PRÁCTICA DE

PROCESADORES DEL LENGUAJE II

Curso 2018 – 2019

Entrega de Junio

APELLIDOS Y NOMBRE: Martínez Montenegro, Alberto

IDENTIFICADOR: amartinez3684

DNI: 36152038V

CENTRO ASOCIADO MATRICULADO: PONTEVEDRA-VIGO

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: PONTEVEDRA

MAIL DE CONTACTO: amartinez3684@alumno.uned.es

TELÉFONO DE CONTACTO: 667827207

GRUPO: A

¿REALIZAS LA PARTE OPCIONAL?: SÍ

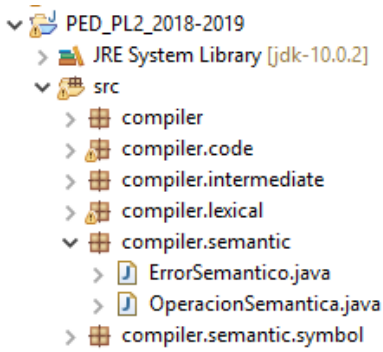
Índice

1. El analizador semántico y la comprobación de tipos

La fase de análisis semántico tiene por objetivo analizar los componentes del árbol de análisis sintáctico que representan el programa con el fin de comprobar que se respetan ciertas reglas semánticas que dan un significado coherente a las construcciones del lenguaje. En el trabajo realizado se han englobado todas estas comprobaciones en dos clases:

- La clase **OperacionSemantica**: Se encarga de todas las operaciones y comprobaciones semánticas.
- La clase **ErrorSemantico**: Se encarga de la gestión de todos los errores semánticos, y los mensajes de información que saldrán por consola.

Ambas están incluidas en el package *compiler.semantic*:



El objetivo de la creación de estás dos clases, es minimizar el código escrito en el 'parser.cup'. De esta forma se evitará repetir muchos fragmentos de código, y lo único que habrá que hacer desde el 'parser.cup' es llamar a los métodos correspondientes.

A continuación, se muestra un fragmento del 'parser.cup' en el que puede observarse lo sencillo que resulta realizar las comprobaciones semánticas necesarias para una sentencia de asignación, apoyándose en estas clases:

```
sentAsign ::=
        TIDENTIFICADOR:id TASSIGN:tassign expresion:exp {:
        //Operaciones y comprobaciones semáticas:
        String idVariable= id.getLexema().toUpperCase();
        OperacionSemantica operacionSemantica= new OperacionSemantica();
        SymbolIF simbolo= operacionSemantica.recuperarSimboloDesdeTablaSimbolos(idVariable, tassign.getLine());
        TypeSimple tipoSimbolo= operacionSemantica.recuperarTipoSimplePorNombre(idVariable, tassign.getLine());
        if(!((simbolo instanceof SymbolVariable) || (simbolo instanceof SymbolParameter))){
            operacionSemantica.getErrorSemantico().lanzarErrorPorSentenciaAsignacionAConstante(tassign.getLine());
        }
        if(!tipoSimbolo.compararNombre(exp.getTipoExpresion())){
            operacionSemantica.getErrorSemantico().lanzarErrorPorSentenciaAsignacionIncorrecta(tassign.getLine());
        }
```

En resumen, para el diseño del analizador semántico se ha seguido esta dinámica basada en esas dos clases. Desde el 'parser.cup' se va llamando a sus métodos, y cada uno de ellos

realiza las operaciones y comprobaciones que le corresponden.

De esta forma la comprobación de tipos es algo sencillo, desde el 'parser.cup' simplemente se invocarán los métodos necesarios para realizar las comprobaciones. Por ejemplo, en el fragmento que se ha expuesto, puede observarse una comprobación de tipos mediante dos métodos que se han descrito.

Para finalizar este apartado, se citan a continuación una serie de **decisiones sobre el analizador semántico**, que se han tomado en base a lo que se ha considerado correcto y a lo que se ha discutido en los foros de la asignatura:

- Las sentencias RETURN dentro de los procedimientos NO están permitidas y generarán un error semántico.
- La declaración de variables (o constantes) cuyo nombre coincida con el nombre de un tipo, o la declaración de un tipo cuyo nombre coincida con el de una variable (o constante) dentro de un mismo ámbito, NO está permitida. Esto es algo que el equipo docente ha indicado en los foros de la asignatura.
- Dentro de una función, toda rama de código debe tener su sentencia RETURN, se generará un error semántico en caso de que haya alguna rama de código que permita que la ejecución de la función termine sin haberse ejecutado una sentencia RETURN. (Por ejemplo: Si sólo existe un RETURN en la función, y éste está dentro de la rama 'ELSE' de una sentencia 'IF', se generará un error semántico.)

