Diseño de Compiladores I – Cursada 2023
Trabajo Práctico Nro. 3

La entrega se hará en forma conjunta con el trabajo práctico Nro. 4 (**16/11/2023**)
OBJETIVO: Se deben incorporar al compilador, las siguientes funcionalidades:

Generación de código intermedio

- Generación de código intermedio para las sentencias ejecutables. Es requisito para la aprobación del trabajo, que el código intermedio sea almacenado en una estructura dinámica. La representación puede ser, según la notación asignada a cada grupo:

Árbol Sintáctico	*Polaca Inversa*	*Tercetos*
1	7	2
3	13	4
5	15	6
9	17	8
12	18	10
14	21	11
19	25	16
20	26	22
23	28	24
30		27

Se deberá generar código intermedio para todas las sentencias ejecutables, incluyendo:

- Asignaciones, Selecciones, Sentencias de control asignadas al grupo, Sentencias PRINT, invocaciones a funciones y métodos de clase y sentencias RETURN**.**

Incorporación de información Semántica y chequeos

- ***Incorporación de información a la Tabla de Símbolos:***
  - ***Tipo:***
    - Para los Identificadores, se deberá registrar el tipo, a partir de las sentencias declarativas.
    - Para las constantes, se deberá registrar el tipo durante el Análisis Léxico.
  - ***Uso:***
    - Incorporar un atributo Uso en la Tabla de Símbolos, indicando el uso del identificador en el programa (variable, nombre de clase, nombre de método, nombre de parámetro, etc.).
  - ***Otros Atributos:***
    - Se podrán Incorporar atributos adicionales a las entradas de la Tabla de Símbolos, de acuerdo a los temas particulares asignados

Chequeos Semánticos:

En esta etapa, se deberán efectuar los siguientes chequeos semánticos, informando errores cuando corresponda:

- Se deberán detectar, informando como error:
  - Variables no declaradas (según reglas de alcance del lenguaje).
  - Variables redeclaradas (según reglas de alcance del lenguaje).
  - Funciones no declaradas (según reglas de alcance del lenguaje).
  - Funciones re declaradas (según reglas de alcance del lenguaje).
  - Clases no declaradas (según reglas de alcance del lenguaje).
  - Clases redeclaradas (según reglas de alcance del lenguaje).
  - Métodos no declarados (según reglas de alcance del lenguaje).
  - Métodos redeclarados (según reglas de alcance del lenguaje).
  - Toda otra situación que no cumpla con las siguientes reglas:

Reglas de alcance:

- Cada variable, función, clase, método será visible dentro del ámbito en el que fue declarada/o y por los ámbitos contenidos en el ámbito de la declaración.

- Cada variable, función, clase, método será visible a partir de su declaración, con la restricción indicada en el ítem anterior.
- Se permiten variables con el mismo nombre, siempre que sean declaradas en diferentes ámbitos.
- Se permiten funciones con el mismo nombre, siempre que sean declarados en diferentes ámbitos.
- Se permiten clases con el mismo nombre, siempre que sean declaradas en diferentes ámbitos.
- No se permiten variables, funciones, clases con el mismo nombre dentro de un mismo ámbito.
- Otras reglas que correspondan a temas particulares.

➢ Chequeo de compatibilidad de tipos:
- Temas 29 y 30: Sólo se permitirán operaciones con operandos de distinto tipo, si se efectúa la conversión que corresponda (implícita o explícita según asignación al grupo).
- Tema 31: Sólo se podrá efectuar una operación (asignación, expresión aritmética, comparación, etc.) entre operandos del mismo tipo. Otro caso debe ser informado como error.

Nota:
- Para Tercetos y Árbol Sintáctico, este chequeo se debe efectuar durante la generación de código intermedio
- Para Polaca Inversa, el chequeo de compatibilidad de tipos y la incorporación de conversiones implícitas (si corresponde) se debe efectuar en la última etapa (TP 4)

➢ Chequeos de tipo y número de parámetros en invocaciones a funciones
- El número de los parámetros reales en una invocación a una función (uno o ninguno), debe coincidir con el número de los parámetros declarados para la función.
- Para el tipo del parámetro, cuando haya un parámetro, se aplicará el chequeo de compatibilidad, indicado en el ítem anterior.

➢ Otros chequeos relacionados con los temas particulares asignados.

Asociación de cada variable con el ámbito al que pertenece:

Para identificar variables, funciones, clases, etc. con el ámbito al que pertenecen, se utilizará "name mangling". Es decir, el nombre de una variable llevará, a continuación de su nombre original, la identificación del ámbito al que pertenece.

