DISEÑO DE COMPILADORES I

Grupo 7

Integrantes:

Ceschini Tomás tomiceschini99@gmail.com
Fratti Joaquin Ignacio jfiati@alumnos.exa.unicen.edu.ar
Pozzi Matias Nicolas pozzi.matias.nicolas@gmail.com

Introducción

En el marco del curso de Compiladores, abordamos dos trabajos prácticos fundamentales. En el TP3, nos enfocamos en la generación de código intermedio para sentencias ejecutables, incorporando información semántica a la Tabla de Símbolos y realizando chequeos específicos. El TP4 se centró en la generación de código Assembler a partir del código intermedio del TP3, empleando variables auxiliares y asegurando su ensamblaje y ejecución sin errores. Este informe detalla de manera sucinta el proceso de resolución de ambos trabajos, enfocando nuestra documentación en las decisiones para la implementación y aplicación de los conceptos aprendidos durante la cursada.

CONSIDERACIONES

- Se decidió que no se puede declarar clases dentro de otra (Anidar).
- Se decidió que la sentencia return solo este al final de una función.
- Para saber si el assembler entra en un if, en un else, o en la parte del do colocar PRINT # entre en X#,
- Si no genera el assembler, puede ser que se tenga un error en la gramática y no se agregue a erroresSintacticos, ya que el token error está a nivel sentencia.
- Las declaraciones de funciones no llevan coma al final, solo si es declaración sin cuerpo, lo mismo las clases.
  (…… VOID f1 () {INT a,} …..)
- Los casos de prueba mencionados y mostrados en las correcciones de la reentrega están agregados en la carpeta pruebasSemantico con la palabra Assembler al comienzo.

Analizador Semántico

TEMAS GENERALES

La forma en que se resolvió el chequeo de declaraciones de variables y por consecuente el ámbito es creando una clase ámbito donde se lleva una lista de string donde se guardan las ID de las funciones y clases en las que se va entrando y se va eliminado cuando se sale de estas, luego al encontrarse con una declaración de variable/método se añade esta lista a su ID para comprobar si no se encuentra ya declarada.
Luego cuando estas variables o funciones son usadas se chequea si efectivamente existen y están al alcance.

TEMAS PARTICULARES

TEMA 18: Herencia por Composición y Uso con nombre

La forma de resolución de la herencia es comprobar si existe la clase de la que se herenda y el uso con nombre se realizó agregando en el token a qué clase pertenece la variable y/o función declarada dentro de una clase y buscando en la tabla de símbolos si la clase hija o padre tiene declarado la función/variable correspondiente.

TEMA 20: Declaración de métodos distribuida

La forma de resolución es primero que todo asegurarse mediante comprobación de ámbito si efectivamente la clase existe, luego con la adhesión de 2 atributos en el token de la tabla de símbolos (String clase y boolean implementado) se chequea si efectivamente este método pertenece la clase a la que se le quiere implementar y con el boolean se chequea si está o no implementado el método.

TEMA 21: Forward declaration.

La forma de resolución es primero que todo asegurarse que existe la declaración de la clase sin cuerpo chequeando el ámbito y luego mediante el uso de un boolean en la clase token (boolean cuerpo) se chequea si este efectivamente tiene o no cuerpo.

TEMA 23:Sobreescritura de métodos.

Con el uso del atributo herencia y clase dentro de token, se buscan atributos en la clase padre que sean iguales a los de la clase hija para no permitirlo.

TEMA 28: El compilador debe chequear e informar, para cada variable declarada, si no fue usada del lado derecho de una asignación en ningún ámbito.

La forma de resolución es añadir un booleano a la clase token (boolean usadoDerecha) en donde se indica si fue o no usado a la derecha, luego antes de la ejecución del generador de código se chequea este booleano sobre la tabla de símbolos para informar si efectivamente no se usaron variables.

TEMA 31: Sin Conversiones, prohibición de operación entre tipos distintos.

