
    Detalle: A partir de <<str>> determina la forma del parámetro de una
asignación. Devuelve la forma y el código de error generado por referencia.
*/
void ParsearFormaParametro(ConstString str, CodigoError &codigo,
FormaParametro &forma);

/*
    Precondición: <<archivo>> ya fue abierto en modo escritura.
    Detalle: Guarda la forma del parametro en un archivo.
*/
void GuardarFormaParametro(FormaParametro forma, FILE *archivo);


/*
    Precondición: <<archivo>> ya fue abierto en modo lectura.
    Detalle: Guarda la forma del parametro en un archivo.
*/
void CargarFormaParametro(FormaParametro &forma, FILE *archivo);


#endif
```

Asignacion.h

```
#ifndef ASIGNACION_H
#define ASIGNACION_H


#include "FormaParametro.h" // Provee ListaPalabras, CodigoError,
ConstString
#include "LlamadaFuncion.h" // Provee ABBVariables


typedef struct{
String variableAsg;
FormaParametro formaParametroDer;

union{
int enteroDer; // formaParametroDer = ENTERO
String variableDer; // formaParametroDer = VARIABLE
LlamadaFuncion llamada; // formaParametroDer = FUNCION
};
} Asignacion;


/*
 Precondiciones:
        * <<palabras>> contiene o bien 3 o 5 palabras (posibles palabras en
asignaciones bien formadas).
        * La primer palabra de <<palabras>> es un nombre de variable
válido, y la segunda es "=".
        * <<asignacion>> NO fue creada.
    Detalle: Crea <<asignacion>> a partir de una lista con una cantidad
suficiente de palabras.
             Valida que las variables manejadas en la asignación existan en
<<variables>>, y devuelve por
             referencia la asignación creada y el código de error
correspondiente.
*/
```

```
void ParsearAsignacion(ListaPalabras palabras, ABBVariables variables,
CodigoError &codigoError, Asignacion &asignacion);


/*
    Precondición: <<asignacion>> fue creada.
    Detalle: Devuelve la forma del parámetro derecho de la asignacion
*/
FormaParametro DarFormaParametroAsignacion(Asignacion asignacion);



/*
    Precondición: <<asignacion>> fue creada.
    Detalle: Devuelve en modo de solo lectura el nombre de la variable
siendo asignada.
*/
ConstString DarNombreVariableAsignacion(Asignacion asignacion);


/*
    Precondiciones:
        * <<asignacion>> fue creada.
        * <<asignacion.formaParametroDer>> es <<ENTERO>>.
    Detalle: Devuelve el lado derecho de <<asignacion>> cuando éste es de
tipo <<ENTERO>>.
*/
int DarEnteroDerAsignacion(Asignacion asignacion);


/*
    Precondiciones:
        * <<asignacion>> fue creada.
        * <<asignacion.formaParametroDer>> es <<VARIABLE>>.
    Detalle: Devuelve el lado derecho de <<asignacion>> cuando éste es de
tipo <<VARIABLE>>. Se devuelve el
            nombre de la variable en modo de solo lectura.
*/
ConstString DarVariableDerAsignacion(Asignacion asignacion);
```

```
/*
    Precondiciones:
        * <<asignacion>> fue creada.
        * <<asignacion.formaParametroDer>> es <<FUNCION>>.
    Detalle: Devuelve el lado derecho de <<asignacion>> cuando éste es de
tipo <<FUNCION>>.
*/
LlamadaFuncion DarLlamadaFuncionDerAsignacion(Asignacion asignacion);


/*
    Precondiciones:
        * <<archivo>> fue abierto en modo escritura "wb".
        * <<asignacion>> fue creada.
    Detalle: Guarda en el archivo una asignación.
*/
void GuardarAsignacion(Asignacion asignacion, FILE* archivo);


/*
    Precondiciones:
        * <<archivo>> fue abierto en modo lectura "rb".
        * <<asignacion>> fue creada.
    Detalle: Carga desde el archivo una asignación.
*/
void CargarAsignacion(Asignacion &asignacion, FILE* archivo);


