Memoria
Procesadores de Lenguaje II

Alumno: David Otero Díaz.
DNI: 21001800V.
Correo: dotero64@alumno.uned.es.
Teléfono: 626728560.
Centro Asociado: Islas Baleares.
Tutor: Josep Silva Galiana (jfsilva@valencia.uned.es) - Valencia.

1. El analizador semántico y la comprobación de tipos.

En esta memoria voy a ceñirme a explicar sólo las cuestiones técnicas o concretas que considero de más interés conforme indica la guía; las decisiones que he tomado menos comunes o intuitivas. Siempre se puede acudir al código para comprobar las especificaciones comunes o más normales. Comentar que he utilizado git a modo de control de versiones, por lo que tengo un repositorio privado de Github con la evolución de la realización de la práctica, por si es de interés para el equipo docente; pueden enviarme un usuario de Github y les invito a mi repositorio (para que puedan ver la evolución de la misma).

Apuntar que la sesión de control la he realizado con el tutor Josep Silva Galiana (jfsilva@valencia.uned.es) de Valencia pues es con él con quien he seguido las tutorías al igual que el semestre pasado.

1.1. Descripción del manejo de la Tabla de Símbolos y la Tabla de Tipos (describir cómo se trabaja con ambas tablas y las comprobaciones realizadas).

- Tabla de tipos:
    - Adición a la tabla de los nuevos tipos: añado los tipos compuestos del usuario en la definición de tipos con su nombre y scope. Cada vez que se declara una constante, variable, parámetro o subprograma, compruebo que el tipo ha sido declarado con anterioridad en las tablas de tipos de su ámbito.
    - Comprobación de tipos:
        - Llamada a subprograma: para obtener los parámetros en declaración utilizo un campo estático en forma de lista de parámetro del objeto ListaVariables, este tipo de estrategia es la misma que utilizo en la declaración de variables. Itero sobre la misma para comprobar que los tipos son correctos y están en el orden correcto en comparación con los tipos de parámetros del subprograma declarados (almacenados en el objeto SymbolFunction en su campo lista de SymbolParameter). También compruebo que el número de parámetros sea igual, que en el caso de que sea un símbolo haya sido declarado con anterioridad y que sólo pueda ser las variables o parámetros las que pueden utilizarse para parámetros declarados como VAR.
        - Rangos: compruebo que el tipo de la expresión del rango es entero.
        - Expresiones: solo utilizo la tabla de tipos para obtener el tipo del identificador de la expresión, según sea el tipo creo una PrimitivaAritmetica o una PrimitivaLogica.
        - Sentencia de asignación: compruebo que el tipo izquierdo se corresponde con el derecho (las comprobaciones son nominales) gracias a campo de nombre de tipo en clases Expresion y la información de la tabla de símbolos.

        - Sentencia devolver: cada vez que se llega a una sentencia devolver añado el tipo devuelto en una lista de tipos que guardo en el propio SymbolFunction como campo estático. Después en la declaración de la función compruebo primero que se ha analizado al menos una sentencia devolver gracias también a variable estática y después que todos los tipos devolver se corresponden con el tipo declarado en la función (utilizo un método estático de SymbolFunction para ello).
    - Tabla de símbolos:
        - Adición a la tabla de nuevos símbolos: siempre que se declara una variable, constante, función o parámetro la añado con al menos su scope y su nombre en la tabla de símbolos. No he encontrado necesario añadir el valor a las constantes ya que el analizador semántico según el enunciado no debe de analizar si los accesos a los vectores son correctos a excepción de las constantes literales (números).
        - Comprobación de la existencia del símbolo: siempre que se declara un identificador en el código, se comprueba que exista algún símbolo declarado con anterioridad. Además, en el caso de por ejemplo las expresiones, compruebo que dicho símbolo haya sido inicializado (asignado valor) con anterioridad con un campo booleano.

2. Generación de código intermedio.

2.1. Descripción de la estructura utilizada.

- Cuadruplas utilizadas y significado:
    - ADD: cuádrupla que suma los operandos y los guarda en referencia.
    - BR: cuádrupla que realiza salto incondicional en referencia.
    - BRF: cuádrupla que realiza salto condicional si la comparación anterior ha sido falsa.
    - BRT: cuádrupla que realiza salto condicional si la comparación anterior ha sido verdadera.
    - DATA: cuádrupla para determinar la reserva de espacio en memoria necesaria inicial (no confundir con DATA de código objeto).
    - EQ: cuádrupla que compara la igualdad de los operandos y guarda el resultado en referencia.
    - GR: cuádrupla que compara si el primer operando es mayor al segundo y guarda el resultado en referencia.
    - HALT: cuádrupla para indicar el fin del programa.
    - INC: cuádrupla que incrementa la referencia en uno.
    - INL: cuádrupla para insertar una label.
    - LS: cuádrupla que compara si el primer operando es menor al segundo y guarda el resultado en referencia.
    - MULT: cuádrupla que multiplica el primer operando por el segundo y lo guarda en referencia.
    - MV: cuádrupla que guarda el primer operando en referencia.
    - MVA: cuádrupla que guarda el valor al que apunta el primer operando en referencia.