Para la prohibición de las operaciones entre los operandos de distintos tipos, se procedió a la creación de una clase llamada TablaTipos la cual contiene cuatro matrices, asignadas para las operaciones de, suma y resta, multiplicaciones y divisiones, comparadores y asignaciones, las cuales fueron cargadas controlando la posición donde caerá el índice para la realización de las comparaciones.

ERRORES CONSIDERADOS

Se consideraron todos los errores de alcance, tanto para uso de variables como para los diferentes temas particulares:

- Clase de la que se hereda fuera de alcance.
- Atributo de clase fuera de alcance.
- Método de clase fuera de alcance.
- Declaración de una instancia de una clase fuera de alcance.
- Implementación de un método de una clase fuera de alcance.
- Implementación de un método de una clase donde el método está fuera de alcance.
- Llamado de función fuera de alcance.
- Pasaje de parámetro fuera de alcance.

Luego diferentes errores que tienen que ver con los temas particulares pero no de alcance:

- Declaración de instancia de clase que no fue declarada (forward declaration).
- Implementación de un método de una clase que ya fue implementado.
- Pasaje de parámetro de distinto tipo.
- Pasaje de parámetro a función sin parámetro.
- Llamado a función sin parámetro a función con parámetro.
- Variables nunca usadas a la derecha de una asignación.
- Asignaciones, comparaciones, operaciones entre variables/constantes de diferentes tipos.
- Anidamientos en clase superiores a 1.

USO DE NOTACIÓN POSICIONAL

La notación posicional se usó durante todo el desarrollo del tp3.
Para la realización del ámbito esta se usó para poder obtener y agregar la ID de cada función y clase a esta.
Para la resolución de los temas particulares se usó para poder utilizar de manera correcta los diferentes atributos de token y hacer los chequeos necesarios de cada tema.
Para la adhesión de los símbolos correspondientes a la polaca, la notación se usó para capturar y añadir estos

CASOS DE PRUEBA

Llamados a función con parámetros no coincidentes.

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 6 : los tipos no coinciden
```

```
        DOUBLE a,
        VOID f1(INT b){
                b = 1_i,
        }
        f1(a),
}
```

Llamado a función

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo a:main:f1 nunca fue


TABLA DE SIMBOLOS
Lexema: f1:main  Token: 257  Ambito: :
```

```
{
        VOID f1(){
                INT a,
                a = 1_i,
        }
        f1(),

}
```

Llamado a función erróneo

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 6 : 'f1' no esta al alcance
Error : el simbolo a:main:f2 nunca fue usado
```

```
Archivo  Edición  Formato  Ver  Ayuda
{
        VOID f2(){
                INT a,
                a = 1_i,
        }
        f1(),
```

Redeclaración de función

```
No hay errores lexicos


No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 6 : Redeclaracion de Nombre de funcion
Error : el simbolo b:main:f2 nunca fue usado
Error : el simbolo a:main:f2 nunca fue usado
```

Archivo Edición Formato Ver Ay

```
{
        VOID f2(){
                INT a,
                a = 1_i,
        }
        VOID f2(){
                INT b,
        }

}
```

Redeclaración correcta, en diferente ámbito

```
No hay errores lexicos


No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo b:main:f
Error : el simbolo a:main:f


TABLA DE SIMBOLOS
```

Archivo Edición Formato Ver Ayuda

```
{
        VOID f2(){
                INT a,
                a = 1_i,
        }
        VOID f1(){
                VOID f2(){
                        INT b,
                }
        }
```

Redeclaración de variable

```
ERRORES SEMANTICOS
Error en la linea 3 : Redeclaracion de Variable
Error : el simbolo a:main nunca fue usado
```

```
{
        INT a,
        INT a,
```

Uso de atributo de clase

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo a:main:ca nunca fue usado
```

```
{
        CLASS ca {
                INT a,
        }
        ca instancia,
        instancia.a = 3_i,
}
```

