PRÁCTICA DE
PROCESADORES DEL LENGUAJE II
Curso 2023 – 2024
Entrega de Septiembre

APELLIDOS Y NOMBRE: Oreilly del Cerro, Jon

DNI: 72836801X

CENTRO ASOCIADO MATRICULADO: Bergara

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: Barabastro

EMAIL DE CONTACTO: joreilly1@alumno.uned.es

TELÉFONO DE CONTACTO: +34644540989

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

Introducción

Antes de empezar, me gustaría comentar dos peculiaridades sobre la estructura de mi implementación.

1. He movido la mayoría del código de las producciones semánticas a funciones estáticas de factoría, dentro de la clase de cada no terminal. Esto ha facilitado el desarrollo y depuración, gracias al soporte de lenguaje Java que Eclipse no puede darnos en el parser.cup.
2. He añadido bastantes mensajes de depuración, y como consecuencia he necesitado comunicar los lexemas de las producciones hijo a las padre. Para facilitar esto he añadido una propiedad "`lexema`" a la clase `NonTerminal`, que contiene una aproximación del código original

También he añadido varias clases de utilidad, como Consola o Contexto, para facilitar el desarrollo. Estas clases se encuentran en `src/compiler/utils/*.`

Analizador semántico

Al principio de la compilación creamos el ambiente global, con su tabla de símbolos y tabla de tipos. Insertamos en ella los tipos primitivos "`entero`" y "`vacio`". Según continue el proceso de compilación, cuando tengamos una definición de tipo la insertaremos en la tabla de tipos del ambiente actual, y cuando tengamos una definición de contante, variable, función o procedimiento la insertaremos a la tabla de símbolos actual. A su vez, cada vez que entremos en un bloque, crearemos un nuevo ambiente sobre el actual, y cuando salgamos de un bloque lo cerraremos y volveremos a usar el ambiente padre.

El objetivo de estas tablas es ayudarnos a comprobar que el programa cumple las normas semánticas del lenguaje. Siempre que encontremos un fragmento de código donde una regla semántica pueda romperse, recuperaremos la información necesaria de las tablas y comprobaremos si se cumple o no. Si no se cumple lanzaremos un error de compilación semántico.

Por ejemplo, en el fragmento de código "`a = 5;`", las normas del lenguaje nos indican que "`a`" debe estar definida anteriormente, y que ambos lados de la igualdad deben ser del mismo tipo. Comprobamos primero si "`a`" existe en algún ámbito activo usando "`scopeManager.containsSimbol("a")`". Si existe, obtendremos su tipo, y lo compararemos con el de 5, que es de tipo entero primitivo ("`scopeManager.searchType("entero")`"). Si ambas reglas se cumplen continuaremos con la compilación.

Una peculiaridad respecto al uso de las tablas de tipos es que, mientras que dos tablas de símbolos pueden tener un símbolo con el mismo nombre simultáneamente (la anterior ocultada por la más reciente), dos tipos no pueden tener el mismo nombre si desde el ámbito de uno se puede alcanzar el otro.

Otra peculiaridad es que las funciones y procedimientos se añaden a la tabla de símbolos antes de abrir su bloque interno, para permitir recursividad.

Generación de código intermedio

En mi implementación de código intermedio he usado las siguientes instrucciones o cuádruplas:

- `COPY X Y`: Copia el valor de Y en X
- `POINT X Y`: Copia la dirección de memoria de Y en X
- `FIND X Y`: Copia el valor almacenado en la posición de memoria a la que apunta el valor de Y en X
- `STORE X Y`: Copia el valor de Y en la posición de memoria a la que apunta el valor de X
- `RETURN X`: no implementada, seria parte de la parte opcional de la practica
- `ADD X Y Z`: Suma Y e Z, y lo almacena en X
- `MUL X Y Z`: Multiplica Y e Z, y lo almacena en X
- `GR X Y Z`: Si Y es mayor que Z pone X a 1, si no pone X a 0
- `LS X Y Z`: Si Y es menor que Z pone X a 1, si no pone X a 0
- `EQ X Y Z`: Si Y es igual a Z pone X a 1, si no pone X a 0
- `BR L`: Salta a la etiqueta L
- `BRT X L`: Si X es positivo, salta a L
- `BRF X L`: Si X no es positivo, salta a L
- `INL L`: Crea la etiqueta L
- `PRINT_STR L`: Muestra por pantalla la cadena almacenada en la etiqueta L
- `PRINT_INT X`: Muestra por pantalla el valor de X
- `PRINT_LINE`: Muestra un salto de línea por pantalla
- `CADENA L S`: Crea una etiqueta con los datos S
- `SET_STACK_POINTER X`: Mueve el puntero de pila a la posición X

