Diseño de compiladores

Fecha de entrega: 19-11-2020

Integrante 1: Cabrera Claudio
Integrante 2: Rodríguez Facundo

Contenido

INTRODUCCIÓN

En este trabajo, se continua con la contrucción del compilador. Se desarrollaran las etapas de generación de código y código assembler para la salida y generación del programa ejecutable

Para la generación de código se pueden utilizar varios caminos para realizar la representación intermedia del código, en nuestro caso se utilizo la representación de polaca inversa.

Descripción de la generación de código intermedio

Estructura

Para la generación de código, como ya se mencionó en la introducción, se generó a partir de la representación de la polaca inversa. Para esta representación se generó una clase "CodigoIntermedio" en donde se tiene una hashmap <Integer,String> para almacenar la estructura de la polaca inversa dada las reglas encontradas por el analizador sintáctico. También se tiene un arreglo en el cual se van a almacenar los errores semánticos encontrados en esta etapa. Por último, se hace uso de la estructura Stack para poder apilar y desapilar los elementos de la polaca para la generación correcta del código assembler.

Uso de la notación YACC

El uso de la notación YACC se utilizó para, principalmente quedarnos con el lexema, por ejemplo, si tenemos un PROC a (UINT b) NI =3 {cuerpo del procedimiento}, nos quedamos con $$ para obtener el nombre del procedimiento. Por otro lado, se utilizó también para pasarnos información entre las diferentes reglas, por ejemplo $$ = $1, el valor de la regla, ahora va a ser igual a lo que tiene $1. Entonces de esta manera la vamos pasando las diferentes reglas.

Pseudocodigo para la generación de las bifurcaciones

Para la generación de bifurcaciones se utilizó el algoritmo presentado en clase para nuestra representación de código intermedio. En este caso para la polaca inversa
TERMINA LA EVALUACIÓN DE LA CONDICIÓN. (Generar BF)
1. Crear espacio en blanco

2. Apilar la dirección del paso incompleto.
3. Crear el paso del BF.

FIN DEL BLOQUE THEN. (Se conoce la dirección destino del BF y necesita un BI)

1. Desapilar la dirección del paso incompleto y completar con #_paso_actual + 2(*)
2. Crear espacio en blanco.
3. Apilar la dirección del paso incompleto.
4. Crear el paso del BF.

FIN DEL BLOQUE ELSE. (Se conoce la dirección destino del BI)

1. Desapilar la dirección del paso incompleto.
2. Completar con #_paso_actual + 1.

Errores considerados

Los errores que se consideraron para estas etapas fueron los siguientes.

- **VAR_NO_DECLARADA**: Variable no declarada. Sucede cuando la variable que se está usando no se encuentra previamente declarada.
- **VAR_RE_DECLARADA**: Variable re-declarada. Sucede cuando hay una variable declarada con el mismo nombre más arriba, lo detecta y genera el error.
- **CONSTANTE_NI**: La constante que indica el número de invocaciones debe ser entera
- **ERROR_CONVERSION**: No se puede hacer la conversión a tipo DOUBLE
- **ERROR_INVOCACION_PAR**: El tipo del parámetro no coincide con los utilizados en la invocación en línea
- **ERROR_CANT_PARAM**: La cantidad de parámetros de la invocación no coincide con la declaración del procedimiento en línea ";
- **PROC_NO_DECLARADO**: El procedimiento no existe
- **ERROR_INVOCACIONES_PROC:** Cantidad de invocaciones a procedimiento excedida en línea ";
- **ERROR_PARAM_PROC:** El parámetro no existe en la declaración del procedimiento

Descripción del proceso de generación de código Assembler

Mecanismo utilizado para la generación de código Assembler

Los mecanismos que utilizamos para la generación de código Assembler se basaron en la traducción de la información almacenada en la estructura de la polaca inversa, de manera de generar a partir del mismo, con la ayuda del mecanismo de variables auxiliares para el tipo double y seguimiento de registro para el tipo de variables enteras sin signo. Para el caso de sentencias de tipo UINT, se utilizó siempre los 16 bits menos significativos de los registros EAX, ECX y EDX, junto a las correspondientes variables declaradas de tipo DW necesarias. Para el caso de Double, se utilizó la pila, junto a las sentencias asociadas, del coprocesador 80x87, junto a variables QWORD necesarias.

Mecanismo utilizado para la generación de etiquetas

Para la generación de etiquetas decidimos implementarlo dentro de la generación de código intermedio, por lo que, dentro de nuestra gramática escribimos la etiqueta explícitamente. Esto lo que hace es que cuando vayamos a realizar el código Assembler, leemos desde la polaca inversa y se genera el mecanismo de etiquetas.