Atributo no existente

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 6 : 'b' no esta al alcance
Error : el simbolo a:main:ca nunca fue usado
```

```
{
        CLASS ca {
                INT a,
        }
        ca instancia,
        instancia.b = 3_i,
}
```

Uso correcto de método de clase

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo b:main:ca:f1 nunca fue usado
```

```
{
        CLASS ca {
                VOID f1(){
                        INT b,
                }
        }
        ca instancia,
        instancia.f1(),
}
```

Uso incorrecto de método de clase

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 8 : 'f2' no esta al alcance
Error : el simbolo b:main:ca:f1 nunca fue usado
```

```
{
        CLASS ca {
                VOID f1(){
                        INT b,
                }
        }
        ca instancia,
        instancia.f2(),
```

Forward declaration

```
No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo b:main:ca nunca fue usado
```

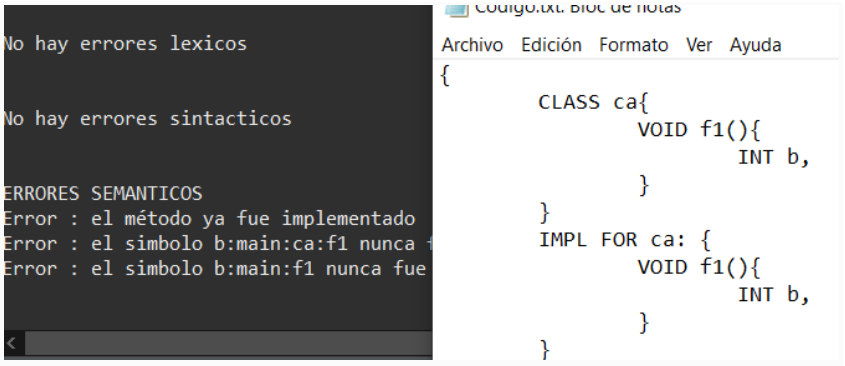
```
{
        CLASS ca,
        CLASS ca{
                INT b,
        }
}
```

Declaración de una instancia de una clase sin cuerpo

```
ERRORES SEMANTICOS
Error en la linea 3 : 'ca' no esta al alcance
```

```
{
        CLASS ca,
        ca instancia,
}
```

Error método ya implementado

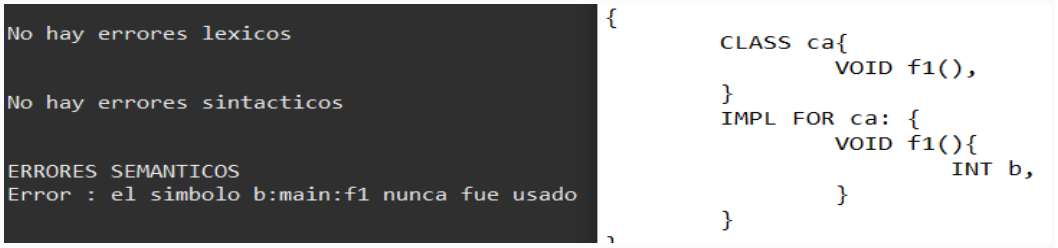


```
No hay errores lexicos


No hay errores sintacticos


ERRORES SEMANTICOS
Error : el método ya fue implementado
Error : el simbolo b:main:ca:f1 nunca
Error : el simbolo b:main:f1 nunca fue
```

Archivo Edición Formato Ver Ayuda

