Trabajo Práctico Especial Diseño de Compiladores I

Número de grupo: 12

Integrantes:

- **Apraiz Thomás: tominola99@gmail.com**
- **Lujan Nicolas: Nico.L.2014@hotmail.com**

Profesor asignado:

- **José Fernández León**

Temas asignados: 3 7 (9) 10* 13 18 20 (21) 22* 24 27

Introducción

En esta última etapa del trabajo práctico especial de la materia “Diseño de compiladores I” se desarrolló la parte de generación de código intermedio y traducción a assembler de un compilador basado en características especiales de un lenguaje dado como consigna por la cátedra de la materia. El código intermedio generado en este caso es Tercetos, es decir, un código intermedio usado por compiladores para poder optimizar y hacer el código fuente más portable entre distintas arquitecturas. Por otro lado, el código assembler generado es para un Pentium de 32 bits. En el presente informe se detalla tanto la consigna pedida por la cátedra como la solución planteada.

Temas particulares asignados

- Tema 3. Enteros (16 bits): Constantes enteras con valores entre –2 15 y 215 – 1. Se debe incorporar a la lista de palabras reservadas la palabra i16
- Tema 7. Punto Flotante de 32 bits: Números reales con signo y parte exponencial. El exponente comienza con la letra F (mayúscula) y el signo es opcional. La ausencia de signo, implica un exponente positivo. La parte exponencial puede estar ausente. Puede estar ausente la parte entera, o la parte decimal, pero no ambas. El „.“ es obligatorio. Ejemplos válidos: 1. .6 -1.2 3.F–5 2.F+34 2.5F-1 15. 0. 1.2F10. Considerar el rango 1.17549435F-38 < x < 3.40282347F+38 -3.40282347F+38 < x < -1.17549435F-38 0.0
- Tema 10: Cláusulas de compilación Incorporar, a las sentencias declarativas, la posibilidad de definir constantes.
- Tema 13. Do until con continue: Incorporar a la lista de palabras reservadas las palabras do, until y continue.
- Tema 18. Continúe con etiquetado.
- Tema 20. Sentencia de control como expresión.
- Tema 22. Conversiones Implícitas.
- Tema 24. Comentarios de 1 línea: Comentarios que comienzan con “<<” y terminan con el fin de línea.
- Tema 27. Cadenas multilínea: Cadenas de caracteres que comiencen y terminen con “ ‘ ” . Estas cadenas pueden ocupar más de una línea, y en dicho caso, al final de cada línea, excepto la última debe aparecer una barra “ / ”. (En la Tabla de símbolos se guardará la cadena sin las barras, y sin los saltos de línea). Ejemplo: ‘¡Hola / mundo!’

Descripción de la Generación del Código Intermedio

Estructura de almacenamiento código intermedio

Para la generación de tercetos se hizo uso de dos clases: Terceto y TercetoManager. Terceto es una clase que almacena los atributos que debería tener un terceto como por ejemplo: operando, operador 1 y operador2. Dichos atributos son de tipo string ya que pueden ser una referencia a la Tabla de Símbolos (constantes o identificadores) como también a otro tercetos (en este caso el operador se encierra entre [ ]). Por otro lado TercetoManager es la clase que almacena, elimina y modifica los tercetos. Esta clase está implementada como un ArrayList de Terceto.

De tal manera que si tenemos la siguiente asignación:

- a =: c + 3;

El ArrayList de TercetoManager nos quedaría de la siguiente manera:

1. (+, c, 3)
2. (=:, a, [1])

Por otro lado en TercetoManager se usó una pila “stackTercetos” la cuál es utilizada para apilar tercetos incompletos o inicios de sentencias de control iterativas. Cuando se genera una bifurcación en el caso de las sentencias de control pueden ser de dos tipos: BI (bifurcación incondicional) y BF (Bifurcación por falso). Cuando se generan bifurcaciones no se conoce el terceto al cuál se debe saltar en caso de que la condición sea falsa o verdadera por lo tanto se debe usar dicha pila de tercetos para apilar el que está incompleto para que en posteriores sentencias se complete el terceto con la dirección de salto.
Por ejemplo:

```
if ( a < b) then{
        a =: 5;
}else{
        b =:5;
}end_if;
```

Tercetos generados:

1. (<, a, b)
2. (BF, [1], [5]) Terceto incompleto hasta que se conozca la dirección de salto
3. (=:, a, 5)
4. (BI, [6],) Terceto incompleto hasta que se conozca la dirección de salto
5. (=:, b, 5)
6. FUERA_DEL_IF

Uso notación posicional Yacc

Por otro lado, en cuanto a la utilización de la notación posicional de Yacc se utilizó en múltiples ocasiones y por diversos motivos.

