PRÁCTICA DE
PROCESADORES DEL LENGUAJE II
Curso 2022 – 2023
Entrega de Junio

APELLIDOS Y NOMBRE: Isabel Valentina Manzaneque Núñez

DNI: 53902577F

CENTRO ASOCIADO MATRICULADO: Londres

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: Barbastro, Bergara, Calatayud, Cantabria, Centros en el extranjero (Europa), Cervera, La Rioja, Pamplona, Tortosa, Vitoria-Gasteiz

EMAIL DE CONTACTO: imanzaneq3@alumno.uned.es

TELÉFONO DE CONTACTO: +447762347351

¿REALIZAS LA PARTE OPCIONAL? (SÍ o NO): NO

1. El analizador semántico y la comprobación de tipos

El analizador semántico realiza la comprobación de tipos utilizando la tabla de símbolos y la tabla de tipos. Cada vez que se crea un nuevo ámbito, se crean también las tablas asociadas a este, que se irán rellenando según se vayan procesando las declaraciones. Los ámbitos que se generan son los siguientes:

- Un ámbito global del programa
- Un ámbito por cada función
- Un ámbito por cada bloque de código

Tabla de símbolos

La tabla de símbolos de cada ámbito va a almacenar la información de las constantes, variables y funciones (y sus parámetros) declaradas en dicho ámbito. Se muestra como ejemplo una sección de la tabla de símbolos del ámbito global:

```
[SEMANTIC DEBUG] - SYMBOL TABLE [global]
[SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = global, name = suma, type = ENTERO] - {address=65535, value=0}
[SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = global, name = a, type = ENTERO] - {address=65534, value=0}
[SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = global, name = b, type = ENTERO] - {address=65533, value=0}
[SEMANTIC DEBUG] - Symbol - SymbolConstant [scope = global, name = MAX, type = ENTERO] - {value=10}
[SEMANTIC DEBUG] - Symbol - SymbolFunction [scope = global, name = escribeEntero, type = VACIO] - {parametros=[name: x, type: ENTERO]}
[SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = global, name = vecGlob15, type = vectorGlobal15] - {address=65532, value=0}
[SEMANTIC DEBUG] - Symbol - SymbolFunction [scope = global, name = funSuma, type = ENTERO] - {parametros=[name: x, type: ENTERO, name: y, type: ENTERO]}
```

Los elementos guardan información sobre su ámbito, nombre y tipo. Además, las variables guardan también su dirección en memoria y valor de inicialización, las constantes su valor y las funciones información sobre el nombre y el tipo de sus parámetros. Las tablas de símbolos de las funciones guardan información sobre sus parámetros (si esque lo tiene). Como no se ha realizado la parte opcional, la dirección en memoria de los parámetros se ha dejado a 0.

La inserción se realiza llamando a la tabla de símbolos del ámbito en el que se declara el símbolo en cuestión. Se declarará entonces un nuevo símbolo constante, variable, función o parámetro y se le pasará a la tabla se símbolos que hemos invocado. A continuación un ejemplo de la inserción de constantes:

```
SymbolTableIF tablaSimbolos = scope.getSymbolTable();
TypeTableIF tablaTipos = scope.getTypeTable();

// Comprobar que la constante no este ya declarada
if(tablaSimbolos.containsSymbol(id.getLexema())){
    semanticErrorManager.semanticFatalError("Error en linea " + id.getLine() + ": constante '" + id.getLexema() + "' ya declarada");
}

TypeIF tipo = scopeManager.searchType("ENTERO");
SymbolConstant symbolConstant = new SymbolConstant(scope, id.getLexema(), tipo);
symbolConstant.setValue(Integer.parseInt(value.getLexema()));
tablaSimbolos.addSymbol(symbolConstant);
dC.setSymbolConstant(symbolConstant);
```

Se puede ver en este ejemplo cómo se controla que las declaraciones sean únicas. Esto se lleva a cabo de la misma manera con todos los símbolos: no se puede declarar un símbolo que tenga el nombre de otro ya declarado en ese ámbito.

Tabla de tipos

Almacena información de los tipos primitivos, compuestos y las funciones:

```
[SEMANTIC DEBUG] - ** TYPE TABLES **
[SEMANTIC DEBUG] - Type - TypeFunction [scope = global, name = principal] - {}
[SEMANTIC DEBUG] - Type - TypeFunction [scope = global, name = saluda] - {}
[SEMANTIC DEBUG] - Type - TypeSimpleVacio [scope = global, name = VACIO] - {}
[SEMANTIC DEBUG] - Type - TypeSimpleEntero [scope = global, name = ENTERO] - {}
[SEMANTIC DEBUG] - Type - TypeArray [scope = global, name = vectorGlobal10] - {size=10}
```

Los tipos primitivos se declaran al comenzar el programa, nada mas abrir el ámbito global para que estén disponibles para el programador. Los usuarios pueden declarar sus propios tipos compuestos. Cada vez que se declara una función, se guarda también como un tipo en la tabla de tipos. Esta tabla se popula como se puede ver en este ejemplo de declaración de tipos compuestos:

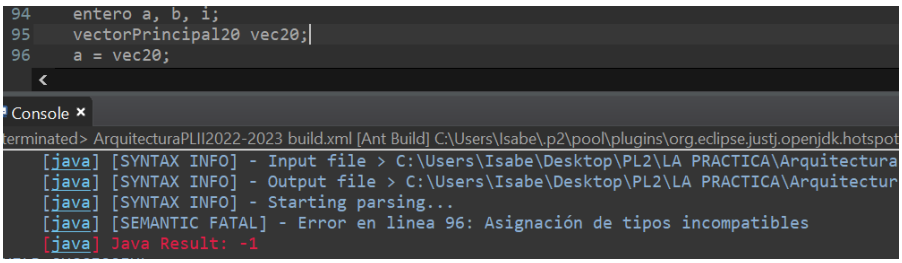
```
ScopeIF scope = scopeManager.getCurrentScope();
TypeTableIF tablaTipos = scope.getTypeTable();

// Comprobar que el tipo no este ya declarado y el tamano sea mayor que 0
int size = (int)tt;
if(scopeManager.containsType(id.getLexema())){
    semanticErrorManager.semanticFatalError("Error en linea " + id.getLine() + ": Tipo '" +id.getLexema()+"' ya declarado");
}
if( size < 1){
    semanticErrorManager.semanticFatalError("Error en linea " + id.getLine() + ": El tamano de un vector debe ser mayor que 0");
}

// Introducir tipo en tabla de tipos
TypeArray tipoVector = new TypeArray(scope, id.getLexema(), size);
tablaTipos.addType(id.getLexema(), tipoVector);
```

Se llama a la tabla de tipos del ámbito en el que nos encontramos y se introduce el nuevo tipo declarado. En este ejemplo se puede ver como se controla que no exista duplicidad de tipos y, además, como los tipos vector declarados por los usuarios deben tener un tamaño mayor a 0.

Comprobación de tipos

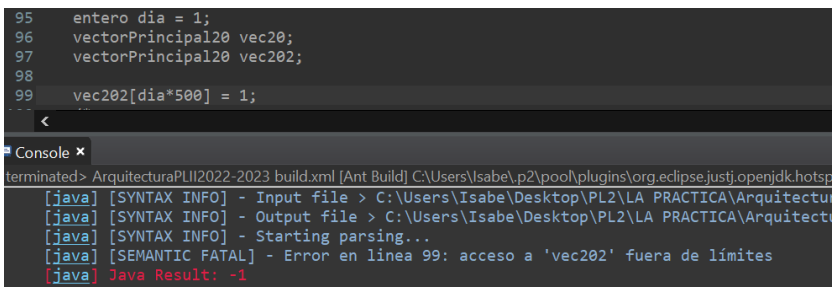
El analizador semántico comprueba que todas las variables y constantes simbólicas referenciadas hayan sido previamente declaradas y el tipo de cada construcción coincide con el previsto en su contexto. Se comprueba que la asignación, comparación y demás operaciones entre los distintos símbolos y estructuras solo se lleven a cabo si estos tienen tipos compatibles.

En esta práctica disponemos de los tipos primitivos entero y vacío. Mientras que las funciones pueden adquirir uno de los dos tipos, las constantes simbólicas, variables, parámetros y elementos que guardan los vectores solo pueden ser de tipo entero. Si al realizar operaciones con estos símbolos los tipos no son compatibles, se producirá un error semántico y se avisará al usuario. Un ejemplo se muestra en la siguiente captura de pantalla, en la que se intenta asignar a una variable de tipo entero un vector entero (y no un elemento):



Concordancia en las referencias

El analizador sintáctico es también el encargado de que las expresiones que referencian una posición de un vector están dentro del rango esperado. En la captura de pantalla se intenta acceder a un vector de tamaño 20 en la posición 500. Se avisa al programador de que el rango no es correcto:



Paso de parámetros y sentencia retorno

En el caso de las funciones, el analizador semántico es el encargado de comprobar que el paso de parámetros es correcto en cuando a tipo y número. Esto se hace comparando la lista de parámetros que se

le ha pasado a la función con la infromación que existe de la función en la tabla se símbolos.
También relativa a las funciones es la comprobación de la sentencia devuelve. Se comprueba que la función esté devolviendo un valor si es de tipo entero o que no devuelva si es de tipo vacío. La manera de realizar esta comprobación es a traves de un booleano "devuelve" en el no terminal ListadoSentencias, que se evalua a True si el listado sentencias contiene una sentencia devuelve y a false en caso contrario.