Resolución tema 8

A continuación incluimos una descripción de como fue resuelto este tema en cada etapa del compilador

- Tema 8: Control de invocaciones efectuadas – Pasaje de parámetros por referencia

El compilador consta de 3 etapas. La etapa del analizador léxico, la etapa del analizador sintáctico y la generación de código. Hay que pasar por estas 3 etapas para poder generar la salida del programa.

Para este tema a desarrollar, en la primer etapa que es la del analizador léxico, no se tuvo encuenta ningun tipo de funcionalidad para resolver el control de invocaciones efectuadas, dado que en esta etapa se realiza la generación de token de cada variable, palabra reservada, etc. Por lo tanto, se agrego las palabras reservadas REF, PROC, UINT,

NI, {, } , para que reconozca cada token del pasaje de parametro en los procedimientos . En la segunda etapa que es la del analizador sintáctico se genera la gramática de nuestro lenguaje. En esta etapa entonces podemos generar la funcionalidad para que primero, se controle que los procedimientos no puedan tener parametros infinitos (se admiten hasta 3) y lo que también se hace en esta etapa es reconocer la estructura (lista de reglas) del procedimiento, chequeando que este bien escrito y no tenga errores sintácticos. Luego se pasa por la tercer etapa que es la de generación de código, es la etapa donde recibe la lista de regla y se implementa el uso de la polaca inversa en nuestro compilador. Analicemos como se comporta el siguiente programa a continuación.

```
PROC ejemplo(REF UINT a, REF UINT b, REF UINT c) NI = 3_ui {
        a = 4_ui;
        b = a;
        c = 3_ui;
        IF ( c > b ) {
                b = 5_ui;
                a = 3_ui;
        }END_IF
}
```

La representación de código intermedio mediante la polaca inversa lo que va a recibir de la etapa del analizador sintáctico, es la lista de reglas generadas para el programa que estamos corriendo. Una vez recibidas las reglas, las reglas se van generando en la estructura de la polaca con el comportamiento correspondiente.

```
1       |       L:ejemplo@main
2       |       4
3       |       a@main
4       |       =
5       |       a@main
6       |       b@main
7       |       =
8       |       3
9       |       c@main
10      |       =
11      |       c@main
12      |       b@main
13      |       >
14      |       24
15      |       BF
16      |       5
17      |       b@main
18      |       =
19      |       3
20      |       a@main
21      |       =
22      |       25
23      |       BI
24      |       L24
25      |       L25
26      |       :endejemplo@main
```

Se puede ver que por ejemplo que 4 va a ser asignado a la variable a, b se le asigna el valor de a y así sucesivamente, como así también las bifurcaciones que se almacenan cuando hay una condición IF (salto en falso) se guarda la etiqueta BF con su correspondiente dirección para saltar y la etiqueta BI que es para las estructuras WHILE y PROC, que saltan nuevamente al procedimiento de la función. En definitiva lo que se tiene con la polaca es la representación de la siguiente manera:

- Operadores Binarios: op 1 operador op2. Se lee como → op 1 op2 operador
- Operadores Unarios: operador op1. Se lee como → op1 operador.

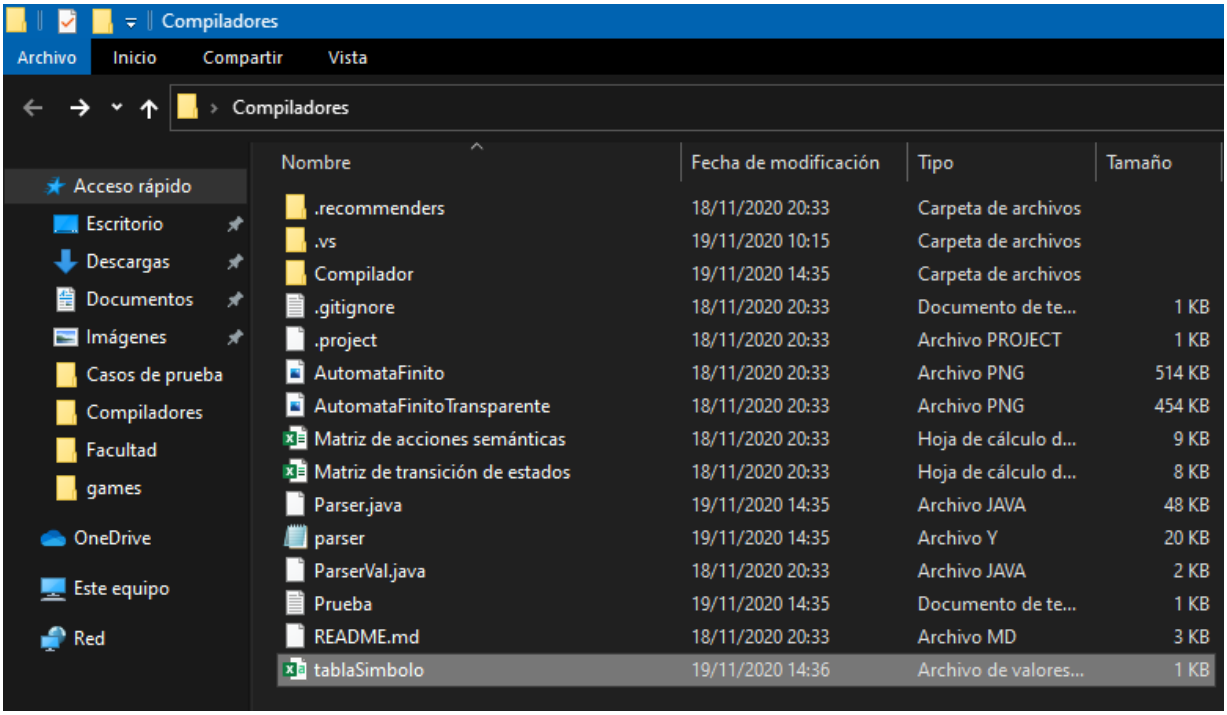
La polaca es una lista (no una pila) que se va leyendo desde el principio hasta el final y Los operandos son referencias a la Tabla de Símbolos.