Uno de los usos principales de la notación posicional fue para concatenar y chequear el ámbito de ciertas variables con el fin de corroborar el alcance de las variables utilizadas (por medio del mecanismo Name Mangling).

```
401  header_funcion: FUN ID '(' lista_parametros ')' ':' tipo {String ambitoAux = $2.sval;
402                                                            Ambito.concatenarAmbito(ambitoAux);}
403  ;
404
405  factor: ID {comprobarAmbito($1.sval);}
406  ;
```

También se usó la notación posicional para chequear tipos entre dos expresiones aritméticas para comprobar si es posible la operación que se desea realizar entre los dos operandos.

```
416  condicion:'(' expresion_aritmetica operador expresion_aritmetica ')' {verificarTipos($2.sval, $4.sval, $3.sval);}
417  ;
418  operador: '<' {$$.sval = "<";}
419  ;
```

Por otro lado la notación posicional de yacc se utilizó para generar los tercetos correspondientes de cada sentencia ejecutable, por ejemplo en la regla:

```
condicion:'(' expresion_aritmetica operador expresion_aritmetica ')' {TercetoManager.crear_terceto($3.sval, $2.sval, $4.sval);}
```

Donde $3 hace referencia al operador y $2 y $4 hacen referencia a los operandos de la condición.

Descripción pseudocódigo algoritmo de generación de bifurcaciones de sentencias

El algoritmo o técnica utilizada para la generación de las bifurcaciones en las sentencias de control fue Backpatching.

Generación de código sentencia IF:

Al detectar el fin de la condición en la gramática:

-Crear un tercetos incompleto para la bifurcación por falso.

-Apilar el número del terceto incompleto creado anteriormente.

Al detectar el fin del cuerpo del then en la gramática:

-Desapilar y completar el tercetos incompleto de la bifurcación por falso.

-Crear un terceto con una bifurcación incondicional y apilar dicho terceto incompleto

Al detectar el fin de la sentencia if:

-Desapilar y completar el terceto incompleto de la bifurcación incondicional.

Generación de código para sentencia do until:

Al detectar el inicio de una sentencia do until:

-Apilar el terceto de inicio de la sentencia do

Al detectar la condición:

-Crear terceto incompleto para bifurcar por falso y apilar.

Al detectar el fin de la sentencia do until:

-Desapilar y completar terceto incompleto de la bifurcación por falso.

-Desapilar y crear un terceto con una bifurcación incondicional al inicio de la sentencia do.

Generación de código para sentencia when:

Al detectar el fin de la condición de una sentencia when:

-Crear terceto incompleto para bifurcar por falso y apilar.

Al detectar el fin de la sentencia when:

-Desapilar y completar el terceto incompleto de la condición.

Nuevos errores considerados

Los nuevos errores que fueron considerados en esta etapas son los errores de tipo semántico como por ejemplo:

- Variables no declaradas.
- Variables redeclaradas.
- Funciones no declaradas.
- Funciones redeclaradas.
- Errores de compatibilidad de tipos.
- Chequeos de tipo y número de parámetros en invocaciones a funciones.

Para el ejemplo ejecutado en el archivo pruebaEtapa2_6.txt, con el siguiente código:

```
1  program {
2
3      i16 i;
4
5      f =: 1;
6
7      i16 i;
8
9      i =: a();
10
11     fun aaa() : i16{
12         return (1);
13     }
14
15     fun aaa() : i16{
16         return (2);
17     }
18
19 }
```

Podemos observar en la siguiente imagen, el compilador no genera errores léxicos ni sintácticos, pero si errores semánticos.

```
No hay errores lexicos.
No hay errores sintacticos.
['f' no esta al alcance, Variable 'i' redeclarada, 'a' no esta al alcance, La funcion 'a' no esta declarada, Funcion 'aaa' redeclarada]
```

Para el caso del archivo pruebaEtapa2_7, con el siguiente código:

```
program {

    f32 a;
    i16 b, c, d;

    b =: a;
    d =: 1;
    c =: 2;

    fun suma(i16 a, i16 b) : i16 {
        return (a+b);
    }


    b =: suma();

    b =: suma(d,c);

    b =: suma(a,a);

}
```

En este caso, el compilador no genera errores léxicos ni sintácticos, pero si errores semánticos.

```
No hay errores lexicos.
No hay errores sintacticos.
[No se puede realizar la operacion =: entre los tipos i16 y f32, No coinciden la cantidad de parametros de 'suma', Los tipos de los parametros no coinciden]
```

Por otro lado también se agregaron ciertos errores sintácticos que no había sido considerados en etapas anteriores, como por ejemplo:

```
227   impresion: OUT '(' CADENA {erroresSintacticos.add("Falta un )");}
228   ;
```

Estos errores son almacenados en un arraylist de errores sintácticos, semánticos y léxicos con una breve descripción del error y su respectiva línea en la que aparece.