1.1 Descripción del manejo de la Tabla de Símbolos y de la Tabla de Tipos

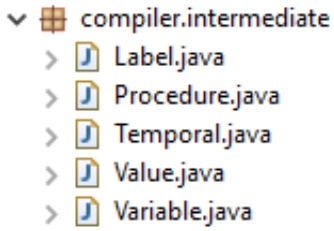
Como se ha descrito en los párrafos anteriores, la clase OperacionSemantica se encarga de realizar todas las operaciones y comprobaciones semánticas, esto incluye también al manejo de la tabla de símbolos y la tabla de tipos.
Como puede observarse en la siguiente captura, la tabla de símbolos y la tabla de tipos son campos privados de la clase, inicializados cuando se construye el objeto:

```
public class OperacionSemantica {
    private ErrorSemantico errorSemantico;
    private ScopeManagerIF scopeManager;
    private ScopeIF scope;
    private SymbolTableIF sTable;
    private TypeTableIF tTable;

    /***
     * Constructor de la clase, toma los valores de los ámbito
     */
    public OperacionSemantica() {
        errorSemantico= new ErrorSemantico();
        scopeManager= CompilerContext.getScopeManager ();
        scope= scopeManager.getCurrentScope();
        sTable= scope.getSymbolTable();
        tTable= scope.getTypeTable();
    }
```

Esto nos permite un control total sobre ambas tablas, de forma que desde 'parser.cup' únicamente haya que llamar a los métodos correspondientes.

2. Generación de código intermedio

Para la generación de código intermedio, se ha partido en principio de los ejemplos propuestos en los ficheros PDF de Javier Vélez Reyes que están disponibles en el curso virtual. A partir de ahí, se ha visto cual era la manera correcta de trabajar con las clases que se nos facilitan relativas al código intermedio:



Y se han ido generando cuádruplas de una forma muy similar a cómo se hace en esos ficheros PDF, pero adaptadas a la gramática de nuestro enunciado.

Durante la traducción a código final, se han tenido que hacer muchas modificaciones en el código intermedio, por haberlo diseñado sin tener en cuenta ciertos aspectos relativos al direccionamiento de memoria o a la invocación de subprogramas.

2.1 Descripción de la estructura utilizada

Para mostrar la estructura del código intermedio generado, se muestran a continuación, todas las posibles cuádruplas que pueden ser construidas en el trabajo realizado, con su descripción:

Operación	Resultado	1er Operando	2º Operando	Descripción
INIT_SUB				Cuádrupla que indica el inicio del programa. Siempre es la primera cuádrupla en la lista.
FIN_SUB				Cuádrupla que indica el fin de un procedimiento que no devuelve ningún valor. (Que no es una función)
RETURN	X			Sentencia RETURN de una función, se devuelve el valor X

RETF	X	Y		Cuádrupla que recoge el valor de una función de su Registro de Activación, y lo almacena en X. El operando Y, es un número entero en el que se indica el número de parámetros, para poder acceder a la posición correcta de Registro de Activación donde se encuentra el valor de retorno de la función.
HALT				Cuádrupla que indica que la ejecución debe ser interrumpida.
MV	X	Y		X:= Y
MVP	X	Y		X:= *Y
MVA	X	Y		X:= &Y
STP	X	Y		*X:= Y
ADD	X	Y	Z	X:= Y + Z
DIV	X	Y	Z	X:= Y / Z
AND	X	Y	Z	X:= Y && Z
EQ	X	Y	Z	X:= (Y == Z)
LS	X	Y	Z	X:= (Y=<Z)
NOT	X	Y		X:= !Y
BR	L			Salto a L
BRF	x	L		Si !X, salto a L
INL	L			Se imprime la etiqueta L
INL	L	X	Y	Cuádrupla que imprime la etiqueta de un procedimiento. Se usa X para operaciones de direccionamiento, y se imprime una etiqueta cuyo texto es la concatenación de la etiqueta L con Y
PARAM	X			*SP:= X ; SP++
CALL	L	Y		Llamada al procedimiento cuya etiqueta es la concatenación de L y Y
PRINTI	X			Se genera instrucción de salida por pantalla para el entero X
PRINTC	L			Se genera instrucción de salida por pantalla para el texto que se encuentra en el destino de la etiqueta L
PRINTLN				Se genera instrucción de salida por pantalla para un salto de línea
CADENA	X	L		Se imprime la etiqueta L, y sus datos asociados, que son la cadena de caracteres X. Este tipo de cuádruplas se posicionan siempre de últimas antes de realizar la traducción.