        - MVM: cuádrupla realizada “ad hoc” para poder implementar el acceso a vectores sin errores, la utilizo para indicar al procesador que hay que recuperar la dirección del puntero del array para acceder al valor que apunta de manera indirecta a través de registro.
        - MVP: cuádrupla para indicar al procesador que debe guardar el puntero a array en referencia.
        - NOT: cuádrupla para invertir el valor de referencia.
        - PARAM: cuádrupla para intentar iniciar la implementación de los registros de activación para subprograma (no realizado).
        - STA: cuádrupla para guardar un valor en la variable referencia.
        - STP: cuádrupla para indicar al procesador que guarde el puntero en referencia.
        - SUB: cuádrupla para restar al primer operando el segundo y guardarlo en referencia.
        - OUT_STRING: cuádrupla para imprimir por pantalla referencia (String).
        - OUT_INT: cuádrupla para imprimir por pantalla referencia (entero).
    - Formateado final del código intermedio: recorro la tabla de símbolos para ir reservando memoria e indicar dónde va a residir dicho símbolo en ejecución (dirección global); esto depende del tamaño del tipo del símbolo que obtengo de la tabla de tipos. Lo mismo hago con los temporales, en este caso su tamaño siempre es de una unidad (una dirección de memoria). Para facilitar el debug del compilador imprimo por pantalla todas las cuadruplas de código intermedio.

3. Generación de código final.

3.1. Descripción de hasta dónde se ha llegado.

Sólo he finalizado la parte obligatoria de la práctica por lo que el compilador se está comportando de manera estática. Me hubiera gustado tener más tiempo para poder finalizar la implementación de los registros de activación en pila pero me ha sido completamente imposible (en parte porque no me he conseguido aclarar del todo). De todos modos, si fuera posible, me gustaría poder ver la solución de los mismos para aprender cómo podrían ser implementados.

- Estructura para la implementación del código final:
    - Siguiendo las recomendaciones de las guías proporcionadas por el equipo docente en el aula virtual; para determinar el formato de cada uno de los operandos de la cuádrupla he utilizado una función privada auxiliar llamada traducir_operando.
    - Para la traducción de la propia operación a código objeto he seguido también la guía docente y he estructurado el código para que cada tipo de operación tenga su propia subclase instanciada por una superclase genérica mediante una función que implementa un switch. Al crear la instancia de la superclase se crea seguidamente la instancia de la subclase correspondiente llamando a su método getCodigoFinal() que devuelve la cuádrupla correctamente traducida.

3.2. Descripción del registro de activación implementado (opcional).

No realizado; no he tenido tiempo suficiente.

4. Indicaciones especiales.

He probado el correcto funcionamiento del compilador pasando con corrección los tests proporcionados por el equipo docente. Además he creado mi propio test donde he añadido todas las variaciones posibles para asegurarme mejor de la corrección del compilador (este test es el que he ido modificando para probar distintos errores semánticos):

```
# programa que escribe una suma
programa tres:
	constantes
		z = 5;
	tipos
		tipoVectorEnteros = vector [0..3] de entero;
		tipoVectorBooleanos = vector [0..3] de booleano;
	variables
		x,w,j,l   : entero;
		g,k       : booleano;
		h,v       : tipoVectorEnteros;
		i,m       : tipoVectorBooleanos;
	subprogramas
		procedimiento ejemplo (var a: entero):
			comienzo
				a = 0;
			fin;
		funcion ejFuncion (var c,b : entero): entero:
			variables
				suma: entero;
			subprogramas
				funcion anidada(VAR a: tipoVectorEnteros): booleano:
					variables
						d:	entero;
						k:	booleano;
					comienzo
						k = falso;
						d = c;
						devolver k;
				fin;
			comienzo
				i[0] = cierto;
				suma = 0;
				v[suma] = 1;
				suma = c + b;
				c = c + 2;
				b = b * 2;
				x = 3;
				si (anidada(v) y i[0] == cierto) entonces:
					devolver ejFuncion(c,b);
				sino:
					devolver suma;
```

```
                                fin si;
                        fin;
comienzo
        j = 1;
        escribir(j);
        h[j] = 2 * 5;
        escribir(h[1]);
        escribir(50 + 50);
        v[1] = 1;
        v[0] = 5;
        v[2] = 5;
        X = h[v[1]];
        ejemplo(x);
        escribir("variable inicializada x: ");
        escribir(x);
        para w en 0 .. x:
                escribir("inicio bucle");
                escribir(w);
                j = j + 1;
                escribir(j);
                para l en 0 .. 2:
                        escribir("inicio bucle interno");
                        escribir(l);
                        si ((h[1] < 15) y (j < 15) y (l == 0)) entonces:
                                si no((h[1] == 2)) entonces:
                                        escribir(0);
                                        escribir();
                                sino:
                                        escribir(5);
                                fin si;
                        sino:
                                escribir("error");
                        fin si;
                fin para;
        fin para;
        escribir("Bucle terminado");
fin.
```