Luego de generar el código intermedio mediante la polaca inversa, pasamos al código Assembler, el cual luego nos va a generar la salida del programa resultante. Para la generación del código Assembler se recorre la estructura de la polaca, apilando mediante vaya encontrando operandos, hasta que se encuentre un operador. Si hay un operador desapila lo que se encuentre en la pila hasta el momento, dependiendo siempre si es un operador binario (desapila 2 operandos) o si es unario (desapila 1 operando) y luego de esto genera el código Assembler correspondiente a ese código, obteniendo así la salida del programa para generar el ejecutable.

Aspectos relevantes

Se realizo una funcionalidad para que la tabla de símbolo se guarde mediante un archivo CSV el cual tendrá la información de las variables de nuestro programa. Se intento de muchas maneras para que la tabla de símbolo sea legible desde el IDE de eclipse, pero nos pareció mas prolijo generar una salida de archivo, y que el docente si quiere consultar la tabla, pueda hacerlo mediante un Excel y poder tener mayor legibilidad de los tokens que se encuentran adentro de la tabla.

El archivo se guarda dentro de la carpeta raíz y se puede ver de la siguiente manera para encontrarlo.



CORRECCIONES

Dada las observaciones del docente, se corrigió el compilador para cumplir correctamente con su funcionalidad. En el léxico no se reconocía el ultimo token en el caso de que el anterior diera como resultado un token error. Ahora si un token da como resultado error, se descarta todo carácter que se encuentre, hasta un espacio o salto de línea. También en el léxico se corrigió para que el compilador informe bien el número de línea donde se encuentra el error.

En el analizador sintáctico se corrigió los errores de estructuras no contemplados como la falta de END_IF, IF (), IF (a < b, IF a < b, IF a < b), entre otros. Detectamos los doubles negativos, que anteriormente no se estaban reconociendo y detectamos todas las estructuras a nivel sintáctico. Esto quiere decir que si hay un WHILE (true) LOOP { a = 3_ui; }
Ahora el compilador detecta la sentencia ejecutable, la condición y la asignación.

También se modificó la gramática para que siempre que se escriba una sentencia ejecutable, se deba usar las llaves para delimitar los bloques de dichas sentencias.

Nuevas correcciones

Tuvimos un error de comprensión de enunciado con las variables double, el docente asignado nos marcó como debería ser el número double para almacenar y se modificó dicho valor. Por lo cual, se corrigió el error de las variables double, ahora se guardan correctamente el número *10 elevado a la potencia.

También se corrigió el problema cuando se escribe mal una estructura, por ejemplo

```
IF ( a < 2_ui)
END_IF;
```

Esto ya no da errores léxicos, sino sintácticos como debería ser.

Conclusiones

Este trabajo fue una gran ayuda para entender las estructuras y las etapas que lleva la realización de un compilador. Si bien existen varias maneras de realizar la generación de código mediante alternativas para su representación, solo aplicamos una de las alternativas posibles que es la de la representación de código mediante la polaca inversa.

Por otro lado, la implementación a nivel Assembler permite entender con mayor profundidad los mecanismos utilizados por un compilador.

De esta manera, entender estos aspectos del compilador ayuda a limitar las situaciones descritas anteriormente y lograr entonces mejores prácticas de programación.