/*
    Precondiciones: <<asignacion>> fue cargada.
    Detalle: Libera toda memoria dinámica asociada a una asignación.
*/
void LiberarAsignacion(Asignacion &asignacion);



#endif
```

TipoInstruccion.h

```
#ifndef TIPO_INSTRUCCION_H
#define TIPO_INSTRUCCION_H


#include "CodigoError.h"
#include "ListaPalabras.h"
#include "Variables.h"


typedef enum{
LEER = 0,
MOSTRAR = 1,
ASIGNAR = 2
} TipoInstruccion;


/*
Detalle: A partir de una lista de palabras detecta el tipo de instrucción y
lo devuelve por referencia
junto con el código de error.
*/
void ParsearTipoInstruccion(ListaPalabras palabras,
CodigoError &codigoError, TipoInstruccion &tipoInstruccion);


/*
    Precondiciones:
        * <<tInst>> fue cargada.
        * <<archivo>> fue abierto en modo "wb".
    Detalle: Guarda el tipo de instrucción en el disco.
*/
void GuardarTipoInstruccion(TipoInstruccion tInst, FILE * archivo);
```

```
/*
    Precondición: <<archivo>> fue abierto en modo "rb".
    Detalle: Carga un tipo de instrucción desde el disco.
*/
void CargarTipoInstruccion(TipoInstruccion &tInst, FILE * archivo);

#endif
```

Instruccion.h

```
#ifndef INSTRUCCION_H
#define INSTRUCCION_H


#include "TipoInstruccion.h" // Provee String, ListaPalabras, CodigoError
#include "Asignacion.h" // Provee ABBVariables


typedef struct{
TipoInstruccion tipoInstr;

union{
String var; // tipoInstruccion == LEER o MOSTRAR
Asignacion asignacion; // tipoInstruccion == ASIGNAR
};
} Instruccion;


/*
Precondición: <<variables>> y <<palabras>> están creadas.
Detalle: Crea una instrucción a partir de una lista de palabras, y devuelve
la instrucción con el código
de error correspondiente por referencia.
*/
void ParsearInstruccion(ListaPalabras palabras, ABBVariables variables,
CodigoError &codigoError, Instruccion &instruccion);
/*
    Precondición: <<instr>> fue creada.
    Detalle: Devuelve el tipo de instrucción de la Instrucción.
*/
TipoInstruccion DarTipoInstruccion(Instruccion instr);


/*
    Precondiciones:
        * <<instr>> fue creada.
```

```
        * <<abb>> fue creado.
    Detalle: Ejecuta la instrucción leyendo, mostrando o modificando el
valor de la variable correspondiente.
             Devuelve el código de error resultante por referencia.
*/
void EjecutarInstruccion(Instruccion instr, ABBVariables &abb, CodigoError &cError);


/*
    Precondiciones:
        * <<instr>> fue creada.
        * <<archivo>> fue abierto en modo "wb".
    Detalles: Guarda la instrucción en el disco.
*/
void GuardarInstruccion(Instruccion instr, FILE * archivo);


/*
    Precondiciones:
        * <<instr>> fue creada.
        * <<archivo>> fue abierto en modo "wb".
    Detalles: Carga una instrucción desde el disco.
*/
void CargarInstruccion(Instruccion &instr, FILE * archivo);


/*
    Precondición: <<instr>> fue creada.
    Detalle: Libera la memoria asociada a la Instruccion.
*/
void LiberarInstruccion(Instruccion &instr);


#endif
```

ListaInstrucciones.h

```
#ifndef LISTA_INSTRUCCIONES_H
#define LISTA_INSTRUCCIONES_H


#include <stdio.h> // Provee FILE
#include "Instruccion.h" // Provee ABBVariables, CodigoError


typedef struct nodoLista{ // Nombre para autoreferencia
nodoLista* sig;
  Instruccion instruccion;
} NodoListaInstrucciones; // Nodo de lista de instrucciones


typedef NodoListaInstrucciones* ListaInstrucciones; // Primero en lista de
instrucciones


