**INFORME SOBRE EL TRABAJO REALIZADO**

Guillermo Berasategui y Ioar Crespo.

22/12/2019

**Modo de compilación y ejecución**

1. Para compilar, ejecutar *make* dentro de la carpeta programa.

2. A continuación, ir al directorio padre (compilador/) y ejecutar:

*./Programa/a.out ./Programas-de-prueba/programaX.alg*

Donde X puede ser 1, 2, 3 o 4.

Los programas 1,2 y 3 son los ejemplos de el enunciado de la practica y el 4 es un programa sin mucho sentido que simplemente utiliza las instrucciones *para, si y mientras* y sirve para verificar que se genera bien el código en tres direcciones.

3. También hemos creado tres programas incorrectos para verificar que el compilador avisa de ciertos errores, para probarlos hay que ejecutar:

*./Programa/a.out ./Programas-de-prueba/programaIncorrectoX.alg*

Donde X puede ser 1 , 2 o 3

4. En el directorio actual (compilador/) se creara un archivo llamado out.proalg con el código generado para el programa (se han añadido el número de líneas para la comodidad de interpretación de los gotos, pero en ningún caso forman parte del código generado).

5. Para limpiar los archivos generados al compilar, ejecutar *make clean* en el directorio Programa/.

**Desarrollo del analizador léxico**

La primera duda que nos surgió fue la de cómo tratar las entradas según su "case", es decir, cómo gestionar las palabras que tienen el mismo significado pero que están escritas con de forma diferente en cuanto a mayúsculas o minúsculas. Por ejemplo, cómo tratar igual "miEntrAS" y "mientras". La solución que dimos a este problema fue usar la opción -i de Flex que significa "case insensitive" o, lo que es lo mismo, que ignora el "case" de los patrones.

Las palabras reservadas las agrupamos en definiciones regulares diferentes, en vez de agruparlas todas en una misma definición regular.

**Desarrollo del analizador sintáctico**

Respecto a la gramática, el compilador es capaz de reconocerla entera aunque no hemos implementado traducciones para toda la gramatica.

Para solucionar el problema de las expresiones booleanas y aritméticas, nos casamos con la opción similar a la utilizada en C, es decir, que tanto las expresiones booleanas como aritméticas son expresiones a secas y la comprobación de la validez de las expresiones la realizamos en las traducciones.

**Desarrollo de las traducciones**

Modificamos la gramática de la declaración de variables de tal manera que la primera reducción es el tipo de la variable y los sucesivos identificadores ya pueden almacenarse en la tabla de símbolos con su tipo.

La grámatica modificada es la siguiente:

*v\_lista\_id: T\_ID T\_SEPARADOR v\_lista\_id*

*| T\_ID T\_DEF\_TIPO v\_d\_tipo*

de tal manera que en primer lugar se reduce la segunda producción.

Por otra parte cuando se reconoce un literal se crea una variable temporal con su valor y se genera una instrucción de ‘asignación literal’ de tal manera que cuando leemos esa instrucción en la tabla de cuádruplas sabemos que el operando1 es un literal y no un identificador.

La asignación que hemos implementado solo es valida para las expresiones artimeticas, es decir, no se pueden hacer asignaciones de expresiones booleanas.

Para crear las traducciones de las sentencias estructuradas también modificamos la gramática para no tener que utilizar colas donde ir guardando información extra de la traduccion.

En la declaración de variables hemos tenido en cuenta que se pueden repetir los identificadores en las variables de entrada y salida, pero el resto de variables no permitimos que se repitan.

**Resumen de traducciones implementadas**

- Declaración de variables.

- Asignación de variables (aritmeticas) .

- Expresiones aritméticas.

- Expresiones booleanas.

- Flujo básico del programa (sentencias estructuradas): if-then, if-then-else, para, mientras.

**Reparto de trabajo**

Guillermo: 60%.

Ioar: 40%.