Universidad de Zaragoza

Procesadores de Lenguajes

Compilador de MiniLeng

Informe sobre el desarrollo del compilador

Fernando Peña Bes (NIA: 756012)

Zaragoza, España

Curso 2019 – 2020

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

En este informe de documenta el desarrollo de un compilador para el lenguaje MiniLeng durante la asignatura Procesadores de Lenguajes.

MiniLeng es un lenguaje procedural, estructurado y fuertemente tipado, cuya sintaxis está inspirada en Pascal y C.

El lenguaje es *case-insensitive* y soporta comentarios tanto de una sola línea como multilínea. Se pueden declarar variables tanto globales como locales de tres tipos de dato: entero, carácter y booleano, y permite la declaración de acciones (procedimientos) anidadas con paso de parámetros tanto por valor como por referencia.

Se pueden utilizar las estructuras de control "mientras que" (while) y "selección" (if), y se soporta la asignación de valores a variables y la evaluación de expresiones aritméticas y lógicas.

El lenguaje tiene dos procedimientos predefinidos: "escribir", que permite escribir por pantalla variables simples y cadenas de caracteres constantes y "leer", que permite leer valores introducidos por el usuario y asignarlos a variables. También tiene dos funciones que permiten transformar un entero a carácter y viceversa: "entacar" y "caraent".

Por último, se ha añadido la posibilidad de trabajar con vectores unidimensionales. Se permite la declaración de vectores, la asignación de valores a las componentes, la asignación directa entre vectores del mismo tamaño, el uso de componentes en expresiones y el paso de componentes y vectores completos como argumentos al invocar una acción.

Un ejemplo de programa de MiniLeng válido semánticamente se puede ver en el Código 1.

```
programa factorialVect;
    entero i, r, v[8];

    accion factorial(val entero n; ref entero r);
    principio
        si n < 2 ent
            r := 1;
        si_no
            factorial(n - 1, r);
            r := n * r;
        fsi
    fin

principio
    i := 0;
    mq i < 8
        factorial(i, v[i]);
        escribir("v[", i, "] = ", v[i], "\n");
        i := i + 1;
    fmq
fin
```

Código 1: Ejemplo de programa en MiniLeng.

Cada vez que se produce un error de compilación, el compilador muestra un mensaje explicativo junto al token más relevante en el error y su número de fila y columna en el programa. Distingue entre errores léxicos, sintácticos y semánticos. Cuando el compilador encuentra un error léxico termina la ejecución, pero cuando el error es sintáctico o semántico, intenta seguir compilando hasta el final del fichero para mostrar la máxima información posible sobre los errores que puede en el programa. También se incluyen avisos de compilación, como *underflow*/*overflow* y divisiones por cero.

Si no se han producido errores durante la compilación del programa (no se tienen en cuenta los avisos), se genera un fichero con el mismo nombre del fichero de entrada y extensión `.code` con el código intermedio que contiene las instrucciones para ejecutar el programa en la máquina P. La generación de código se realiza utilizando el esquema AST, que da flexibilidad a la generación de código y permite introducir optimizaciones interesantes.

Para implementar el compilador, se ha utilizado JavaCC, un metacompilador de código abierto para el lenguaje Java. Se ha desarrollado principalmente, sobre Eclipse 2020-06 con el plugin JavaCC Eclipse Plug-in 1.5.33.

Índice

1. Análisis léxico

El primer paso al construir el compilador es implementar el analizador léxico. Su trabajo es tomar el programa como una cadena de caracteres y devolver un conjunto de *tokens* o símbolos, que son las unidades léxicas del lenguaje.

1.1. Expresión regular utilizada para los identificadores

Según la especificación del lenguaje, los identificadores están compuestos por letras, números y el símbolo '_', no pueden comenzar por número y no pueden terminar en '_'.

Una posible expresión regular que reconoce identificadores es la siguiente:

```
< #DIGITO : [ "0"-"9" ] >
< #LETRA : [ "a"-"z" ] >
< tIDENTIFICADOR :
  (< LETRA > | "_"(< LETRA > | < DIGITO >))(("_")?(< LETRA > | < DIGITO >))* >
```

1.2. Procesamiento de comentarios

En el lenguaje se permiten definir comentarios de una línea (simples) y multilínea. Los comentarios de una línea se indican con el carácter '`%`', todo lo que haya desde ese carácter hasta el final de la línea se considerará como comentario. Los comentarios multilínea empiezan y terminan con la secuencia '`%%`'.

Para implementar los comentarios en JavaCC se han utilizado contextos léxicos. Inicialmente el analizador se encuentra en el contexto `DEFAULT`. Se puede indicar el contexto al que pertenece una declaración léxica entre los signos '`<`' y '`>`', de forma que sólo se tendrá en cuenta cuando el contexto actual sea ese. Para realizar un cambio de contexto después de reconocer una expresión utiliza el signo '`:`'.

El objetivo es filtrar los comentarios para que no sean enviados al analizador sintáctico. JavaCC define la declaración léxica `SKIP` que permite justamente descartar la entrada reconocida. Teniendo estos aspectos en cuenta, la implementación propuesta para procesar comentarios es la siguiente:

```
SKIP : { " "| "\r"| "\t"| "\n"
| < "%"~["%"] > : COMENTARIO
| "%%"        : MULTICOMENTARIO}

// Ignorar comentarios en la entrada
<COMENTARIO> SKIP : {
  "\n" : DEFAULT
}

<COMENTARIO> MORE : {
   < ~[] >
}

<MULTICOMENTARIO> SKIP : {
  "%%" : DEFAULT
}

<MULTICOMENTARIO> MORE : {
   < ~[] >
}
```

Las expresiones reconocidas en las declaraciones `MORE` son añadidas como prefijo a la siguiente expresión reconocida. En este caso se ha utilizado la expresión `< ~[] >` dentro de declaraciones `MORE` para descartar la entrada mientras se está dentro de un contexto de comentario.

1.3. Implementación del modo *verbose*

El compilador dispone de una opción *verbose*. Cuando está activada, se muestra al final de la compilación una trabla con el número de ocurrencias de cada token que se han encontrado en el programa. La tabla tiene la siguiente forma:

```
+-------------------------------------+
| Número de ocurrencias de los tokens |
+-------------------------------------+
| Token                         num.  |
+-------------------------------------+
| Palabras reservadas                 |
|     PROGRAMA                   1    |
|     PRINCIPIO                  3    |
|     FIN                        3    |
|     SI                         3    |
|     ENT                        3    |
|     FSI                        3    |
| Caracteres de agrupación            |
|     PARENTESIS_IZQ            27    |
|     PARENTESIS_DER            27    |
| Operadores aritméticos              |
|     SUMA                       2    |
|     RESTA                      1    |
| Operadores lógicos                  |
|     AND                        2    |
|     OR                         1    |
| Tipos de dato                       |
|     BOOLEANO                   1    |
|     CARACTER                   4    |
+-------------------------------------+
```

Para implementarla se ha creado una clase llamada `TablaOcurrencias` en el paquete `lib.lexico`. Esta clase contiene un tipo enumerado para cada tipo de token (Palabras reservadas, Operadores lógicos, Tipos de dato, Valores...) con los nombres de los tokens que pertenecen al tipo. Cada tipo tiene asociado un vector con un contadores para cada uno de sus tokens, que se van incrementando conforme se reconocen en el código. Para incrementar los contadores, la clase define una función `incrementar` que se llama desde el analizador léxico pasándole el token cuyo contador se quiere incrementar.

Por ejemplo, la expresión del analizador léxico que reconoce la palabra reservada `PROGRAMA`, realiza la siguiente llamada a la clase:

```
TOKEN : /* PALABRAS RESERVADAS */
{
  < tPROGRAMA : "programa" >
  {
    minilengcompiler.tabla_ocurrencias.incrementar(TablaOcurrencias.Reservadas.
        tPROGRAMA);
  }
  ...
}
```

1.4. Gestión de errores léxicos

Si mientras se está procesando un programa se encuentra un error léxico, el compilador termina y muestra un error como el siguiente:

```
ERROR LÉXICO (<línea>, <columna>): símbolo no reconocido: '...' (\u...)
```

Si el carácter que ha producido el error no es ASCII, se muestra escapado con su código Unicode.

En JavaCC, los errores léxicos son capturados como excepciones de tipo `Error` durante la ejecución del compilador. Cuando se captura esta excepción, se toma el carácter que ha producido el error junto con su fila y columna en la entrada y se muestran utilizando una función personalizada (definida en la clase `ErrorLexico` de `lib.lexico`).