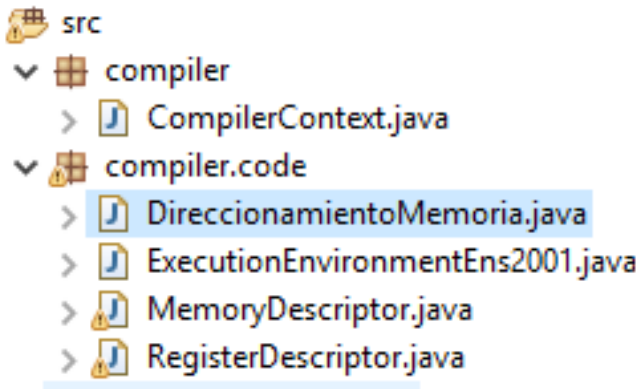
La operación ‘DistintoQue’, se ha implementado combinando las cuádruplas NOT y EQ, por lo que no tiene una cuádrupla exclusiva para su operación.

3. Generación de código final

Para la fase de generación de código final se han partido, al igual que con el código intermedio, de la información y ejemplos encontrados en los ficheros PDF de Javier Vélez Reyes. También ha sido necesario revisar en profundidad el manual de usuario de ENS2001.

Se ha dividido el trabajo de esta fase en dos partes:

- **Direccionamiento de memoria**: Se ha creado una clase específica llamada DireccionamientoMemoria en 'compiler.code' para gestionar las direcciones de memoria de las diferentes variables y temporales.



  Una vez generadas todas las cuádruplas de código intermedio, se llama desde 'parser.cup' al método *direccionar* de la clase DireccionamientoMemoria para dar valor a los campos adress de los SymbolVariable, SymbolParameter y Temporal:

```
axiom:ax
  {:
        DireccionamientoMemoria.direccionar(scopeManager, ax);
        // No modificar esta estructura, aunque se pueden aÃ±adir mÃ¡s acciones semÃ¡nticas
        List intermediateCode = ax.getIntermediateCode();
```

- **Traducción a código final**: Se ha editado el método *translate* de la clase ExecutionEnvironmentEns2001 facilitada por el equipo docente. En ese método se ha añadido una sentencia "if" por cada tipo de cuádrupla. Para traducir de forma especifica cada tipo de cuádrupla a formato ensamblador.

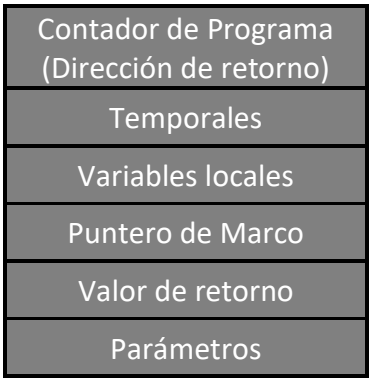
3.1 Descripción de hasta dónde se ha llegado

Se ha implementado el trabajo del enunciado en su totalidad, incluyendo la parte opcional de subprogramas, la recursividad y demás requisitos y restricciones que se han especificado en los foros de la asignatura.

La técnica empleada para la gestión del enlace de acceso de los subprogramas ha sido la **técnica del Display**, para su implementación se ha seguido la descripción que está en los ficheros PDF de Javier Vélez Reyes.

3.2 Descripción del registro de activación implementado

El diseño propuesto para el registro de activación es el siguiente:



Como puede observarse, no se ha incluido el elemento de enlace de acceso, por el siguiente motivo:

Aunque podría haberse incluido un elemento Display[nivel] dentro del registro de activación, no ha sido necesario, ya que puede ser calculado fácilmente en tiempo de traducción. Un subprograma sólo necesitará saber su nivel de anidamiento para poder acceder al vector Display correspondiente, de esta forma, ante referencias no locales se consulta Display[nivel anidamiento] en lugar de acceder a un campo del registro de activación.