/*
Precondiciones:
* <<archivoFuente>> es archivo de programa de CalcuSimple del que ya se
leyó hasta la línea de
variables.
* <<variables>> fue creado.
* <<instrucciones>> NO fue creado.
Detalle: Parsea el resto del <<archivoFuente>> y construye la lista de
instrucciones del programa.
Devuelve la lista y el código de error correspondiente por referencia.
*/
void ParsearListaInstrucciones(FILE* archivoFuente, ABBVariables variables,
CodigoError &codigoError, ListaInstrucciones &instrucciones);

/*
    Precondiciones:
        * <<lista>> esta creada
    Detalle: Inserta la instruccion al comienzo de la lista
```

```
*/
void InsertarComienzoListaInstrucciones(ListaInstrucciones &lista, Instruccion inst);


/*
    Precondiciones:
        * <<lista>> esta creada
    Detalle: Inserta la instruccion al final de la lista

*/
void InsertarFinalListaInstrucciones(ListaInstrucciones &lista, Instruccion inst);

//Crea una lista vacia
void CrearListaInstrucciones(ListaInstrucciones &listaInstrucciones);

//Devuelve La primera instruccion de la lista
Instruccion DarPrimeroListaInstrucciones(ListaInstrucciones &lista);

//elimina la primer instruccion de la lista
void EliminarPrimeroListaInstrucciones(ListaInstrucciones &lista);

//Libera la memoria eliminando los nodos de la lista
void EliminarListaInstrucciones(ListaInstrucciones &lista);

//Dada una lista, nos dice si la misma esta vacia
Boolean EsVaciaListaInstrucciones(ListaInstrucciones &lista);

void EjecutrarListaInstruccion(ListaInstrucciones lista, ABBVariables abb, CodigoError &cError);


/*
    Precondiciones:
        * <<lista>> esta creada y no es vacia
    Detalle: Inserta la instruccion al comienzo de la lista
    GuardarListaInstrucciones(archivoInst,prog.instrucciones);
    Guarda de forma recursiva y deja la lista invertida para que a la hora
de leer desde
    el archivo sea más eficiente el alogritmo
*/
void GuardarListaInstrucciones(ListaInstrucciones instrucciones, FILE * archivo);
```

```
/*
    Precondiciones:
        * <<archivo>> está abierto en modo escritura binaria.
        * <<instrucciones>> fue creada.
    Detalles: Carga desde el disco una lista de instrucciones previamente
compilada y la devuelve por
              referencia.
*/
void CargarListaInstrucciones(ListaInstrucciones &instrucciones, FILE* archivo);



/*
    Precondición: <<instrucciones>> fue creada.
    Detalle: Libera la memoria dinámica asociada a la lista de
instrucciones y la deja vacia
*/
void LiberarListaInstrucciones(ListaInstrucciones &instruciones);

#endif
```

Programa.h

```
#ifndef PROGRAMA_H
#define PROGRAMA_H


#include "ListaInstrucciones.h" // Provee FILE, Boolean, String,
ConstString
#include "ABBVariables.h"


typedef struct{
String nombre;
ABBVariables variables;
ListaInstrucciones instrucciones;
} Programa;


/*
    Precondiciones:
        * <<nombre>> es un nombre de programa válido.
        * <<prog>> NO fue cargado.
    Detalle: Verifica que exista el archivo "nombre.csim" y lo abre en modo
lectura. Lee la primer línea y
             verifica que <<nombre>> sea igual al nombre que aparece en
esta línea.
             Invoca los procedimientos de parsear las variables e
instrucciones, y devuelve el código de error
             correspondiente.
*/
void ParsearPrograma(ConstString nombre, CodigoError &codigoError, Programa &prog);


/*
    Precondición: <<prog>> fue creado.
    Detalle: Devuelve el nombre del programa.
*/
ConstString DarNombrePrograma(Programa prog);
```

```
/*
    Precondición: <<prog>> fue creado.
    Detalle: Devuelve el ABBVariables del programa.
*/
ABBVariables DarABBVariablesPrograma (Programa prog);