Aspectos considerados

Para realizar los distintos chequeos semánticos de variables, funciones, llamados a funciones, etc. Se utilizó la técnica NameMangling vista en la materia para identificar las variables y funciones relacionadas con el ámbito al que pertenecen. Para implementar esta técnica se utilizó una clase Ámbito que almacena el ámbito actual y cada vez que entra a una función se le concatena el nombre de dicha función (y cada vez que finaliza la función se la remueve del ámbito) de tal manera que si tenemos la siguiente función.:

```
pepe{
    fun aa(): i16{
        fun cc(): i16{
            i16 z;
            z =: 5;
            return (z);
        }

        i16 x;
        x =: 4;
        return (x);
    }

    fun bb(): f32{
        f32 y;
        y =: 3;
        return (y);
    }
}
```

Las variables quedan almacenadas de la siguiente manera en la tabla de símbolos:
x:main:aa
y:main:bb
z:main:aa:cc

(probar archivo “pruebaEtapa3”)

Además se incorporó a la tabla de símbolos información del tipo para los Identificadores a partir de las sentencias declarativas, para las constantes se registró el tipo durante el Análisis Léxico y el uso de los identificador en el programa (variable, constante, nombre de función, nombre de parámetro, etc.). Otros atributos que se agregaron a la tabla de símbolos fueron el ámbito, la cantidad y tipos de los parámetros (para el caso donde el identificador es un nombre de función), etc. También se agregó un atributo tercetoSalto para los identificadores que son etiquetas.
Otro tema a considerar fue el de conversiones implícitas donde el compilador incorporará una conversión del tipo i16 a f32, si se intenta efectuar una operación entre dos operandos de dichos tipos y, por otro lado, si el lado izquierdo es del tipo entero en una asignación, y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos. En cambio, si el lado izquierdo es del tipo de punto flotante y el lado derecho es de tipo entero el compilador deberá incorporar la conversión que corresponda al lado derecho de la asignación. Como veremos en el siguiente ejemplo:

```
program{

    i16 a,b,d,f,z;
    f32 x,e;

    x =: a + b * z + d / e + f;

}
```

Donde los tercetos generados son los siguientes, haciendo sus respectivas conversiones de tipos:

```
Tercetos:
0. (*, b, z)
1. (+, a, [0])
2. (tof32, d, _)
3. (/, [2], e)
4. (tof32, [1], _)
5. (+, [4], [3])
6. (tof32, f, _)
7. (+, [5], [6])
8. (=:, x, [7])
```

Descripción del proceso de Generación de Código Assembler

Temas particulares asignados

Controles en Tiempo de Ejecución: Incorporar los chequeos en tiempo de ejecución que correspondan al tema particular asignado al grupo. Cada grupo deberá efectuar los chequeos indicados para situaciones de error que pueden producirse en tiempo de ejecución. El código generado por el compilador deberá, cada vez que se produzca la situación de error correspondiente, emitir un mensaje de error, y finalizar la ejecución.

a) **División por cero para datos enteros y de punto flotante**
El código Assembler deberá chequear que el divisor sea diferente de cero antes de efectuar una división. Este chequeo deberá efectuarse para los dos tipos de datos asignados al grupo.

b) **Overflow en sumas de enteros**
El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos entero asignado al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

h) **Recursión mutua en invocaciones de funciones**
El código Assembler deberá controlar que si la función A llama a la función B, ésta no podrá invocar a A.