Esta función (Código 2) toma como parámetros la línea y columna en la que ha ocurrido el error, el carácter erróneo y el carácter escapado a código Unicode, e imprime el error con el formato mencionado.

```
public static void deteccion(int linea, int columna, String error, String
    error_escapado) {
    contadorErrores++;

    // Imprime el caracter erroneo, si puede no ser imprimible por la terminal,
    // imprime su equivalente escapado.
    System.err.println("ERROR LÉXICO (" + linea +
        ", " + (columna - 1) + "): " +
        "Caracter no reconocido: '" + error + "'" +
        (!error.equals(error_escapado) ? " (" + error_escapado + ")" : "")
    );
}
```

Código 2: Función `ErrorLexico.deteccion`.

2. Análisis sintáctico

En este compilador, el analizador sintáctico guia la compilación. Va pidiendo *tokens* al analizador léxico uno a uno y comprueba que siguen la gramática definida para el lenguaje. Como resultado, genera un árbol de análisis para el programa de entrada. Si durante este proceso se detecta algún token incorrecto según la gramática, se genera un error sintáctico.

2.1. Modificaciones sobre la gramática propuesta y decisiones de diseño

En la gramática propuesta, algunas reglas estaban incompletas. A continuación, se detalla como se han completado y cuáles han sido las decisiones de diseño adoptadas.

2.2. Delimitadores e identificadores

Los tokens delimitadores, identificadores e índices de vectores se han envuelto en reglas, que son llamadas por el resto de reglas de la gramática. Esto facilitará posteriormente el análisis sintáctico y semántico ya que permite escribir sólo una vez el tratamiento que se le da a cada token dentro de la gramática.

```
fin_sentencia ::= <tFIN_SENTENCIA>
sep_variable ::= <tSEP_VARIABLE>
parentesis_izq ::= <tPARENTESIS_IZQ>
parentesis_der ::= <tPARENTESIS_DER>
corchete_izq ::= <tCORCHETE_IZQ>
corchete_der ::= <tCORCHETE_DER>
longitud_const ::= <tCONSTENTERA>
principio ::= <tPRINCIPIO>
fin ::= <tFIN>
identificador ::= <tIDENTIFICADOR>
```

2.3. Declaración de variables

En la parte de la declaración de variables se ha modificador la regla `identificadores` para añadir soporte a vectores de la siguiente forma:

```
identificadores ::= identificador_declaracion ( sep_variable
    identificador_declaracion )*
identificador_declaracion ::= identificador ( corchete_izq longitud_const
    corchete_der )?
```

Se ha añadido la regla `identificador_declaración` para no repetir la definición de los identificadores de variables dos veces. Como resultado, se pueden realizar declaraciones como:

```
entero a, b, c[10], d;
booleano ok;
caracter x[100], y, z[1];
```

El lenguaje sólo admite la declaración del tamaño de un vector con una constante positiva, la regla `longitud_const` reconoce estas constantes.

2.3.1. Parámetros formales

Los parámetros formales definen las entradas y las salidas de una acción, su signatura. Se ha utilizado una serie descendente de reglas para reconocerlos, que termina en los identificadores.

```
cabecera_accion ::=  <tACCION> identificador ( parametros_formales )?

parametros_formales ::= parentesis_izq ( parametros ( fin_sentencia parametros )*
     )? parentesis_der
parametros ::= clase_parametros tipo_variables lista_parametros
lista_parametros ::= identificador_parametro ( sep_variable
    identificador_parametro )*
clase_parametros ::= ( <tVAL> | <tREF> )
identificador_parametro ::= identificador ( corchete_izq longitud_const
    corchete_der )?
```

Los parámetros, al igual que las variables, pueden ser simples o vectores. Para mantener la sintaxis del lenguaje sencilla se ha decidido utilizar una sintaxis similar a la declaración de variables. Los siguientes ejemplo muestran posibles cabeceras de acciones:

```
accion uno;
accion dos(val entero a, b);
accion tres(val entero a, b[10]; ref caracter c, d);
```

Como esta estructura es opcional al invocar una acción, se ha utilizado el símbolo '`?`' después de la regla `parametros_formales` en `cabecera_accion`.

2.3.2. Sentencias

La gramática del lenguaje debe ser $LL(1)$, es decir debe utilizar un *lookahead* de 1. En la gramática propuesta había un conflicto entre las reglas `asignacion` e `invocacion_accion`, ya que ambas tenían como prefijo común un identificador. Para solucionarlo, se ha creado una nueva regla (`identificacion`) que contiene el prefijo, y desde ahí se da la opción de que el identificador sea parte de una asignación o de una invocación a acción. También se ha completado la definición de `lista_asignables`, `lista_escribibles` y `seleccion`:

```
bloque_sentencias ::= principio lista_sentencias fin
lista_sentencias ::= sentencia ( sentencia )*

sentencia ::= ( leer | escribir | identificacion | seleccion | mientras_que )
leer ::= <tLEER> parentesis_izq lista_asignables parentesis_der fin_sentencia
lista_asignables ::= lista_expresiones
escribir ::= <tESCRIBIR> parentesis_izq lista_escribibles parentesis_der
    fin_sentencia
lista_escribibles ::= escribible ( sep_variable escribible )*
escribible ::= ( expresion | <tCONSTCAD> )

mientras_que ::= <tMQ> expresion lista_sentencias <tFMQ>
seleccion ::= <tSI> expresion <tENT> lista_sentencias ( <tSI_NO> lista_sentencias
     )? <tFSI>

identificacion ::= identificador ( ( corchete_izq expresion corchete_der )?
    asignacion fin_sentencia | ( argumentos )? fin_sentencia )
asignacion ::= <tOPAS> expresion
argumentos ::= parentesis_izq ( lista_expresiones )? parentesis_der
```

A la hora de acceder a las componente un vector es posible utilizar expresiones como índices, como se puede ver en la parte de `identificacion` correspondiente a la asignación.

Una lista de asignables se ha definido como una lista de expresiones. Después, durante el análisis semántico, se comprobará si las expresiones son asignables o no. En cuanto a la escritura, es posible escribir tanto expresiones como cadenas de caracteres constantes, así que ha hecho falta utilizar una regla un poco más complicada para reconocer listas de escribibles. La definición de lo que es un "escribible" se recoge en la regla `escribible`.

2.3.3. Expresiones

Las reglas que se han utilizado para reconocen las expresiones son muy similares a las de Pascal. La precedencia de las operaciones se define de forma implícita en la gramática según la clase de los operadores (relacionales, aditivos y multiplicativos). La asociatividad también es implícita, va de izquierda a derecha, ya que es el orden en el que el analizador lee cada subexpresión.

```
lista_expresiones ::= expresion ( sep_variable expresion )*
expresion ::= expresion_simple ( operador_relacional expresion_simple )?
operador_relacional ::= ( <tIGUAL> | <tMENOR> | <tMAYOR> | <tMAI> | <tMEI> | <tNI
    > )
expresion_simple ::= termino ( operador_aditivo termino )*
operador_aditivo ::= ( <tMAS> | <tMENOS> | <tOR> )
termino ::= factor ( operador_multiplicativo factor )*
operador_multiplicativo ::= ( <tPRODUCTO> | <tDIVISION> | <tMOD> | <tDIV> | <tAND
    > )
factor ::= ( <tMENOS> factor | <tMAS> factor | <tNOT> factor | parentesis_izq
    expresion parentesis_der | <tENTACAR> parentesis_izq expresion parentesis_der
     | <tCARAENT> parentesis_izq expresion parentesis_der | identificador (
    corchete_izq expresion corchete_der )? | <tCONSTENTERA> | <tCONSTCHAR> | <
    tTRUE> | <tFALSE> )
```

En la regla `factor` se ha incluido el reconocimiento de acceso a componentes de vectores, de forma similar a la definición de `identificacion`.

2.4. Gestión de errores sintácticos

Los errores sintácticos se producen cuando el analizador sintáctico recibe un token inesperado del analizador léxico. Por defecto JavaCC genera una excepción de tipo `ParseException` que termina con la ejecución del programa. Para poder continuar la compilación a pesar de que se produzcan estas excepciones, se capturan en las reglas de la gramática mediante bloques `try`/`catch` y se imprimen por pantalla utilizando una función personalizada.