2. Generación de código intermedio

En esta fase se han insertado en las acciones semánticas las cuádruplas de código intermedio que se utilizarán para realizar la traducción al código final.
Como solo se ha realizado la parte obligatoria de la práctica, se ha implementado la generación de código intermedio de todo el lenguaje a excepción del relativo a las funciones. Es por esto que las funciones no generan código intermedio cuando se utilizan como sentencias y simplemente propagan el valor 1 cuando se utilizan como expresiones.
En lo que se refiere al resto de producciones, que si deben generar código intermedio, en sus acciones semánticas se han diferenciado dos secciones: la primera de análisis semántico y la segunda de generación de código intermedio. El código intermedio generará y propagará elementos temporales cuando sea necesario y generará las cuádruplas que van a subir hacia arriba en el árbol de análisis hasta llegar al axioma. En la imagen a continuación, se pueden apreciar las secciones de análisis semántico y código intermedio bien diferenciadas en la regla que produce una expresión de suma:

```
| expresion:e1 PLUS:op expresion:e2 {:

    Expresion exp = new Expresion();
    ScopeIF scope = scopeManager.getCurrentScope();

    /*----- Análisis semántico -----*/

    if(!(e1.getType() instanceof TypeSimpleEntero) || !(e2.getType() instanceof TypeSimpleEntero)){
        semanticErrorManager.semanticFatalError("Error en linea " + op.getLine() + ": Incompatibilidad de tipos");
    }
    exp.setType(e1.getType());
    exp.setValue(e1.getValue() + e2.getValue());

    /*----- Código intermedio -----*/

    TemporalFactory tf = new TemporalFactory(scope);
    IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
    TemporalIF temp1 = e1.getTemporal();
    TemporalIF temp = tf.create();

    cb.addQuadruples (e1.getIntermediateCode());
    cb.addQuadruples (e2.getIntermediateCode());
    cb.addQuadruple ("ADD", temp, e1.getTemporal(), e2.getTemporal());

    exp.setTemporal(temp);
    exp.setIntermediateCode(cb.create());
    RESULT = exp;
```

Las expresiones son un ejemplo de producciones que generan y propagan temporales. Otras producciones, como las sentencias, generan código intermedio pero no propagan elementos temporales.

3. Generación de código final

La traducción al código final traduce las cuádruplas de código intermedio en instrucciones para la arquitectura ENS2001. Como solo se ha realizado la parte obligatoria, se ha realizado la traducción de todo el lenguaje a excepción del relativo a funciones. La memoria se ha gestionado de manera estática.

La traducción a código final se ha realizado en la clase ExecutionEnvironmentEns2001.java. La función principal de esta clase, translate(QuadrupleIF quadruple), contiene una sentencia switch que genera alternativas en función del operador de la cuádrupla recibida como parámetro. Cada alternativa genera instrucciones específicas de la arquitectura ENS2001 en función de los operandos y el operador y devuelve una cadena con la instrucción de código final que será reconocida por la arquitectura. En la captura de pantalla a continuación, vemos el caso específico de para el operador "EQ" y como se realiza la traducción de la comprobación de igualdad entre dos operandos:

```
case "EQ":

    b.append("CMP " + op1 + ", " + op2 + "\n");
    b.append("BNZ /" + l2 + "\n");        // si resultado no es 0 (op1 != op2) salta a l2
    b.append("MOVE #1, " + res + "\n");
    b.append("BZ /" + l1 + "\n");         // si resultado es 0 (op1 == op2) salta a l1
    b.append(l2 + ": \n");                // l2
    b.append("MOVE #0, " + res + "\n");
    b.append(l1 + ": \n");                // l1

    return b.toString();
```

4. Indicaciones especiales

Por último quería hacer mención a las dificultades encontradas con la gramática proporcionada y mi solución al problema.

La gramática original proporcionada por el equipo docente me estaba dando muchos problemas y conflictos a la hora de abrir ámbitos en las acciones semánticas. Para resolver las ambigüedades que producían estos conflictos hice lo siguiente:

- Dividir la producción del axioma en 2: el no terminal **axiom** crea el ámbito global y llama a **axiom2**, que genera las secciones principales del programa
- Modificar las producciones de la función principal y las funciones para que ambas utilicen el no terminal “cuerpoFuncion”, ya que los cuerpos de ambos tipos de funciones eran iguales.

Estos cambios están también indicados en el parser.cup. Una vez realizados los cambios, no tuve mas problemas de conflictos al crear ámbitos, por lo que decidí no utilizar la nueva versión de la gramática proporcionada por el equipo docente.

Conclusiones

Esta práctica me ha resultado tan entretenida como frustrante.

Tras haber estudiado Teoría de los lenguajes de programación y Procesadores del lenguaje 1, esta ha sido una bonita conclusión a los fundamentos del funcionamiento de los lenguajes de programación. Ha sido posiblemente la práctica en la que más he trabajado de la carrera y sin embargo también la más interesante. Ha sido una lástima no poder haber realizado la parte opcional por falta de tiempo ya que realmente me hubiera gustado, lo cual me lleva al segundo punto de mis impresiones.

El análisis semántico no me ha resultado especialmente difícil, aunque sí muy largo. El código intermedio y final si me han resultado más complicados, ya que era algo que no había utilizado nunca. Esto me ha supuesto tener que reescribir grandes secciones de código, pasar horas delante del ordenador debugeando y tener que sacrificar el realizar la parte opcional por falta de tiempo, ya que también tengo que dedicar tiempo a preparar el examen.

Con todo esto, me quedo con un buen sabor de boca y la sensación de haber derrotado un pequeño dragón. Espero volver a encontrarme en el mundo de los procesadores del lenguaje en el futuro y continuar aprendiendo ya que me ha resultado muy interesante.