Generacion de codigo Assembler

Para la generación de código, creamos una clase encargada de generar el código assembler para un Pentium de 32 bits. En la misma se recorre la lista de tercetos y se va procesando uno a uno. El método utilizado fue el de variables auxiliares, agregandolas a la tabla de símbolos para luego verificar su tipo.
La clase procesa los operadores y a partir de ellos genera el código assembler correspondiente a cada terceto.
Para la conversión de tipos implícita, creamos una clase encargada de verificar el tipo correspondiente a un terceto. Esto se hace mediante el operador y los operandos del terceto, correspondiendose a una suma, resta, división, multiplicación.
Para el control de división por cero en el caso de los enteros y flotantes se movió el segundo operando a un registro y se chequea si es cero (similar en el caso de datos flotantes). Por ejemplos si tenemos el siguiente programa:

```
1 program{
2
3     i16 w, x, y, z;
4
5     w =: x / y + z;
6
7 }
```

Primero movemos y a AX y lo comparamos con cero. En caso de que sea igual se invocará a una función que avisa que el segundo operando es cero, de lo contrario, salta al label para continuar con la ejecución del programa.
Por otro lado para chequear el overflow de la suma usamos la instrucción JNC que salta si no hubo acarreo (es decir overflow) y por lo tanto sigue con la ejecución normal del programa. Por otro lado si hubo overflow imprime un mensaje avisando del error como podemos ver en la siguiente imagen.

```
START:
FNINIT
MOV AX, y
CMP AX, 00h
JNE Label@aux0
invoke MessageBox, NULL, addr DIVISIONPORCERO, addr DIVISIONPORCERO, MB_OK
invoke ExitProcess, 0
Label@aux0:
MOV AX, x
MOV DX, x
DIV y
MOV @aux0, AX
MOV AX, @aux0
ADD AX, z
JNC Label@aux1
invoke MessageBox, NULL, addr OVERFLOW, addr OVERFLOW, MB_OK
invoke ExitProcess, 0
Label@aux1:
MOV @aux1, AX
MOV AX, @aux1
MOV w, AX
END START
```

Modificaciones a las etapas anteriores

Con respecto a modificaciones de etapas anteriores se realizaron varias, principalmente con respecto a las reglas de la gramática ya que, por ejemplo si teníamos una regla de la siguiente manera en etapa anterior:

sentencia_if: IF condicion THEN lista_sentencias ELSE lista_sentencias END_IF;

La tuvimos que modificar de tal manera de detectar el inicio de la sentencia if, el fin de la condición y el fin de toda la sentencia para poder generar los tercetos correspondientes.
De tal manera que nos quede:

seleccion: inicio_if condicion_if cuerpo_if END_IF;
inicio_if: IF
condicion_if: condicion
cuerpo_if: THEN lista_sentencias ELSE lista_sentencias

Por otro lado se agregó código en las etapas anteriores como pueden ser más atributos en la tabla de símbolos, distintos métodos como modificarTipoParametros(String key, String tipo1, String tipo2) en la clase TablaSimbolos para setear los tipos de los parámetros de la funciones.
En la gramática agregamos funciones para verificar el número y tipo de parámetros, tipos de las variables y si es una etiqueta.

Modo de resolución temas: 10, 18 y 20

Tema 10: Cláusulas de compilación

Para resolver este ejercicio lo que se planteó fué que cada vez que se declara una constante const en el programa principal se debe registrar que se trata de una constante, el valor y el tipo que corresponda. De esta manera cuando se realiza una asignación en la gramática se chequea que no se esté realizando esta asignación a una constante por medio del atributo uso del identificador.

Por otro lado para la resolución de la sentencia when <condición> then <bloque_de_sentencias> se debe hacer uso de la pila “stackTercetos” de la clase TercetoManager. Cuando detectamos el fin de la condición del bloque when creamos un terceto con bifurcación por falso y apilamos dicho terceto incompleto en la pila para que posteriormente pueda ser completado con el salto fuera del bloque when. Al finalizar el bloque when hacemos un pop de la pila y obtenemos el índice del terceto incompleto creado anteriormente, de esta manera, podemos completarlo con la direccion de salto.

(probar archivo pruebaEtapa2_2).

Tema 18: Continue con etiquetado

Al detectar una etiqueta la guardamos en la tabla de símbolos junto con la dirección al terceto de salto de la etiqueta. Cuando dentro de la sentencia do until o de un do until anidado nos encontramos con un continue seguido de una etiqueta buscamos en la tabla de símbolos si existe y está al alcance y creamos un terceto de bifurcación incondicional al terceto de salto indicado por la etiqueta de la tabla de símbolos. Por ejemplo dado el siguiente programa:

```
1 program {
2     i16 a,b,i;
3     b =: 227;
4     a =: 0;
5
6     outer: do{
7
8         i =: i + 1;
9         do{
10            a =: a + 2;
11
12            b =: b + 1;
13
14            continue: outer;
15        }until (a < b);
16    }until (i < 100);
17 }
```

Se crean los siguientes tercetos:

```
0. (=:, b, 227)
1. (=:, a, 0)
2. (Label2, _, _)
3. (+, i, 1)
4. (=:, i, [3])
5. (Label5, _, _)
6. (+, a, 2)
7. (=:, a, [6])
8. (+, b, 1)
9. (=:, b, [8])
10. (BI, [2], _)
11. (<, a, b)
12. (BF, [11], [14])
13. (BI, [5], _)
14. (Label14, _, _)
15. (<, i, 100)
16. (BF, [15], [18])
17. (BI, [2], _)
18. (Label18, _, _)
```

(Aclaración: si bien en la imágen se ven el nombre de los identificadores en el TercetoManager dichos identificadores son referencias a la tabla de símbolos donde se le concatena su ámbito).