Por ejemplo, la regla `bloque_sentencias` está definida de la siguiente manera:

```
void principio() :
{}
{
  try {
    < tPRINCIPIO >
  }
  catch (ParseException e) {
    ErrorSintactico.deteccion(e, "Se esperaba el delimitador de principio de
        bloque: ‘principio’");
  }
}

void fin() :
{}
{
  try {
    < tFIN >
  }
  catch (ParseException e) {
    ErrorSintactico.deteccion(e, "Se esperaba el delimitador fin de bloque: ‘fin
        ’");
  }
}

ListaInstr bloque_sentencias() :
{}
{
  try {
    principio() lista_sentencias() fin()
  }
  catch (ParseException e) {
    ErrorSintactico.deteccion(e, "Se esperaba un bloque se sentencias");
  }
}
```

Como se ha comentado antes, los tokens que aparecen varias veces en la gramática, se envuelto en reglas adicionales para no tener que repetir el tratamiento de las excepciones.

La clase `ErrorSintactico` está definida en el paquete `lib.sintactico` (Código 2). La función `deteccion` toma como parámetros la excepción de análisis semántico capturada y un mensaje explicativo, y se encarga de crear un mensaje de error con la fila y columna del token incorrecto, su imagen y el mensaje explicativo.

```
public static void deteccion(ParseException e, String mensaje) {
    contadorErrores++;

    System.err.println("ERROR SINTÁCTICO (" + e.currentToken.next.beginLine +
           ", " + (e.currentToken.next.beginColumn) + "): Token incorrecto: ’" +
           e.currentToken.next + "’. " + mensaje);
}
```

El mensaje de error final tiene la siguiente forma:

```
ERROR SINTÁCTICO (<línea>, <columna>): Token incorrecto: ‘...’. <mensaje
    explicativo>
```

3. Tabla de símbolos

La tabla de símbolos es una estructura auxiliar que se utiliza sobretodo durante la fase de análisis semántico. En ella se almacena la información asociada a los símbolos declarados en el programa. De esta forma, cuando se utiliza un símbolo durante el programa, es posible conocer si está definido, cuál es su tipo, etc.

Para implementarla se ha utilizado una tabla hash en la que las colisiones se resuelven mediante listas enlazadas.

3.1. Simbolos

El primer paso fue definir lo que es un símbolo. Para ello se creó la clase `lib.semantico.Simbolo` cuyos atributos se pueden ver en el Código 3.

```
public enum Tipo_simbolo {
    PROGRAMA, VARIABLE, ACCION, PARAMETRO
};

// Representa el tipo de variable
public enum Tipo_variable {
    DESCONOCIDO, ENTERO, BOOLEANO, CHAR, CADENA
};

// Representa la clase de los parámetros en las acciones
public enum Clase_parametro {
    VAL, REF
};

////////////////////////////
// Atributos del símbolo //
////////////////////////////

String nombre;
Integer nivel; // Nivel en el que se ha declarado el símbolo (primer nivel = 0)
Integer dir; // Dirección del símbolo

Tipo_simbolo tipo;
Tipo_variable variable;
Clase_parametro parametro;

ArrayList<Simbolo> lista_parametros; // Lista de símbolos que representan los par
    ámetros de una acción

Boolean vector = false; // Vale true si el símbolo es una variable o parametro
    vector
Integer longitud; // Longitud para los vectores

Boolean inicializado = false; // Vale true si el símbolo es una variable o pará
    metro y ha sido inicializado
```

Código 3: Atributos de la clase `lib.semantico.Simbolo`

Para representar los vectores no se han añadido tipos nuevos de símbolo, se ha decidido añadir un booleano (`vector`) que valdrá `true` si el símbolo es una variable o parámetro vector y un entero, que almacenará la longitud del vector.

3.2. Función de hash

Como función de hash se ha escogido la función Pearson. Dada una cadena, es capaz de calcular un hash rápidamente, no hay clases de entradas más propensas a colisiones que otras, las cadenas que difieren en un carácter nunca colisionan y tampoco se generan colisiones de anagramas.

En su versión más sencilla, esta función genera hashes de 8 bits (0-255) y requiere un array de enteros con los números del 0 al 255 ordenados aleatoriamente.

No se espera que el compilador trabaje con programas muy complejos y con gran cantidad identificadores diferentes, así que esta función es adecuada. Su implementación se puede ver en el Código 4.

```
/*
 * Función de hash utilizando el algoritmo de Pearson
 */
private int h(String cadena) {
    int h = 0;
    for (int i = 0; i < cadena.length(); i++) {
        h = T[h ^ cadena.charAt(i)];
    }
    return h;
}
```

Código 4: Función `lib.semantico.TablaSimbolos.h`.

Para generar el vector `T`, se incializa un vector de 256 componentes con los números de 0 al 255 en orden y se permutan utilizando el algoritmo Fisher-Yates. Este algoritmo se muestra en el código 5.

```
/*
 * Algoritmo de Fisher-Yates para permutar aleatoriamente los elementos de un
 * vector
 */
private void mezclaVector(int[] a) {
    Random rnd = new Random();
    for (int i = a.length - 1; i > 0; i--) {
        // Genera un número aleatorio j tal que 0 <= j 0 <= i
        int j = rnd.nextInt(i + 1);

        // Intercambia a[j] y a[i]
        int aux = a[j];
        a[j] = a[i];
        a[i] = aux;
    }
}
```

Código 5: Función `lib.semantico.TablaSimbolos.mezclaVector`.

3.3. Implementación de la tabla y manejo de colisiones

La tabla hash se ha implementado como un array de listas enlazadas. Los nuevos elementos se concatenan al principio de las listas correspondientes a sus hashes. Por tanto, cuando se produce una colisión, el elemento más reciente será el que primero esté en la lista.

En MiniLeng es posible declarar identificadores con el mismo nombre en niveles distintos, de forma que las declaraciones locales oculten a las globales. La estructura de esta tabla es muy útil para obtener este comportamiento. Para encontrar cuál es el último símbolo declarado con un cierto nombre, basta con calcular su hash y recorrer desde el principio la lista enlazada correspondiente al hash hasta encontrar el primer símbolo cuyo nombre es el que se estaba buscando. Cada vez que un símbolo es ocultado por otro con el mismo nombre, se muestra un aviso para que el programador lo tenga en cuenta.

En cada nivel, hay un único espacio de nombres. No se permite introducir símbolos con el mismo nombre en el mismo nivel, ya que habría una ambigüedad a la hora de decidir a qué símbolo se está haciendo referencia en el código.

Se pensó en implementar un espacio de nombres distinto para las acciones y las variables/parámetros, y distinguir el tipo de símbolo según el uso que se le de en una sentencia. Sin embargo, esta alternativa puede llevar a varios problemas. Por ejemplo, si el lenguaje soportara las funciones y la sintaxis para definir el valor a devolver fuera asignarlo al símbolo de la función, no se saber si realmente se está asignando un valor a la función o a una variable con el mismo nombre. Este problema se podría solucionar dando al programador la opción de crear y especificar espacios de nombres como en Pascal o C, aunque esto aumentaría la complejidad la sintaxis del lenguaje.

En los Códigos 6 y 7 se muestra la implementación de las funciones para insertar y buscar símbolos en la tabla de símbolos, siguiendo las ideas anteriores.

```
    /*
     * Si existe un símbolo en la tabla del mismo nivel y con el mismo, nombre,
     * lanza una excepción. De lo contrario, introduce el símbolo pasado como
     * parámetro.
     */
    private Simbolo introducir_simbolo(Simbolo simbolo) throws
        SimboloYaDeclaradoException {
        int clave = h(simbolo.getNombre());

        for (Simbolo s : tabla_hash[clave]) {
            // Si el símbolo ya está declarado en ese mismo nivel, lanzar una
                excepción
            if (s.getNombre().equals(simbolo.getNombre()) && s.getNivel() ==
                simbolo.getNivel()) {
                throw new SimboloYaDeclaradoException();
            }
            // Si hay un símbolo ya declarado con ese nombre en otro nivel,
                mostrar un aviso
            else if (s.getNombre().equals(simbolo.getNombre())) {
                /*
                Aviso.deteccion("El símbolo '" +
                        simbolo.nombre + "' definido en el nivel " + simbolo.nivel
                        + " va a ocultar a otro definido con el mismo nombre en el
                            nivel " + s.nivel + "");
                 */
                Token t = minilengcompiler.token;
                if (simbolo.ES_VECTOR()) {
                    // Evitar que el token sea ']'
                    t.image = simbolo.nombre;
                }
                Aviso.deteccion("Este símbolo, definido en el nivel " + simbolo.
                    nivel +
                        ", va a ocultar a otro definido con el mismo nombre en el
                            nivel " + s.nivel + "",
                        t);
            }
        }

        // Si no, se añade
        tabla_hash[clave].addFirst(simbolo);

        return simbolo;
    }
```