/*
    Precondición: <<prog>> fue creado.
    Detalle: Devuelve la ListaInstrucciones del programa.
*/
ListaInstrucciones DarListaInstruccionesPrograma(Programa prog);


/*
    Precondición: <<prog>> fue cargado.
    Detalle: Ejecuta todas las instrucciones del programa.
*/
void EjecutarPrograma(ConstString nombrePrograma, CodigoError &cError, Programa &prog);


/*
    Precondición: <<nombre>> fue cargado.
    Detalle:
    + Dado un nombre y el tipo de archivo que busca y si es Lectura o
Escritura, devuelve un puntero
        al archivo los tipos son:
        - csim: archivo de texto elaborado por el usuario
        - vars: archivo donde se almacena el ABB de variables de la
compilacion
        - inst: archivo donde se almacena la linsta de instrucciones
compiladas
    + isRead pudes ser TRUE si es Lectura o FALSE si es Escritura. La
apertura de tipo append no se maneja
        ya que cada vez que se escribe en disco se debe arrancar de 0 todo
el archivo
*/
void AbrirArchivoPrograma(ConstString nombre, ConstString tipo, Boolean isRead, FILE *
&archivo);
```

```
/*
    Precondiciones:
        * <<prog>> fue creado.
    Detalle: Abre los archivos de variables e instrucciones y ejecuta los
procedimientos de guardar el árbol
            de variables y la lista de instrucciones en dichos archivos.
Devuelve por referencia el código de
            error correspondiente.
*/
void GuardarPrograma(CodigoError &cError, Programa prog);


/*
    Precondiciones:
        * <<nombre>> es un nombre de programa válido.
        * <<prog>> NO fue creado.
    Detalle: Carga un programa desde los archivos de variables
"nomProg.vars" e instrucciones "nomProg.inst".
             Devuelve por referencia cualquier error resultante.
*/
void CargarPrograma(ConstString nombre, CodigoError &cError, Programa &prog);


/*
    Precondición: <<prog>> fue cargado.
    Detalle: Libera la memoria dinámica asociada a <<prog>>.
*/
void LiberarPrograma(Programa &prog);

#endif
```

Tipo Operacion.h

```
#ifndef TIPO_OPERACION_H
#define TIPO_OPERACION_H


/*
El compilador ofrece un menú de operación.
*/
typedef enum {COMPILAR, EJECUTAR, SALIR, AYUDA, AYUDA_COMPILAR,
AYUDA_EJECUTAR, INFORMACION, EDITAR} TipoOperacion;


/*
    Precondición: <<listaPal>> fue creada.
    Detalle: Parsea el tipo de operación a partir de una lista de palabras.
Devuelve por referencia el código
             de error correspondiente.
*/
void ParsearTipoOperacion(ListaPalabras listaPal, CodigoError &cError, TipoOperacion &tipoOp);


#endif
```

Operacion Consola.h

```
#ifndef OPERACION_CONSOLA_H
#define OPERACION_CONSOLA_H


#include "Programa.h" // Provee ListaPalabras, CodigoError
#include "TipoOperacion.h"
#include <stdlib.h>

typedef struct {
  TipoOperacion tipoOp; // Discriminante
  String dire; // Usado solo en operaciones COMPILAR y EJECUTAR
  String nomProg; // Usado solo en operaciones COMPILAR y EJECUTAR
} OperacionConsola;


/*
    Precondiciones:
        * <<listaPal>> fue creada.
        * <<opCon>> NO fue creada.
    Detalle: Requiere a traves de la consola que el usuario escoja la
siguiente operación.
*/
void ParsearOperacionConsola(ListaPalabras listaPal, CodigoError &cError,
OperacionConsola &opCon);


/*
    Precondición: <<opCon>> fue creada.
    Detalle: Ejecuta la operación correspondiente.
*/
void EjecutarOperacionConsola(OperacionConsola opCon);

/*
    Precondición: <<opCon>> fue creada.
    Detalle: Libera la memoria asociada a la operación.
*/
```

```
void LiberarOperacionConsola(OperacionConsola &opCon);
#endif
```