Durante el proceso de compilación, almacenaremos en cada no terminal un conjunto de cuádruplas que reflejen la lógica descrita por el código fuente que los generó (`NonTerminal.intermediateCode`). Los no terminales que encapsulan a otros no terminales copiarán el código intermedio de sus hijos y, a su vez, se lo transmitirán a sus padres. Gracias a esto, el no terminal `Axiom` contendrá todo el código intermedio generado por el resto de las producciones inferiores.

Las cadenas de caracteres que necesitamos para imprimir mensajes por pantalla no serán propagadas como el resto del código. Queremos tener estas etiquetas después del código útil. Para ello almacenaremos las cadenas en una lista (`Contexto.listaCadenas`), y se lo añadiremos al final del código que obtenemos del no terminal `Axiom`.

Como entradas y salidas de las cuádruplas tenemos varias posibilidades:

1- Valores: podemos usar valores numéricos directamente.
2- Variables: estas corresponden a variables definidas en el código fuente, podemos identificarlas mediante su nombre y el ambiente en el que están definidas. Podremos obtener más información (como su tipo) mediante la tabla de símbolos de su ambiente.
3- Temporales: corresponden a almacenes de valores que no se corresponden con una variable. Suelen ser usados para facilitar cálculos intermedios. Hay una tabla de temporales en cada ambiente, que mantiene un registro de ellos.
4- Etiquetas: Son marcas que se pueden usar para saltos en el código, o para almacenar cadenas de caracteres.

A veces nos encontraremos con situaciones donde un no terminal no solo necesita el código intermedio generado por sus hijos no terminales, sino que también necesitará acceso a algunos operandos para usar en sus cuádruplas. En estos casos, el no terminal hijo tendrá una propiedad donde hará accesible este operando a su padre. Este puede ser un temporal donde se ha almacenado una suma, o una etiqueta a la que saltar en un caso específico.

Respecto al código intermedio de las funciones, no he llegado a implementar nada. El código generado en las funciones y procedimientos es descartado, y las llamadas a funciones y procedimientos devuelven siempre "`0`". El código generado en la función principal es añadido al código intermedio como si fuese parte del ámbito global.

Código final

Hay dos partes en la conversión de código intermedio a código final. La primera es la traducción de cuádruplas a instrucciones específicas, y la segunda es la conversión de usar variables y temporales a usar posiciones de memoria.

Respecto a la traducción de instrucciones, la mayoría de cuádruplas son muy similares a sus instrucciones en el procesador objetivo, pero algunas son mas complicadas. Por ejemplo, la cuádruple de igualdad "`EQ`" requiere una instrucción de comparación, dos de salto y otras dos de asignación.

Para convertir variables y temporales a direcciones de memoria podemos asignarles una dirección especifica fija a cada una, o usar registros de activación donde su posición es relativa al puntero IX. El uso de registros de activación nos permite tener funciones recursivas, pero como no he llegado a implementar la parte opcional de la práctica, he usado direcciones sucesivas en la pila.

Antes de traducir el código intermedio al código final, recorremos todos los ambientes y, empezando desde la última posición de memoria, asignaremos a cada variable y temporal posiciones sucesivas en memoria. Dependiendo del tipo de la variable o temporal, la cantidad de memoria reservada para ese valor cambiará. A la hora de traducir las cuádruplas a instrucciones podremos recuperar estas direcciones y usarlas como operandos.

Indicaciones especiales

A la hora de crear los tipos de funciones y procedimientos, he decidido no añadirlos a la tabla de tipos. Los he dejado como tipos anónimos porque si los añadiésemos con un nombre, ese nombre ya no sería usable como nombre de tipo, y eso no forma parte de la especificación del lenguaje.