```
{
        CLASS ca{
                VOID f1(){
                        INT b,
                }
        }
        IMPL FOR ca: {
                VOID f1(){
                        INT b,
                }
        }
```

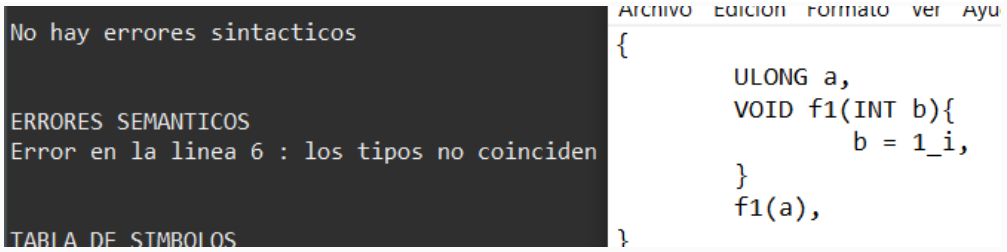
Implementación de método de clase



```
No hay errores lexicos


No hay errores sintacticos


ERRORES SEMANTICOS
Error : el simbolo b:main:f1 nunca fue usado
```

```
{
        CLASS ca{
                VOID f1(),
        }
        IMPL FOR ca: {
                VOID f1(){
                        INT b,
                }
        }
}
```

Pasaje de parámetros incorrecto

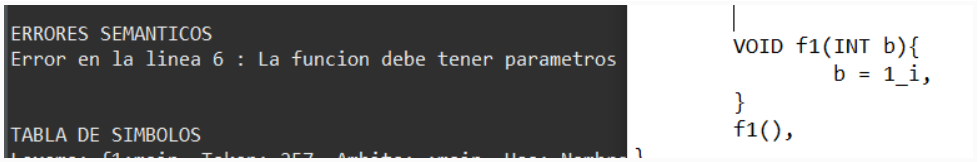


```
No hay errores sintacticos


ERRORES SEMANTICOS
Error en la linea 6 : los tipos no coinciden


TABLA DE SIMBOLOS
```

Archivo Edición Formato Ver Ayu

```
{
        ULONG a,
        VOID f1(INT b){
                b = 1_i,
        }
        f1(a),
}
```

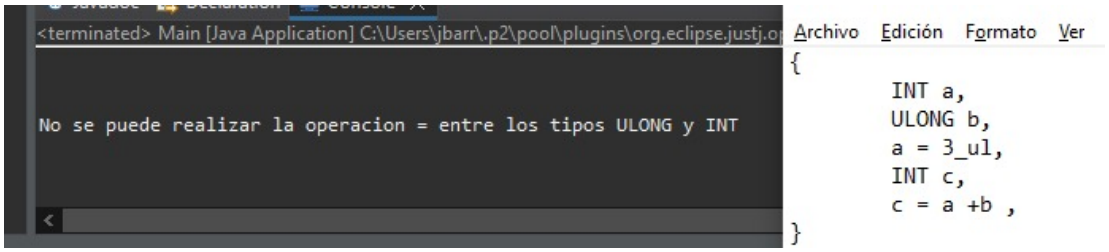
Llamado a función sin parámetro



```
ERRORES SEMANTICOS
Error en la linea 6 : La funcion debe tener parametros


TABLA DE SIMBOLOS
```

```
        VOID f1(INT b){
                b = 1_i,
        }
        f1(),
}
```

Tipos incompatibles



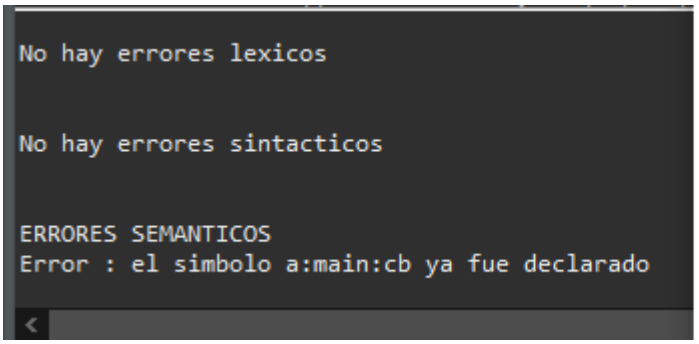
<terminated> Main [Java Application] C:\Users\jbarr\.p2\pool\plugins\org.eclipse.justj.o