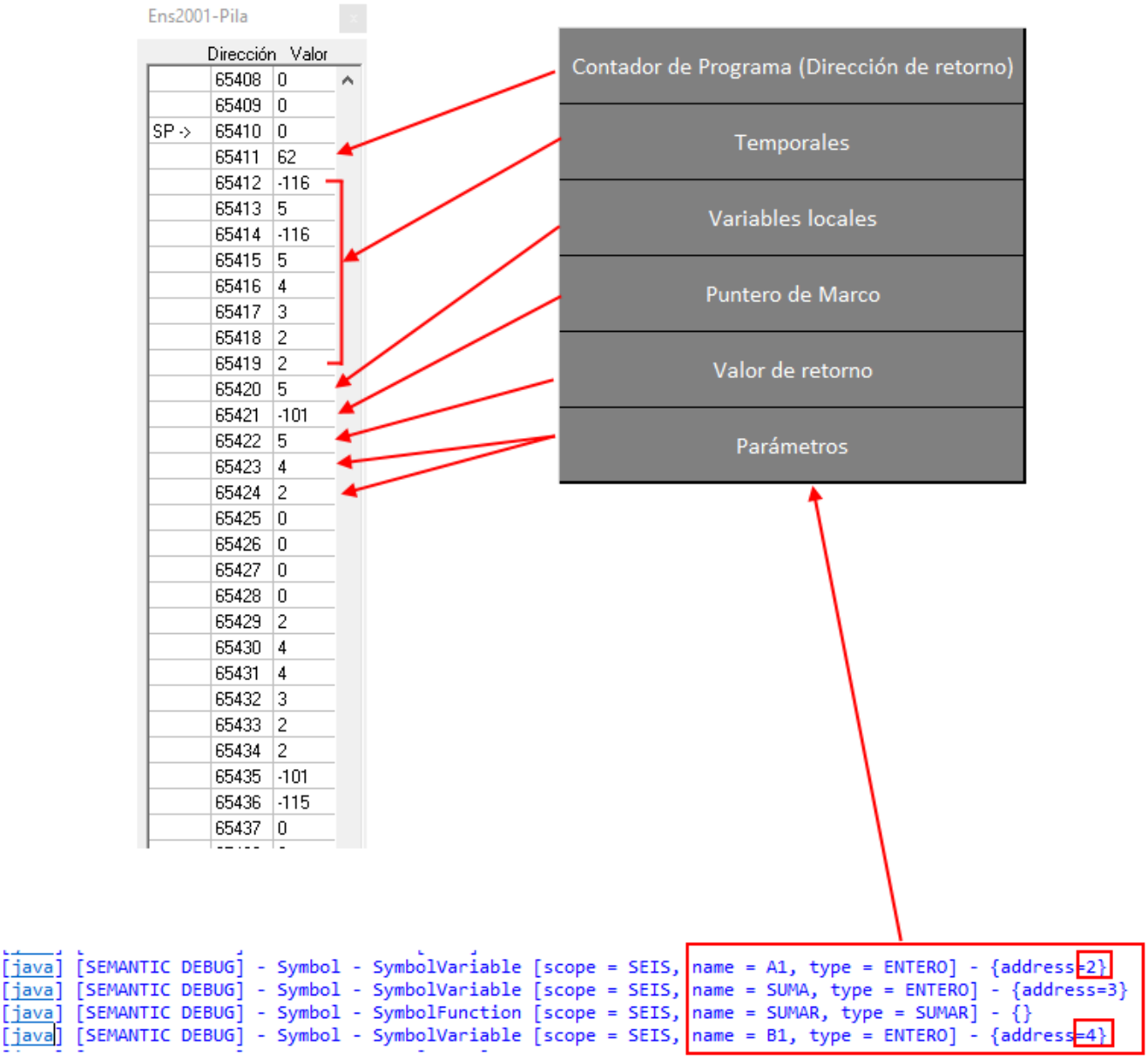
Para ver un ejemplo práctico de como funciona el registro activación en ENS2001, usaré de nuevo el testCase06 facilitado por el equipo docente. Si pausamos la ejecución del programa justo cuando la función sumar ha terminado, y justo antes de que se ejecute “RET” para volver a la función llamante:

```
MODULE seis;

  VAR suma : INTEGER;
        a1:INTEGER;
        b1:INTEGER;

  PROCEDURE Sumar (VAR a: INTEGER; VAR b: INTEGER) : INTEGER;
    VAR s: INTEGER;
    BEGIN
        s := a+b;
        RETURN s;
    END Sumar;

  BEGIN
    a1:=2;
    b1:=3;
    suma := Sumar(a1,b1);
    WRITESTRING("Suma:");
    WRITEINT(suma);

  END seis;
```

Ens2001-Fuente

	Dirección	Instrucción
	143	*NO IMPLEMENTADA*
	145	*NO IMPLEMENTADA*
	147	*NO IMPLEMENTADA*
	148	MOVE .IX,.R1
	150	INC .R1
	152	MOVE #-8[.IX],[.R1]
PC ->	154	RET
	155	*NO IMPLEMENTADA*
	156	*NO IMPLEMENTADA*
	157	*NO IMPLEMENTADA*
	158	*NO IMPLEMENTADA*
	159	*NO IMPLEMENTADA*
	160	NOP
	161	NOP
	162	NOP
	163	NOP
	164	NOP
	165	NOP
	166	NOP
	167	NOP
	168	NOP

Si observamos la memoria justo en ese momento, puede distinguirse claramente el registro de activación creado:



4. Indicaciones especiales

Para comprobar que la práctica funciona correctamente a todos los niveles, se han diseñado dos nuevos test muy exigentes:

- *testCase09.muned* (Módulo Fibonacci)
- *testCase10.muned* (Módulo Vectores).

Entre los dos test, se comprueba el correcto funcionamiento de todas las partes esenciales de este trabajo, entre las que destacan:

- Recursividad directa
- Acceso a variables globales.
- Acceso a variables locales.
- Acceso a variables no locales.
- Acceso a parámetros locales.
- Acceso a parámetros no locales