Como podemos observar en la imágen el terceto 10 es el salto incondicional creado por la sentencia “continue: outer”.

(probar archivo pruebaEtapa2_3)

Tema 20: Sentencia de control como expresión

Para realizar este tipo de sentencias hicimos uso de las reglas de la gramática y utilizamos un idAux de tipo String para almacenar el lexema asociado a cada variable de una asignación a sentencias de control como expresión. Al detectar una sentencia break de la siguiente forma en la gramática:

```
BREAK CTE ';' {TercetoManager.add_break_cte(idAux, $2.sval);}
```

creamos un terceto de asignación y otro incompleto con una bifurcación incondicional y lo pusheamos en un pila aparte llamada “TercetoAsignacion”. Al detectar el fin de la condición de una sentencia de control como expresión creamos un terceto que bifurque por falso con la condición anterior y con un salto dos tercetos posteriores (ya que el terceto posterior es la bifurcación incondicional al inicio del do until).
Finalmente cuando nos encontramos con la siguiente regla en la gramática:

```
ELSE CTE {TercetoManager.add_else_cte(idAux, $2.sval);}
```

Creamos un terceto de asignación con el id auxiliar y el valor de la constante.
Por ejemplo dado el siguiente programa:

```
1 program {
2     i16 a,i,number,end;
3 
4     a =: do {
5               if (i = number)then{
6                  break 1;
7               }else{
8                  i =: i + 1;
9               }end_if;
10          }until (i> end) else 3 ;
11 }
```

Se crean los siguientes tercetos:

```
0. (Label0:, _, _)
1. (=, i, number)
2. (BF, [1], [6])
3. (=:, a, 1)
4. (BI, [14], _)
5. (BI, [9], _)
6. (Label6, _, _)
7. (+, i, 1)
8. (=:, i, [7])
9. (Label9, _, _)
10. (>, i, end)
11. (BF, [10], [13])
12. (BI, [0], _)
13. (=:, a, 3)
14. (Label14:, _, _)
```

Donde el terceto 3 y el terceto 13 son los tercetos correspondientes a la asignación de la sentencia de control (Cabe aclarar que el terceto 5 se crea debido a la sentencia if que hace un salto fuera del if).

(probar archivo pruebaEtapa2_4).

Lista detallada de consignas que no funcionan

Se testearon todos los puntos de cada uno de los trabajos practicos y llegamos a la conclusion de que los puntos que no funcionan son los siguientes:

- Funciones con parámetros no compila correctamente en Assembler.
- Llamada de funciones dentro de otra función no compila correctamente en Assembler.
- Recursión mutua en invocaciones de funciones por el punto anterior.

Conclusiones

Como conclusión aprendimos tanto el funcionamiento interno de un compilador como a poder desarrollar uno a lo largo de estos cuatro trabajos prácticos. Aprendimos las distintas etapas que tiene un compilador y como entre ellas se conectan para llevar a cabo la compilación de un programa escrito en un lenguaje determinado.
Entendimos el funcionamiento del analizador léxico encargado de detectar tokens en el programa a compilar mediante la implementación de un autómata finito y enviarlos al analizador sintáctico.
Un analizador sintáctico recibe dichos tokens y chequea, según las reglas de una gramática formal, el cumplimiento de las reglas gramaticales del lenguaje, es decir es el encargado de corroborar la sintaxis del código fuente.
En la etapa de generación de código intermedio se van creando los tercetos correspondientes conforme se ejecutan las reglas del parser para poder optimizar y hacer el código fuente más portable entre distintas arquitecturas.
Al final se procesan los tercetos generados y se realiza la traducción del código intermedio a assembler el cuál finalmente podrá ser ejecutado.
También aprendimos a usar otros tipos de herramientas como yacc para construir el parser a partir de la gramática y masm32 para ejecutar el código assembler.
Finalmente, y para cerrar, como grupo pudimos comprender el funcionamiento de un compilador con una arquitectura monolítica(usada por yacc) donde un detalle en el código puede ocasionar grandes fallas.