Código 6: Función `lib.semantico.TablaSimbolos.introducir_simbolo`

```
/*
 * Busca en la tabla el símbolo de mayor nivel cuyo nombre coincida con el del
 * parámetro (se distinguen minúsculas y mayúsculas). Si existe, devuelve un
 * puntero como resultado, de lo contrario lanza una excepción.
 */
public Simbolo buscar_simbolo(String nombre) throws SimboloNoEncontradoException
    {
    int clave = h(nombre);
    for (Simbolo s : tabla_hash[clave]) {
        if (s.nombre.equals(nombre)) {
            return s;
        }
    }

    // Si no se ha encontrado
    throw new SimboloNoEncontradoException();
}
```

Código 7: Función `lib.semantico.TablaSimbolos.buscar_simbolo`

4. Análisis semántico

En el análisis semántico se comprueba que el uso de los elementos del lenguaje es correcto, se realizan comprobaciones sobre los tipos o sobre si los símbolos se usan en los lugares adecuados.

4.1. Gestión de errores y avisos (*warnings*) semánticos

Cada vez que se encuentra un error semántico durante la compilación, se imprime un mensaje con la siguiente forma:

```
ERROR SEMÁNTICO (<línea>, <columna>): Símbolo: '...'. <mensaje explicativo>
```

Se señala la línea y columna en la que se encuentra el símbolo en el que se ha detectado el error, la imagen del símbolo y un mensaje explicativo.

Hay algunos usos del lenguaje que pueden llevar a problemas durante la ejecución, estos se tratan como avisos y se muestran con un formato similar.

```
AVISO (<línea>, <columna>): Símbolo: '...'. <mensaje explicativo>
```

La diferencia principal entre ambos es que los avisos no impiden generar el programa de salida, aunque generalmente el programador querrá solucionar esos problemas antes de ejecutarlo.

A continuación se detallan los tipos de errores y avisos que se han tenido en cuenta y los mensajes explicativos que se muestran en cada caso.

4.1.1. Errores

- Utilización de símbolos no declarados. Este error se puede producir al evaluar expresiones, asignaciones o llamadas a acción.

  ```
  El símbolo no está definido
  ```

- Redeclaración de símbolos dentro del mismo nivel. Se puede producir en cualquier tipo de declaración (variables, acciones y parámetros).

  ```
  No se puede redefinir el símbolo
  ```

- No se puede asignar un valor a una expresión no asignable dentro de una sentencia de asignación o en la función 'leer'. No son asignables las expresiones compuestas, ni los parámetros por valor.

  ```
  La expresión es un vector, no es asignable.
  La expresión es un parámetro por valor, no es asignable.
  La expresión no es asignable.
  ```

- Tampoco son asignables las acciones ni el programa que se está compilando.

  ```
  No se puede realizar una asignación a una acción
  No se puede realizar una asignación a un programa
  ```

- Asignaciones en las que los tipos de ambos lados de la asignación no coinciden. Para realizar una asignación directa entre dos vectores, además del tipo debe coincidir la longitud de ambos.

  ```
  Tipos incompatibles en la asignación
  No se puede realizar la asignación directa de vectores de diferente
      longitud
  ```

- No se permite escribir vectores completos con la función 'leer', sólo componentes individuales.

  ```
  No se pueden escribir vectores
  ```

- Las condiciones de las sentencias 'mientras que' y 'selección' deben ser expresiones booleanas.

  ```
  La condición del 'mientras_que' debe ser una expresión booleana
  La condición de la selección debe ser una expresión booleana
  ```

- En las invocaciones a acciones se comprueba que el símbolo invocado sea una acción, se comprueba que el número y tipo de los argumentos coincida con el de los parámetros. También se comprueba si se está pasando un no asignable a un parámetro por referencia.

```
El símbolo no es una acción
La acción require ‘n’ argumentos
El número de argumentos es incorrecto, se esperaban ‘n’
El tipo del argumento ‘i’ no coincide con el del parámetro, se esperaba
    ‘tipo’
El argumento ‘i’ es un parámetro por valor, no es asignable y no se
    puede pasar por referencia
La expresión del argumento ‘i’ no es asignable, así que no se puede
    pasar por referencia
```

- En cuanto a las operaciones, sólo se permiten las operaciones entre operandos del mismo tipo. Las operaciones aritméticas sólo se pueden realizar entre enteros y las lógicas entre booleanos. Las operaciones de comparación se pueden realizar entre enteros y caracteres, para booleanos sólo están definidos los operadores = y <>.

```
Tipo incompatible. Se esperaba ‘tipo’
Los operandos deben ser del mismo tipo
El operando 1 debe ser ‘tipo’
El operando 2 debe ser ‘tipo’
El operador ‘>=’ no esta definido para booleanos
El operador ‘>’ no esta definido para booleanos
El operador ‘<=’ no esta definido para booleanos
El operador ‘<’ no esta definido para booleanos
```

- No se permite operar directamente con vectores, sólo con sus componentes como si fueran variables.

```
No se pueden realizar operaciones con vectores
```

- Las expresiones para acceder a componentes de un vector deben ser enteras.

```
La expresión para acceder a una componente del vector debe ser entera
```

- No se puede acceder a componentes de símbolos que no sean vectores

```
El símbolo no es un vector
```

- Dentro de las expresiones no se pueden utilizar acciones ni el símbolo de programa, sólo variables, parámetros y constantes.

```
No se puede utilizar una acción dentro de una expresión
No se puede utilizar un programa dentro de una expresión
```

- En la operación ‘entacar se comprueba que la expresión a convertir sea entera y que el entero a convertir es representable como un carácter ASCII, es decir su valor está entre 0-255.

```
La expresión no se puede convertir en un carácter válido
La expresión no produce un entero ASCII válido
```

- En la operación ‘caraent’ se comprueba que la expresión a convertir sea de tipo carácter.

```
La expresión no se puede convertir en un entero válido
```

4.1.2. Avisos

- En las sentencias ‘mientras que’ se comprueba si la condición es constante a `true`, ya que se produciría un bucle infinito durante la ejecución. Si por el contrario la condición es constante `false`, el interior del bloque es código muerto.

```
La expresión del 'mientras_que' siempre es 'true', se produce un bucle
    infinito
La expresión del 'mientras_que' siempre es 'false', el interior del
    bloque es código muerto
```

- Si una selección es simple (no hay 'si_no') y la condición es constante `false`, el bloque completo es código muerto. Si hay 'si_no' y la condición es siempre `true`, el bloque del 'si_no' es código muerto; si la condición es siempre `false`, el bloque del 'si' es código muerto.

```
La expresión del 'si' es siempre 'false', el interior del bloque es có
    digo muerto
La expresión de la selección es siempre 'true', el interior del bloque '
    si_no' es código muerto
La expresión de la selección es siempre 'false', el interior del bloque
    'si' es código muerto
```

- Al realizar operaciones aritméticas se comprueba si se va a producir un overflow o un underflow en memoria. También se considera overflow o underflow cuando la expresión de acceso a la componente de un vector es constante y es más grande o más pequeña que los límites del vector.

```
La operación produce overflow
La operación produce underflow
```

- En las operaciones de división `div`, `/` (ambas son equivalentes) y `mod`, se comprueba si el segundo operando es 0.

```
La operación produce una división por cero
```

- Como se comentó en la sección sobre la tabla de símbolos (Sección 3), cada vez que un identificador va a ocultar a otro en un nivel superior, se muestra un aviso.

```
Este símbolo, definido en el nivel 'n2', va a ocultar a otro definido
    con el mismo nombre en el nivel 'n1'
```

También se intentó crear un aviso cuando se intenta acceder al valor de variables no inicializadas, sin embargo, la estructura del lenguaje no permite cubrir todos los casos de una forma sencilla. Por ejemplo, si se lee una acción que accede a variables globales, no se puede saber con una sola pasada si esas variables se inicializan más tarde antes de llamar a la acción.

