**Informe de TP Final: Compilador Micro Extendido**

Alumno: Mathias Castets

**FUNCIONAMIENTO**

El compilador desarrollado procesa codigo fuente “Micro” y procesa un codigo intermedio. Su arquitectura se basa en un **analizador sintáctico descendente recursivo** (ASDR) que valida gramática, y al mismo tiempo, invoca a rutinas semánticas que se encargar de comprobar los tipos de datos y generar el codigo.

Este proceso inicia con el **scanner**: se trata de un autómata finito que esta implementado con tabla. Este fue modificado para reconocer nuevos tokens, como por ejemplo REAL y CARÁCTER. A medida que se descomponen las declaraciones (ListaDeclaraciones), se carga en la Tabla de Símbolos (TS) los nombres de las variables que el usuario declara en el codigo, sino tambien con su tipo (TIPOENTERO, TIPOREAL o TIPOCARACTER).

La parte central del análisis semántico son las funciones GenInfijo y Asignar. La primera funcion fue modificada para implementar un sistema de “**promoción” de tipos**: cuando se detecta una operación aritmética en la que interviene al menos un dato de tipo real, el resultado se promueve a tipo real y se genera el codigo de operación “SumarReal”/”RestarReal”. Si nos encontramos ante numeros de tipo entero, se genera “Sumar”/”Restar”. Esto será vital a la hora de comunicar con la Maquina Virtual, ya que no es lo mismo sumar 2 enteros, que 2 numeros reales, los flags empleados varían, asi como la forma en la que se interpreta el resultado.

Al mismo tiempo, Asignar impone una comprobación de tipos, solo podemos asignar datos que correspondan al tipo de variable declarada (entero := entero, carácter := carácter, real := real), o si el tipo del lado derecho puede ser promovido al mismo que el del lado izquierdo (real := entero). Cualquier otra combinación genera un error semántico. Además, en el caso de operaciones lógicas, los resultados booleanos al ser 0 y 1, se establece como tipo entero.

Para el caso de las sentencias de control (si, mientras, repetir), implementamos un sistema de generación de etiquetas y saltos. Desarrollamos la funcion Condicion que evalúa una comparación cuyo resultado booleano (0 o 1) es guardado en una variable temporal (temp). Luego, ChequearCondicion usa ese temporal para generar SaltoSiFalso o SaltoSiVerdadero. De esta forma, la lógica de cada estructura se construye combinando este tipo de saltos. La estructura **mientras** comprueba la condición y salta al final cuando es falsa; los **si** saltan al bloque sino si la condición es falsa y el bloque **entonces** usa un salto incondicional para omitir el sino; y un **repetir** salta de vuelta al inicio si la condición es falsa.

Finalmente, el programa llega a su fin con la instrucción **detiene**.

**DESARROLLO**

Para arrancar, fue necesario identificar cuáles serían los nuevos TOKENS necesarios para llevar a cabo las implementaciones solicitadas (si, mientras, repetir hasta). Una vez hecho eso, modificamos el typedef enum TOKEN, agregando los nuevos TOKENS. Junto a esto definimos un nuevo typedef enum llamado TIPO, que contiene los 3 tipos de datos posibles con los que el compilador trabajara: TIPOENTERO, TIPOREAL, TIPOCARACTER y TIPONULO (para manejo de errores). Y finalmente, tuvimos que modificar la Tabla de Símbolos (TS), agregamos una variable TIPO tipo en el struct RegTS (establece la forma que tendrán los registros en la TS) y agregamos en la RegTS las nuevas palabras reservadas: ENTERO, REAL, CARÁCTER, SI, ENTONCES, SINO, FINSI, MIENTRAS, FINMIENTRAS, REPETIR Y HASTA.

Una vez construida esta nueva base, era momento de extender la tabla para contemplar los nuevos estados posibles. La tabla tuvo que ser ampliada a 30 estados y 18 columnas (se cambio el valor de NUMCOLS y NUMESTADOS). Al agregar columnas, fue necesario modificar la funcion columna para que el scanner pueda interpretarlas correctamente. Estas nuevas columnas contemplan los símbolos necesarios para las implementaciones que realizaremos (como por ejemplo el . para los reales, el < y el > para las comparaciones, etc). Modificamos la funcion estadoFinal agregando al if los nuevos estados NO finales para cuando devuelve 0 (false) para que el do-while funcione correctamente. Y finalmente en el switch(estado) del scanner añadimos los cases necesarios para los nuevos estados finales (17, 20, 21, 23, 24, 26, 27, 29) para que devuelvan los TOKEN correctos (CONSTANTEREAL, CONSTANTECARACTER, IGUAL, MAYORIGUAL, etc). Además, ajustamos la condición col != 16 (espacio) para que coincida con la nueva numeración de columnas y el ungetc funcione correctamente.