Ejemplos:

Ámbitos

```
{
      ...
      // Declaraciones en el ámbito global
      LONG a,    // la variable a se llamará a:main
                 // main identifica el ámbito global
      ...
      VOID aa(LONG x)                          // Ámbito aa
      {
              INT a,                           // la variable a se llamará a:main:aa
              ...
              VOID aaa(INT w)                  // Ámbito aaa
            {
                    ...
                    LONG x,                   // la variable x se llamará
                                              // x:main:aa:aaa
                    ...
            }                                 // Fin Ámbito aaa
      }                                       // Fin Ámbito aa
      ...

      VOID bb(INT z)                          // Ámbito bb
      {
             ...
              INT a,                          // la variable a se llamará a:main:bb
              ...
      }                                       // Fin Ámbito bb
```

```
        ...
        VOID cc(INT y)                          // Ámbito cc
        {
                INT a,                          // la variable a se llamará a:main:cc
                ...
        }                                       // Fin Ámbito cc
        ...
}                                              // Fin Ámbito main
```

TEMAS PARTICULARES

Funciones (Para todos los temas)

Registro de información en la Tabla de Símbolos

En esta etapa, se deberá registrar:

- Información referida a la función:
  - Si tiene parámetro, se deberá registrar el tipo del mismo.
- Otros atributos que el compilador requiera para permitir esta funcionalidad en el lenguaje.

Código intermedio

- Se deberá generar código para cada función declarada. El código de cada función se podrá generar en una única estructura, junto con el programa principal, o bien, utilizando una estructura independiente para el programa principal y para cada función.
- Se deberá generar código para la invocación a cada función, chequeando el tipo del parámetro en caso que la función tenga parámetro.
- El pasaje de parámetros será por copia-valor. Por lo tanto, se deberá generar el código para el pasaje de parámetros correspondiente previo a cada invocación.

Temas 9 a 12 - Operadores

- **Tema 9**: **++**

  Cada vez que se encuentre el operador ++ luego de una variable, luego del uso de dicha variable, se deberá generar el código correspondiente a la actualización de su valor actual.

  Por ejemplo:

  ```
  a = b++ * 7_i,          // luego del uso de la variable b con su valor actual, se deberá generar el
                          // código para actualizar su valor (b = b + 1)
  ```

- **Tema 10**: **--**

  Cada vez que se encuentre el operador -- luego de una variable, luego del uso de dicha variable, se deberá generar el código correspondiente a la actualización de su valor actual.

  Por ejemplo:

  ```
  a = b-- * 7_i,          // luego del uso de la variable b con su valor actual, se deberá generar el
                          // código para actualizar su valor (b = b - 1)
  ```

- **Tema 11**: **+=**

  Ante una sentencia de asignación que utilice el operador +=, se deberá generar código para asignar a la variable su valor actual + el resultado del lado derecho de la asignación:

  Por ejemplo:

```
a += 10_i,          // El código generado debe efectuar la asignación a = a + 10_i,
```

- **Tema 12**: **-=**

  Ante una sentencia de asignación que utilice el operador -=, se deberá generar código para asignar a la variable su valor actual - el resultado del lado derecho de la asignación:

  Por ejemplo:

```
a -= 10_i,          // El código generado debe efectuar la asignación a = a - 10_i,
```

Temas 13 a 16: Sentencias de Control

- **Tema 13**: ***WHILE DO***

  WHILE (<condicion>) **DO** <bloque_de_sentencias_ejecutables> ,

  El bloque de sentencias ejecutables se ejecutará mientras la condición sea verdadera.

- **Tema 14**: ***DO UNTIL***

```
DO <bloque_de_sentencias_ejecutables> UNTIL ( <condicion> ),
```

  El bloque de sentencias ejecutables se ejecutará hasta que la condición sea verdadera. La evaluación de la condición se efectuará luego de la ejecución del bloque.

- **Tema 15**: ***DO WHILE***

```
DO <bloque_de_sentencias_ejecutables> WHILE ( <condicion> ),
```

  El bloque de sentencias ejecutables se ejecutará mientras la condición sea verdadera. La evaluación de la condición se efectuará luego de la ejecución del bloque.

- **Tema 16**: ***FOR***

  FOR i **IN RANGE** (m ; n ; j) < bloque_de_sentencias_ejecutables > ,

  La variable de control i, comenzará la iteración con el valor m.
  La constante n se utilizará para controlar el final de la ejecución del bloque.
  En cada ciclo, el incremento de la variable de control será igual a j.

  Ejemplos:

```
FOR i IN RANGE (0;10;2)…    // La ejecución del bloque se iniciará con un valor inicial 0 para la
                                // variable i, y se ejecutará mientras su valor sea menor que 10.
                                // La actualización de i en cada ciclo será de 2.
FOR j IN RANGE (15;5;-1) …  // La ejecución del bloque se iniciará con un valor inicial 15 para la
                                // variable j, y se ejecutará mientras su valor sea mayor que 5.
                                // La actualización de j en cada ciclo será de -1.
```

  Se deberán efectuar los siguientes chequeos de tipo:
  - i debe ser una variable de tipo entero (1-2-3-4-5-6).
  - m, n y j deben ser constantes de tipo entero (1-2-3-4-5-6).