4.2. Propagación y manejo de valores constantes en las expresiones

Para implementar la propagación de la información sobre el tipo de las expresiones y su valor en caso de que sean constantes, se ha creado la clase `RegistroExpr` en el paquete `lib.semantico`.

La clase tiene los siguientes atributos:

```
private Integer valorEnt;
private Boolean valorBool;
private Character valorChar;
private String valorCad;

private Tipo_variable tipo;
private Clase_parametro parametro;
private Boolean asignable = false;

private Boolean vector = false;
private Integer longitud;

// Se puede almacenar un símbolo cuando la expresión es un
// factor identificador
private Simbolo s;

// Generacion codigo
// Contiene la lista de instrucciones necesaria para calcular la
// expresión que representa el registro
private ListaInstr instrucciones = new ListaInstr();
```

El objetivo es que la regla `expresion` devuelva un `RegistroExpr` con la información asociada a la expresión: tipo, calculado en caso de que sea una expresión constante o la clase si es un parámetro.

Cuando se está procesando una expresión, primero se crean registros para los factores y se van propagando hacia arriba a través de las reglas `termino` y `expresion_simple` hasta llegar a `expresion`. Cada vez que se encuentra una operación, se crea un nuevo `RegistroExpr` con el resultado de la operación. Para operar dos registros mediante una operación binaria se ha añadido un método llamado `operar` en la clase.

Las expresiones son constantes hasta que incluyen un identificador, en ese momento deja de ser posible calcular su valor durante la compilación.

Cuando se encuentran símbolos no declarados dentro de una expresión, se muestra el error semántico correspondiente y el tipo de la expresión cambia a `DESCONOCIDO`, pera evitar que el error vuelva a aparecer posteriormente al intentar utilizarla en otras partes de la gramática.

4.3. Controles en las expresiones

Cuando las expresiones son constantes, se pueden hacer comprobaciones sobre los valores que producen.

El compilador es capaz de detectar la aparición de overflow, underflow o división por cero al realizar operaciones aritméticas con enteros. Esta comprobación se realiza al llamar a `operar` dentro de las funciones auxiliares `hayUndeflowOverflow` y `hayDivisionPorCero`. Cuando se utilizan expresiones constantes para acceder a las componentes de los vectores también se comprueba que el índice esté dentro de los límites del vector.

Como MiniLeng es un lenguaje fuertemente tipado, no se realizan conversiones implícitas entre los tipos, de forma que los tipos deben coincidir al realizar operaciones, asignaciones y llamadas a acciones.

Por último, como la precedencia y asociatividad operaciones están definidias de manera implícita en la gramática las expresiones se evalúan siguiendo el orden correcto de las operaciones sin realizar ningún cambio.

4.4. Comprobaciones sobre parámetros de las acciones

Para comprobar los argumentos al llamar a una acción se utilizan parámetros ocultos. Como se está trabajando con objetos Java, las acciones pueden tener una lista de referencias a los parámetros. Cada vez que se cierra una acción, se eliminan sus parámetros de la tabla de símbolos, pero los objetos siguen referenciados dentro el símbolo de la acción y es posible seguir accediendo a ellos. Se eliminarán definitivamente cuando se elimine el símbolo de la acción de la tabla.

Al analizar la declaración de las acciones es posible que el programador se haya equivocado y haya declarado varios parámetros con el mismo nombre. Esto es un error semántico, ya que en un mismo nivel no puede haber dos símbolos con el mismo nombre. Cuando sucede esto, se muestra el error semántico, pero el parámetro erróneo se añade en la lista de parámetros de la acción bajo el nombre '`_anonymous`', aunque no se añade a la tabla de símbolos. De esta forma más adelante se podrá comprobar si la invocación a la acción es correcta a pesar del error.

Si un parámetro se llama igual que la acción, la situación es diferente, ya que estos se definen en un nivel más de profundidad que la acción. Cuando esto sucede, el parámetro oculta a la acción, y no es posible invocarla recursivamente. Como se ha comentado en la Sección 4.1.2, cada vez que un símbolo es ocultado por otro se muestra un aviso al programador.

El siguiente programa (Código 8) aparecen los dos casos anteriores.

```
programa errores_acciones;

accion error1(val entero a, p, p);
principio
    escribir(a, p, p);
fin

accion error2(val entero a, error2);
principio
    escribir(a, error2);
    error2(1, 2);
fin

principio
    error1(1, 2, 3);
    error2(1, 2);
fin
```

Código 8: Programa `errores_acciones`.

En la declaración de la primera acción hay un parámetro repetido (`p`). El compilador muestra el error correspondiente, y no añade el repetido en la tabla de símbolos, pero si que se apunta en la lista de parámetros de la acción con el nombre '`_anonymous`'. En el bloque principal es posible comprobar que la acción está bien utilizada y no se muestran más errores.

```
ERROR SEMÁNTICO (3, 32): Símbolo: 'p'. No se puede redefinir el símbolo
Antes de cerrar la acción: error1. Nivel 1
+-------------------------------------------------------------------+
| Tabla de símbolos                                                 |
+-------------------------------------------------------------------+
|    63  PARAMETRO VAL ENTERO:   a [1, 3]                           |
|    89  PARAMETRO VAL ENTERO:   p [1, 4]                           |
|   195  ACCION:                 error1(a, p, _anonymus) [0, 0]     |
|   202  PROGRAMA:               errores_acciones [0, -]            |
+-------------------------------------------------------------------+
```

La declaración de la segunda acción contiene un parámetro con el mismo nombre que la acción. Al procesar la cabecera tan sólo se muestra el aviso de que se va a ocultar el símbolo de la acción, pero al intentar invocarla se genera un error semántico ya que no es posible invocar a un parámetro.

```
AVISO (8, 29): Símbolo: ‘error2’. Este símbolo, definido en el nivel 1, va a
    ocultar a otro definido con el mismo nombre en el nivel 0
ERROR SEMÁNTICO (11, 5): Error al invocar a: ‘error2’. El símbolo no es una acció
    n
Antes de cerrar la acción: error2. Nivel 1
+-------------------------------------------------------------+
| Tabla de símbolos                                           |
+-------------------------------------------------------------+
|    17  PARAMETRO VAL ENTERO:   a [1, 3]                     |
|   108  PROGRAMA:               errores_acciones [0, -]      |
|   144  ACCION:                 error1(a, p, _anonymus) [0, 0]  |
|   221  PARAMETRO VAL ENTERO:   error2 [1, 4]                |
|        ACCION:                 error2(a, error2) [0, 2]     |
+-------------------------------------------------------------+
```

Al procesar la invocación a una acción, se comprueba que los argumentos que se han pasado encajen con los parámetros. Se comprueba que coincidan en número, en tipo y que no se pasen por referencias expresiones no asignables (parámetros por valor y expresiones compuestas). Si se reconoce algún error en el paso de argumentos, se muestra el error semántico correspondiente.

Se pueden usar vectores completos como parámetros. En la declaración es necesario especificar el tamaño del vector y la longitud del vector que se quiere pasar debe coincidir con la del parámetro.

En el siguiente programa (Código 9) se define una acción y desde el bloque principal se realiza una serie de invocaciones correctas e incorrectas.

```
programa errores_invocacion;
entero miEnt, miVecEnt[100], miVecEnt2[10];
booleano miBool, miVecBool[10];

accion miAccion(val booleano b, vb[10]; ref entero e, vn[10]);
principio
    escribir("Bien");
fin

principio
    miAccion(miBool); % error
    miAccion(miEnt, miVecBool, miEnt, miVecEnt2); % error
    miAccion((1 = 2) and (2 <> 2), miVecBool, 1 + 2, miVecEnt2); % error
    miAccion(true, miVecBool, miEnt, miVecEnt); % error
    miAccion(miBool, miVecBool, miEnt, miVecEnt2); % bien
fin
```

Código 9: Programa `errores_invocacion`.