Ahora llegamos a una parte clave de nuestra solución, las variables deben ser definidas según su tipo al principio del archivo .m, justo debajo de la palabra inicio y antes de la lista de sentencias. Este decisión fue tomada para mantener simple el desarrollo de las nuevas funciones del compilador, y mantener un orden en la estructura de los programas. Para esto definimos una nueva funcion llamada ListaDeclaraciones, que se encargara de inicializar las variables y establecer su tipo de dato si es necesario. Mientras se encuentre con ProximoToken una palabra reservada de declaración (ENTERO, REAL o CARÁCTER) se llama a la funcion declaración la cual se encargará de procesar la linea de declaración.

**<ListaDeclaraciones> -> <Declaracion> {<Declaracion>}**

Para establecer el tipo de dato, declaramos la función TipoDato que se encarga de identificar el tipo y consumir el TOKEN correspondiente, o llamar a un errorSintactico si fuera el caso. Una vez obtenido el tipo, se invoca a otra funcion que creamos: listaDeIds que recibe un tipo. Esta se encarga de parsear la lista de identificadores que estén separados por comas. Asigna el tipo que recibe y se lo asigna a cada variable que encuentra con la funcion colocar. Dicha funcion fue ligeramente modificada, ahora recibe un parámetro TIPO tipo. En cuanto a su funcionamiento, primero verifica si ya fue declarada antes, si fuera el caso lanzara un error. Si no, guarda el nombre y el tipo, que serán colocados en la TS. Se genera la instrucción con Generar y se mueve el centinela $ al final de la TS.

Con esto se termina lo referido a las declaraciones. Paralelamente a las mismas, fue fundamental adaptar el análisis semántico de las expresiones. Para esto fue necesario realizar cambios en las funciones GenInfijo y Asignar. En cuanto a GenInfijo, ahora compara los tipos de los dos operandos que recibe y genera el codigo intermedio correspondiente a realizar esa operación, devolviendo un nuevo REG\_EXPRESION que representa el resultado. El proceso es el siguiente: se genera una variable temporal y se analizan los tipos de los operandos al llamar a la funcion TIPO ObtenerTipoExp(). Esta funcion comprueba el reg (registro) que recibe, verifica si es una variable, en cuyo caso retorna lo que devuelva la funcion ObtenerTipo, que se invoca pasándole el nombre del registro. Esta última funcion buscará en la tabla el nombre y devolverá su tipo. En otro caso, si se trata de una constante, la funcion simplemente devolverá el tipo que ya esta guardado en el registro(reg->tipo). En su defecto, si no fuera ninguno de los dos casos anteriores, devuelve TIPONULO indicando un error.

Una vez obtenido ambos tipos de los dos operandos, se procede a comprobar la compatibilidad. Por ejemplo, para sumas y restas ambos deben ser enteros o reales, si se intenta operar con un carácter, se reporta un “Error Semantico”, la misma validación se hace con los operadores lógicos (>,<, etc). Una vez comprobado esto, tambien se comprueba el tipo del resultado, en el caso de suma y resta, si al menos uno de los operandos es real el tipo resultado será real, o si ambos son de tipo entero, el resultado tambien lo será. Por otro lado, en los operadores lógicos el resultado siempre se considera entero (0 o 1).

Tras determinar los tipos implicados, se debe determinar el codigo de operación. Si la operación es + o – y la variable tipoResultado almacena TIPOREAL, entonces selecciona los códigos de operación “SumarReal” o “RestarReal”. Si es TIPOENTERO, entonces será simplemente “Sumar” o “Restar”. Y si se trata de un operador relacional selecciona el codigo correspondiente (“Mayor”, “Menor”, etc).

Entonces se procede a llamar a la funcion ChequearTemporal, que es una variación de la funcion Chequear pero para trabajar especialmente con las variables temporales. Recibe el nombre de la variable (Temp&N) y su tipo (que será el tipo correspondiente al resultado), y las agrega a la TS. Finalmente llama a Generar para la generación de la instrucción de la declaración de dicha variable temporal.

Volviendo a GenInfijo, invoca nuevamente a Generar, pero en este caso para generar el Codigo de Operación empleando el codigo de operación seleccionado, los operandos y el nombre de la variable temporal correspondiente. Finalmente, se crea un REG\_EXPRESION reg que será lo que retorne la funcion. En este registro se carga el nombre de la variable temporal, su tipo y su ID.

(FALTA Asignar, tema de etiquetas, saltos, sentencias de control de flujo en general).

**DIFICULTADES ENCONTRADAS**

* Adaptar la tabla de transiciones para reconocer los nuevos estados y símbolos fue un proceso detallado y propenso a errores, que requirió bastante tiempo de prueba.
* Diferenciar correctamente los errores semánticos de los errores sintácticos. Esto requirió modificar varias funciones como Chequear, Asignar y GenInfijo.
* Decidir como implementar los nuevos tipos de datos, tras proponer e intentar varias ideas. Terminamos decidiendo por establecer la sección ListaDeclaraciones al inicio del codigo.
* Gestionar los tipos, implementar su comprobación y establecer las reglas de compatibilidad entre los enteros y reales, incluyendo la promoción de los tipos enteros a reales para permitir hacer algo como real := entero + real.