```
No se puede realizar la operacion = entre los tipos ULONG y INT
```

Archivo Edición Formato Ver

```
{
        INT a,
        ULONG b,
        a = 3_ul,
        INT c,
        c = a +b ,
}
```

Redeclaración de atributo




```
CLASS ca {
        INT a,
}

CLASS cb {

        INT a,
        ca,
}
```

Descripción de la Generación de Código intermedio

Para la generación de código intermedio contamos con una clase completa la cual llamamos Polaca. En la misma contamos con varias estructuras listas para el control de la correcta aplicación y construcción de la Polaca.
Para agregar a la estructura de la polaca, contamos con un chequeo en donde se pregunta si el ámbito actual es igual al main, para poder diferenciar entre las distintas secciones de la polaca. Esto nos permite que a la hora de su construcción poder controlar en donde van a estar las bifurcaciones.

BIFURCACIONES EN SENTENCIAS DE CONTROL

Cuando se debe hacer una bifurcación en una sentencia de control, lo que se hace en la polaca es poner la condición y luego colocar un símbolo especial que indica una bifurcación. Este símbolo especial en realidad puede ser 2 cosas: BI, el cual indica que hay una bifurcación incondicional; y BF, el cual indica que hay una bifurcación por falso. Antes de este símbolo, se guarda la dirección de salto.

Generación Código Assembler

Descripción del proceso:

MECANISMO UTILIZADO PARA LA GENERACIÓN DEL CÓDIGO ASSEMBLER

Para la generación de código Assembler se optó por el mecanismo de variables auxiliares en el cual aprovechando la generación del código intermedio, es decir, la polaca y junto a dos estructuras, un StringBuilder para la generación del código y una estructura LIFO para la pila de tokens. Logramos establecer el mecanismo para la generación del código Assembler.

MECANISMO UTILIZADO PARA EFECTUAR CADA UNA DE LAS OPERACIONES ARITMÉTICAS Y GENERACIÓN DE LAS ETIQUETAS DESTINO DE LAS BIFURCACIONES

Para controlar las distintas operaciones aritméticas las cuales se irán leyendo a través de la polaca, se optó por la utilización de la estructura Switch. La misma nos permite que a medida que se va leyendo los valores de la Polaca generada en la generación de código intermedio se generen decisiones de control de flujo en base al valor de una expresión. Con esto logramos comparar el valor de la operación aritmética para cada caso y a su vez las bifurcaciones correspondientes para las comparaciones. En el caso de las etiquetas, lo que decidimos hacer fue crearlas y agregarlas dentro de la polaca con la palabra “Label” , para que después cuando se vaya generando el código assembler, si encuentra dicha palabra invocará a la función correspondiente para generar la etiqueta.

Conclusión

A lo largo de estos 4 trabajos prácticos comprendimos cómo funciona internamente compilador y la conexión/comunicación entre las etapas del análisis léxico, análisis sintáctico, generación del código intermedio y la generación del código assembler o ensamblador, para un procesador Pentium de 32 bits.

En el **análisis léxico** se van generando los tokens utilizando chequeos desde una matriz que contiene funciones para la generación de tokens.

En la etapa de **análisis sintáctico** se reciben secuencialmente los tokens desde la etapa anterior y se realizan chequeos en el código corroborando el cumplimiento de las reglas gramaticales, para verificar la sintaxis de nuestro lenguaje.

En la etapa de **generación del código** intermedio se agregan los diferentes tokens a la polaca inversa conforme se agregan desde el parser y las distintas etiquetas para las bifurcaciones de las sentencias de control. También se chequea la semántica del mismo.

Por último se realiza la generación a **código assembler,** el cual finalmente podrá ser ejecutado.

Como conclusión grupal podemos decir que se requiere mucha coordinación y agregado de diferentes métodos para conectar las etapas, y que para hacer un buen compilador se requieren muchos chequeos y eficiencia a la hora de generar código, ya que pequeños detalles pueden ocasionar grandes fallas.