1. En la primera invocación no coincide el número de argumentos y parámetros, por lo que se muestra lo siguiente:

   ```
   ERROR SEMÁNTICO (11, 5): Error al invocar a: ’miAccion’. El número de
       argumentos es incorrecto, se esperaban 4
   ```

2. En la segunda invocación, el tipo del primer argumento es incorrecto:

   ```
   ERROR SEMÁNTICO (12, 5): Error al invocar a: ’miAccion’. El tipo del
       argumento 1 no coincide con el del parámetro, se esperaba BOOLEANO
   ```

3. En la tercera, se está intentando pasar en el tercer argumento una expresión como valor por referencia.

   ```
   ERROR SEMÁNTICO (13, 5): Error al invocar a: ’miAccion’. La expresión del
       argumento 3 no es asignable, así que no se puede pasar por referencia:
   ```

4. En la cuarta, no coinciden los tamaños de los vectores del argumento 4:

   ```
   ERROR SEMÁNTICO (14, 5): Error al invocar a: ’miAccion’. El tipo del
       argumento 4 no coincide con el del parámetro, se esperaba ENTERO[10]
   ```

5. La última invocación a la acción es correcta y no se muestra ningún error.

5. Generación de código

Esta es la última fase del compilador. Tras su ejecución se genera un fichero con las instrucciones para ejecutar el programa de MiniLeng en la máquina P.

5.1. Justificación del esquema escogido

Para la fase de generación de código se ha escogido el esquema AST. En esta implementación no se genera el árbol abstracto de sintaxis de forma explicita, si no que se aprovecha el árbol de análisis generado por el analizador sintáctico para mejorar el código final.

Este esquema es mucho más flexible que el secuencial, ya que permite reordenar las instrucciones del programa y eliminar las que no sean necesarias. Cada instrucción del programa para la máquina P se almacena como una cadena de caracteres y estás sólo se imprimen al final de la compilación si no se han encontrado errores en el código.

Las cadenas que representan las instrucciones se van guardando en listas. Se ha creado una clase llamada `ListaInstr` en el paquete `lib.generacioncodigo` con la que se pueden gestionar estas listas. El único atributo de la clase es un `ArrayList<String>` y se incluyen métodos para añadir en la lista todas las instrucciones necesarias para generar el código final.

Los métodos introducen bloques de instrucciones que realizan una determinada función. Se llaman desde el lugar correspondiente de la gramática para generar el conjunto de instrucciones que se necesite.

La idea es que las reglas de la gramática manipulan y devuelven las listas de instrucciones hasta que todas llegan a la regla inicial (`programa`). En este lugar se juntan las listas devueltas y se genera una lista final con todas las instrucciones del programa.

Esta es una interfaz limpia y fácil de modificar, ya que casi toda la generación de código esta centralizada en una clase. Por ejemplo, el siguiente método (Código 10) genera el código necesario para añadir una selección a la lista.

```
public void addSeleccion(ListaInstr expresion, ListaInstr si, ListaInstr sino,
    Integer etiq1, Integer etiq2) {
   // El código generado tiene la siguiente forma:
   //
   //         <expresion>
   //         JMF ETIQ1
   //         <si>
   //         JMP ETIQ2
   // ETIQ1:
   //         <sino>
   // ETIQ2:
   addComentario("Seleccion");
   concatenarLista(expresion);
   addSaltoFalse(etiq1);
   addComentario("Si");
   concatenarLista(si);
   addSaltoIncod(etiq2);
   addEtiqueta(etiq1);
   addComentario("Si no");
   concatenarLista(sino);
   addEtiqueta(etiq2);
}
```

Código 10: Método `lib.generacioncodigo.ListaInstr.addSeleccion`

Toma la listas de instrucciones necesaria para calcular la condición de la selección, además del código que hay que ejecutar en el 'si' y en el 'si_no'.

Si en el análisis semántico se detecta que no hay bloque 'si_no', se puede generar el código utilizando el método del Código 11 para evitar el salto incondicional.

```
public void addSeleccionSimple(ListaInstr expresion, ListaInstr si, Integer etiq)
      {
    // El código generado tiene la siguiente forma:
    //
    //         <expresion>
    //         JMF ETIQ
    //         <si>
    // ETIQ:
    addComentario("Seleccion simple");
    concatenarLista(expresion);
    addSaltoFalse(etiq);
    addComentario("Si");
    concatenarLista(si);
    addEtiqueta(etiq);
}
```

Código 11: Método `lib.generacioncodigo.ListaInstr.addSeleccionSimple`

Esta flexibilidad para mejorar el código dependiendo del programa de entrada no se tiene al utilizar un esquema de generación de código secuencial.

El código que se encarga de gestionar el nivel actual durante la compilación, los números de las etiquetas y las direcciones del bloque de activación, además de la escritura de la lista final en el fichero de salida, se encuentra en la clase `GeneracionCodigo`, también en el paquete `lib.generacioncodigo`.

6. Mejoras introducidas

En esta sección se describen las mejoras que se han hecho sobre el compilador planteado en las prácticas.

6.1. Mejoras en el uso del compilador

Para facilitar al usuario el uso del compilador, se han incluido una serie de opciones extra. Cuando el usuario intenta utilizar opciones inválidas o el fichero a compilar no es correcto, se muestran mensajes de error explicativos. Además, al final de la compilación, se muestra un recuento con el número de errores encontrados de cada tip.

6.1.1. Uso del compilador

El compilador se invoca de la siguiente manera: `minilengcompiler [opciones] fichero`.

El fichero debe tener extensión `.ml` y se ofrecen las siguientes opciones:

`-v, --verbose` Al final de la compilación se muestra un resumen de los tokens utilizados en el programa y el número de ocurrencias de cada uno.

`-p, --panic` Compila utilizando panic mode. El panic mode se activa cada vez que se detecta que falta un token '`;`' su finalidad es intentar que el compilador se recupere del error para evitar que el error afecte a la compilación de las siguientes partes del código. El compilador descarta la entrada hasta encontrar el siguiente '`;`' y continúa compilando desde ese punto.

`-t, --tokens` Se imprimen los tokens conforme se van encontrando durante el análisis léxico.

`-d, --debug` Se muestra la tabla de símbolos antes y después de cerrar cada bloque.

`-h, --help` Muestra el uso del programa y las opciones disponibles.

`--version` Imprime información sobre la versión del compilador.

A continuación, se muestran algunos ejemplos de uso del compilador:

```
minilengcompiler -h
minilengcompiler --version
minilengcompiler -v fichero.ml
minlengcompiler -p -v fichero.ml
minlengcompiler -pv fichero.ml
```

6.1.2. Errores al utilizar el compilador

Si se utiliza una opción inválida o no se especifican correctamente, se muestra el siguiente error:

```
MiniLeng: Opción inválida <error>
```

Si el fichero a compilar no tiene extensión `.ml`, se muestra el error:

```
MiniLeng: El fichero a compilar tiene que tener extensión .ml
         Fichero introducido: '...'
```

Si no se encuentra el fichero a compilar, se indica de la siguiente forma:

```
MiniLeng: No se ha encontrado el fichero '...'
```

Al final de la compilación se muestra un recuento de los errores encontrados:

```
Errores léxicos: ...
Errores sintácticos: ...
Errores semánticos: ...
Avisos: ...
Veces activado panic mode: ...
```

Si ha aparecido algún error durante la compilación se muestra:

```
No se ha podido compilar el programa.
```

En caso contrario:

```
Compilación finalizada. Se ha generado el fichero '...'.
```

Si durante la compilación se ha activado el modo pánico, se muestra el siguiente mensaje para indicar al usuario que es necesario volver a compilar.

```
Se ha activado el panic mode durante la compilación. Corrige los errores y vuelve
     a compilar.
```

El fichero `.code` sólo se escribe si la compilación ha sido exitosa. Si el programa contenía algún error y ya existía un fichero `.code` con el nombre del programa, este se mantiene intacto.

6.2. Mejoras en el análisis léxico

6.2.1. Opción *tokens*

Se ha añadido la opción *tokens* (`-t`, `--tokens`), que muestra por pantalla los nombres de los tokens que se van reconociendo conforme se analiza el programa. Por ejemplo:

```
tPROGRAMA
tIDENTIFICADOR (Valor: mi_programa)
tFIN_SENTENCIA
tENTERO
tIDENTIFICADOR (Valor: n)
tFIN_SENTENCIA
...
tFIN
```

6.3. Mejoras en el análisis sintáctico

6.3.1. Anidamiento de bloques

Se permiten acciones anidadas con cualquier nivel de profundidad. Las declaraciones de acciones anidadas deben colocarse entre la zona de declaración de las variables y la palabra 'principio' de la función padre. Notar que las acciones anidadas son locales a la acción padre, de forma que no son visibles desde el nivel de la acción padre y superiores.

```
accion accion1;
  % Declaración de variables de accion1
  accion accion2;
    % Declaración de variables de accion2
    accion accion3;
      % Declaración de variables de accion3
    principio
      % Sentencias de accion3
    fin
  principio
    % Sentencias de accion2
  fin
principio
  % Sentencias de accion1
fin
```

También se permiten bloques 'seleccion' y 'mientras que' anidados con cualquier nivel de profundidad.

```
SI <condicion> ENT
  % Sentencias
  SI <condicion> ENT
    % Sentencias
  SI_NO
    % Sentencias
  FSI
  % Sentencias
SI_NO
  % Sentencias
FSI

MQ <condicion>
  % Sentencias
  MQ <condicion>
    % Sentencias
  FMQ
  % Sentencias
FMQ
```

Cualquier bloque ('principio/fin', 'seleccion' o 'mientras que') debe contener al menos una sentencia (el punto y coma no es una sentencia). Esto está definido en la gramática del programa, si un bloque está vacío se generará un error sintáctico.

6.3.2. Panic Mode

El compilador dispone de la opción panic (`-p`, `--panic`), que activa el modo pánico durante la compilación. Se entra en este modo cada vez que se produce un error sintáctico porque se esperaba

un punto y coma. El analizador descarta todos los tokens siguientes al error hasta que encuentra un carácter ‘;’ o el fichero se acaba, entonces sale del modo y sigue analizando la entrada.

Cuando se entra en el modo pánico, se informa al usuario con las siguientes líneas:

```
ERROR SINTÁCTICO (<linea>, <columna>): Token incorrecto: ‘...’. Se esperaba ‘;’
PANIC MODE: Iniciado panic mode
```

Y se va mostrando en una línea diferente cada token descartado:

```
> PANIC MODE: Token descartado: ‘...’
```

Cuando se sale del modo, se muestra:

```
  > PANIC MODE (<linea>, <columna>): Se ha encontrado ‘;’
PANIC MODE: Terminado panic mode
```

6.4. Mejoras en el análisis semántico

6.4.1. Opción *debug*

La opción *debug* (`-d`, `--debug`) hace que se muestre la tabla de símbolos antes y después de cerrar cada bloque, para facilitar la depuración del compilador. Ejemplo:

```
   ...
Antes de cerrar la acción: fib. Nivel 1
+-------------------------------------------------------------+
| Tabla de símbolos                                           |
+-------------------------------------------------------------+
|    26  ACCION:                cambiar_de_linea() [0, 0]     |
|    39  VARIABLE ENTERO:       r1 [1, 5]                     |
|    96  ACCION:                fib(n, r) [0, 9]              |
|   128  PARAMETRO VAL ENTERO:  n [1, 3]                      |
|        VARIABLE ENTERO:       n [0, 3]                      |
|   157  PARAMETRO REF ENTERO:  r [1, 4]                      |
|        VARIABLE ENTERO:       r [0, 4]                      |
|   198  ACCION:                dato(dato) [0, 2]             |
|   224  PROGRAMA:              fibbonaci [0, -]              |
|   233  VARIABLE ENTERO:       r2 [1, 6]                     |
+-------------------------------------------------------------+
Después de cerrar la acción
+-------------------------------------------------------------+
| Tabla de símbolos                                           |
+-------------------------------------------------------------+
|    26  ACCION:                cambiar_de_linea() [0, 0]     |
|    96  ACCION:                fib(n, r) [0, 9]              |
|   128  VARIABLE ENTERO:       n [0, 3]                      |
|   157  VARIABLE ENTERO:       r [0, 4]                      |
|   198  ACCION:                dato(dato) [0, 2]             |
|   224  PROGRAMA:              fibbonaci [0, -]              |
+-------------------------------------------------------------+
   ...
```

Los campos de cada símbolo la tabla de símbolos se muestran con el siguiente formato:

```
‘hash’ ‘tipo’:   ‘nombre’ [‘nivel’, ‘dirección’]

* Si el símbolo es una acción:
     ‘nombre’ := ‘nombre acción’(‘nombres parámetros’)
* Si el símbolo es un vector:
     ‘nombre’ := ‘nombre vector’[‘longitud’]
```

Los símbolos están ordenador por el hash de forma descendente. Si el hash coincide, se ordenan descendentemente por nivel y el hash sólo se imprime una vez junto al primer símbolo.

6.5. Mejoras en la generación de código

Gracias al esquema AST ha sido posible introducir una serie de mejoras interesantes, que hacen que el código final sea más pequeño y más rápido en algunos casos.

6.5.1. Bloques mientras que

El código de los bloques mientras que se genera utilizando el esquema mejorado visto en clase:

```
JMF ETIQ1
ETIQ2:
                <sentencias>
ETIQ1:
                <expresion>
                JMP ETIQ2
```

Si la expresión del 'mientras que' es constante y es evaluada a `true`, durante la ejecución se producirá un bucle infinito. Se muestra un aviso al programador, pero genera el código igualmente. Si por el contrario, la expresión siempre es `false`, el código del bloque está muerto por lo que no incluye en el programa final y se muestra un aviso.

6.5.2. Bloques selección

Como se explica en la Sección 5.1, cuando en una selección no hay 'si_no', se genera únicamente el código necesario para el bloque 'si', evitando un salto incondicional.

Cuando la expresión del bloque selección es constante se pueden dar varios escenarios, el objetivo es no generar código innecesario:

- Si la expresión es `true` y no hay 'si_no', el código del 'sí' siempre se ejecuta, por lo que se incluye directamente, eliminando el cálculo del valor de la expresión y los saltos.
- Si la expresión es `true` y hay 'si_no', el código del 'si_no' está muerto, por lo que se incluye directamente el código del 'si' y se muestra un aviso.
- Si la expresión es `false` y no hay 'si_no', no se genera código para la selección y se muestra un aviso.
- Si la expresión es `false` y hay 'si_no', el código del 'si' está muerto, así que se incluye directamente el código del 'si_no' y se muestra un aviso.

6.5.3. Expresiones constantes

Todas las expresiones constantes del programa de entrada se sustituyen por un apilamiento (`STC`) del valor de la expresión contenido en su `RegistroExpr`.

Si el índice para acceder a una componente de un vector es constante, se genera una instrucción `SRF` que calcula la dirección de la componente directamente, sin necesidad de realizar una suma sobre la dirección base. Si por ejemplo hay un vector local de tamaño 10 declarado en la dirección 3 del BA y se sabe que el programa accede a la componente 1, se genera la instrucción:

```
SRF 0 4
```

en vez de:

```
SRF 0 3
STC 1
PLUS
```

Sin embargo esta mejora no es posible cuando se está accediendo a una componente de un vector por referencia, ya que el BA de la acción invocada únicamente contiene la dirección de la primera componente del vector, y esta sólo se conoce durante la ejecución. Si por ejemplo el vector fuera el primer parámetro, se generaría el siguiente código para acceder a la componente 1:

```
SRF 0 3
DRF
STC 1
PLUS
```

Si el vector se pasa por valor, se puede aplicar la mejora porque al pasarlo por valor se realiza una copia completa del vector en el BA.

6.5.4. Función 'escribir'

La escritura de booleanos produce las cadenas `"True"` y `"False"`. Cuando la expresión booleana es constante, se escribe directamente una de las dos cadenas. Si no lo es, se genera una selección que escribe una cadena u otra dependiendo del valor de la expresión durante la ejecución.

En las cadenas constantes se sustituyen las secuencias `\r`, `\t` y `\n` por los caracteres ASCII 13, 9 y 10 respectivamente, para que dar formato al texto sea más sencillo.

7. Pruebas realizadas

En esta sección se detalla la metodología seguida para verificar el funcionamiento del compilador desarrollado en cada fase. Para cada fase se incluyen algunos tests junto a sus resultados, no se tratan de unas pruebas intensivas pero sirven para dar una idea del proceso que se ha seguido en la fase de pruebas.

7.1. Analizador léxico

El objetivo del analizador léxico es que reconozca todos los tokens de la entrada y que se lancen errores léxicos cuando se encuentran caracteres no soportados. Para probarlo, se intentó compilar varios programas diferentes activando la opción *tokens* (`-t`, `--tokens`), que muestra los tokens que se van reconociendo en en análisis, para verificar que todos son reconocidos correctamente.

7.1.1. Test 1

Vamos a considerar que tenemos el siguiente programa:

Al compilar el programa utilizando la opción `-t`, se obtiene la siguiente salida:

7.1.2. Test 2

También se comprobó que si se encuentra algún carácter no soportado por el lenguaje, se produce un error léxico.

7.2. Analizador sintáctico

Para probar el analizador sintáctico la idea es intentar compilar programas con errores y comprobar que se lanza un error en el lugar adecuado.

7.2.1. Test 1

7.2.2. Test 2

7.3. Analizador semántico

En cuanto al analizador semántico, se siguió un procedimiento parecido al del sintáctico.

7.3.1. Test 1

7.3.2. Test 2

7.4. Generación de código

Para la generación de código, se compilaron programas y se analizó el código generado. También se ejecutó en Hendrix utilizando en ensamblador e intérprete proporcionados y esperar el resultado esperado. En esta fase ha sido de gran utilizar generar comentarios con lo que se hace en cada grupo de instrucciones, para saber que partes del código intermedio corresponden con el programa de MiniLeng.

7.4.1. Test 1

7.4.2. Test 2

7.4.3. Test 3

Anexo A Gramática BNF de MiniLeng

En este anexo se incluye la gramática BNF que se ha utilizado en la implementación del compilador. El código de JavaCC que la implementa se encuentra en el fichero `src/analizador/minilengcompiler.jj`. Encima de cada regla no terminal del fichero `.jj` se ha añadido un comentario con la regla BNF correspondiente para facilitar la lectura.

A.1 Tokens

```
<DEFAULT> SKIP : {
" "
| "\r"
| "\t"
| "\n"
| <"%" ~["%"]> : COMENTARIO
| "%%" : MULTICOMENTARIO
}

<COMENTARIO> SKIP : {
"\n" : DEFAULT
}

<COMENTARIO> MORE : {
<~[]>
}

<MULTICOMENTARIO> SKIP : {
"%%" : DEFAULT
}

<MULTICOMENTARIO> MORE : {
<~[]>
}

<DEFAULT> TOKEN : {
  <tPROGRAMA: "programa"> : {}
| <tVAR: "var"> : {}
| <tPRINCIPIO: "principio"> : {}
| <tFIN: "fin"> : {}
| <tSI: "si"> : {}
| <tENT: "ent"> : {}
| <tSI_NO: "si_no"> : {}
| <tFSI: "fsi"> : {}
| <tMQ: "mq"> : {}
| <tFMQ: "fmq"> : {}
| <tESCRIBIR: "escribir"> : {}
| <tLEER: "leer"> : {}
| <tENTACAR: "entacar"> : {}
| <tCARAENT: "caraent"> : {}
| <tACCION: "accion"> : {}
| <tVAL: "val"> : {}
| <tREF: "ref"> : {}
}

<DEFAULT> TOKEN : {
  <tENTERO: "entero"> : {}
| <tBOOLEANO: "booleano"> : {}
| <tCARACTER: "caracter"> : {}
}

<DEFAULT> TOKEN : {
```

```
  <tLLAVE_IZQ: "{"> : {}
| <tLLAVE_DER: "}"> : {}
| <tPARENTESIS_IZQ: "("> : {}
| <tPARENTESIS_DER: ")"> : {}
}

<DEFAULT> TOKEN : {
  <tCORCHETE_IZQ: "["> : {}
| <tCORCHETE_DER: "]"> : {}
}

<DEFAULT> TOKEN : {
  <tOPAS: ":="> : {}
| <tFIN_SENTENCIA: ";"> : {}
| <tSEP_VARIABLE: ","> : {}
| <tMAS: "+"> : {}
| <tMENOS: "-"> : {}
| <tPRODUCTO: "*"> : {}
| <tDIVISION: "/"> : {}
| <tMOD: "mod"> : {}
| <tDIV: "div"> : {}
| <tAND: "and"> : {}
| <tOR: "or"> : {}
| <tNOT: "not"> : {}
| <tMAYOR: ">"> : {}
| <tMENOR: "<"> : {}
| <tIGUAL: "="> : {}
| <tMAI: ">="> : {}
| <tMEI: "<="> : {}
| <tNI: "<>"> : {}
}

<DEFAULT> TOKEN : {
  <#DIGITO: ["0"-"9"]>
| <#LETRA: ["a"-"z"]>
| <tTRUE: "true"> : {}
| <tFALSE: "false"> : {}
| <tIDENTIFICADOR: (<LETRA> | "_" (<LETRA> | <DIGITO>)) (("_")? (<LETRA> | <DIGITO>))*> : {}
| <tCONSTENTERA: (["0"-"9"])+> : {}
| <tCONSTCHAR: "\"" (~["\""])? "\""> : {}
| <tCONSTCAD: "\"" (~["\""])* "\""> : {}
}
```

A.2 No terminales

```
programa ::= <tPROGRAMA> identificador fin_sentencia declaracion_variables
             declaracion_acciones bloque_sentencias <EOF>

fin_sentencia ::= <tFIN_SENTENCIA>
sep_variable ::= <tSEP_VARIABLE>
parentesis_izq ::= <tPARENTESIS_IZQ>
parentesis_der ::= <tPARENTESIS_DER>
corchete_izq ::= <tCORCHETE_IZQ>
corchete_der ::= <tCORCHETE_DER>
longitud_const ::= <tCONSTENTERA>

principio ::= <tPRINCIPIO>
fin ::= <tFIN>
identificador ::= <tIDENTIFICADOR>
declaracion_variables ::= ( declaracion fin_sentencia )*
declaracion ::= tipo_variables identificadores
```

```
tipo_variables ::= ( <tENTERO> | <tCARACTER> | <tBOOLEANO> )
identificadores ::= identificador_declaracion ( sep_variable identificador_declaracion )*
identificador_declaracion ::= identificador ( corchete_izq longitud_const corchete_der )?

declaracion_acciones ::= ( declaracion_accion )*
declaracion_accion ::= cabecera_accion fin_sentencia declaracion_variables
                       declaracion_acciones bloque_sentencias
cabecera_accion ::= <tACCION> identificador ( parametros_formales )?
parametros_formales ::= parentesis_izq ( parametros ( fin_sentencia parametros )* )? parentesis_der
parametros ::= clase_parametros tipo_variables lista_parametros
lista_parametros ::= identificador_parametro ( sep_variable identificador_parametro )*
clase_parametros ::= ( <tVAL> | <tREF> )
identificador_parametro ::= identificador ( corchete_izq longitud_const corchete_der )?

bloque_sentencias ::= principio lista_sentencias fin
lista_sentencias ::= sentencia ( sentencia )*
sentencia ::= ( leer | escribir | identificacion | seleccion | mientras_que )
leer ::= <tLEER> parentesis_izq lista_asignables parentesis_der fin_sentencia
lista_asignables ::= lista_expresiones
escribir ::= <tESCRIBIR> parentesis_izq lista_escribibles parentesis_der fin_sentencia
lista_escribibles ::= escribible ( sep_variable escribible )*
escribible ::= ( expresion | <tCONSTCAD> )
mientras_que ::= <tMQ> expresion lista_sentencias <tFMQ>
seleccion ::= <tSI> expresion <tENT> lista_sentencias ( <tSI_NO> lista_sentencias )? <tFSI>
identificacion ::= identificador ( ( corchete_izq expresion corchete_der )?
                   asignacion fin_sentencia | ( argumentos )? fin_sentencia )
asignacion ::= <tOPAS> expresion
argumentos ::= parentesis_izq ( lista_expresiones )? parentesis_der

lista_expresiones ::= expresion ( sep_variable expresion )*
expresion ::= expresion_simple ( operador_relacional expresion_simple )?
operador_relacional ::= ( <tIGUAL> | <tMENOR> | <tMAYOR> | <tMAI> | <tMEI> | <tNI> )
expresion_simple ::= termino ( operador_aditivo termino )*
operador_aditivo ::= ( <tMAS> | <tMENOS> | <tOR> )
termino ::= factor ( operador_multiplicativo factor )*
operador_multiplicativo ::= ( <tPRODUCTO> | <tDIVISION> | <tMOD> | <tDIV> | <tAND> )
factor ::= ( <tMENOS> factor | <tMAS> factor | <tNOT> factor
             | parentesis_izq expresion parentesis_der
             | <tENTACAR> parentesis_izq expresion parentesis_der
             | <tCARAENT> parentesis_izq expresion parentesis_der
             | identificador ( corchete_izq expresion corchete_der )?
             | <tCONSTENTERA> | <tCONSTCHAR> | <tTRUE> | <tFALSE> )
```

Referencias

[1] *Documentación de JavaCC*. URL: `https://javacc.github.io/javacc/` (visitado 13-07-2020).
[2] Javier Fabra y José Neira. *Material de la asignatura Procesadores de Lenguajes*.