  Nota:
  - Para Tercetos y Árbol Sintáctico, los chequeos de tipos se deben efectuar durante la generación de código intermedio (TP03)
  - Para Polaca Inversa, los chequeos de tipos se deben efectuar en la última etapa (TP04)

Temas 17 a 24: Objetos

- **Tema 17**: ***Herencia por Composición - Uso anónimo***

Ejemplos de declaración de clases:

```
CLASS ca {
   INT a,       // declaración de atributo
   VOID m() {   // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT c,     // declaración de atributo
   VOID n() {   // declaración de método
      …
   },
   ca,          // nombre de clase
}
…
```

Semántica de la herencia: La clase cb hereda de la clase ca.

Ejemplos de declaración de objetos para las clases ca y cb:

```
ca a1; a2,
cb b1; b2; b3,
```

Ejemplos de referencias a métodos y atributos de los objetos declarados:

```
a1.a = 3_i,  // atributo declarado en ca
b1.b = 1.2,  // atributo declarado en cb
b2.a = 5_i,  // atributo heredado de ca
b1.m(),      // método heredado de ca
b1.n(),      // método declarado en cb
a2.m(),      // método declarado en ca
```

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- **Tema 18**: ***Herencia por Composición - Uso con nombre***

Ejemplos de declaración de clases:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {   // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT a,     // declaración de atributo
   VOID n() {   // declaración de método
      …
   },
   ca,          // nombre de clase
}
…
```

Semántica de la herencia: La clase cb hereda de la clase ca.

Ejemplos de declaración de objetos para las clases ca y cb:

```
ca a1; a2,
cb b1; b2; b3,
```

Ejemplos de referencias a métodos y atributos de los objetos declarados:

```
a1.a = 3_i,     // atributo declarado en ca
b1.ca.a = 3_i, // atributo heredado de ca
b1.a = 2.3,     // atributo declarado en cb
b1.b = 1.2,     // atributo declarado en cb
b1.ca.c = 1_i, // atributo heredado de ca
b1.ca.m(),      // método heredado de ca
b1.n(),         // método declarado en cb
a2.m(),         // método declarado en ca
```

Se indica explícitamente si el atributo o método corresponden a la clase heredada por composición.

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- **Tema 19**: ***Declaración de métodos concentrada con control de niveles de herencia.***

Los métodos se declaran dentro de la declaración de la clase a la que pertenecen.
Además, el compilador debe chequear que no haya más de 3 niveles de herencia. Por ejemplo, si la clase A hereda de la clase B, y B hereda de la clase C, no se debe permitir que la clase C herede de ninguna otra clase.
El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

Ejemplo:

```
CLASS cx{
   …              // declaración de atributos y métodos de la clase cx
}

CLASS cc{
   …              // declaración de atributos y métodos de la clase cc
   },
   cx,            // Se debe informar como error por exceder el límite de niveles de herencia
}

CLASS cb{
   …              // declaración de atributos y métodos de la clase cb
   },
   cc,           // la clase cb hereda de la clase cc
}


CLASS ca {
   …              // declaración de atributos y métodos de la clase ca
   },
   cb,           // la clase ca hereda de la clase cb
}
…
```

- **Tema 20**: ***Declaración de métodos distribuida***

Los métodos pueden declararse dentro de la declaración de la clase a la que pertenecen, o mediante una cláusula IMPL. Ejemplo de implementación de métodos mediante cláusula IMPL:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {  // declaración de método
     …
   },
   VOID p(),    // prototipo de método
}

ca x; y; z,

x.p();          // Debe ser informado como error

IMPL FOR ca: {
   VOID p() {  // implementación del método p de la clase ca
      ….
    },
   …
}
```

El compilador debe efectuar los chequeos necesarios para controlar el correcto acceso a los métodos declarados. En el ejemplo, el método p no podrá ser utilizado hasta su implementación.

- **Tema 21**: ***Forward declaration***

Incorporar la posibilidad de declarar las clases con posterioridad a su uso.

Por ejemplo:

```
CLASS ca {      // Declaración de clase ca
   VOID m() {
      …
      }
   ...
   }
CLASS cc,
CLASS cd {
   cc c1,
   ca a1,
   VOID j() {
     c1.p(),   // Debe ser informado como error por el compilador
     a1.m(),   // Invocación correcta
   }
}
CLASS cc {      // forward declaration para clase cc
  INT z,
  VOID p() {
     …
     }
  …
}
```

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- **Tema 22**: ***Interfaces***

Ejemplo de declaración de interfaces, y su implementación.

Por ejemplo:

```
INTERFACE z {
   VOID q(...),
   VOID s(...),
  }

CLASS a IMPLEMENT z {
  VOID q(...) {
     …
     }
  VOID s(...) {
     …
     }
}
// Los métodos q y s sólo podrán ser invocados luego de su implementación.
```

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- **Tema 23**: ***Sobreescritura de métodos***

Una clase que hereda de otra puede sobreescribir métodos declarados en la clase de la que hereda, pero no puede sobreescribir atributos.

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- **Tema 24**: ***Sobreescritura de atributos***

Una clase que hereda de otra puede sobreescribir atributos declarados en la clase de la que hereda, pero no puede sobreescribir métodos.

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

Temas 25 al 28 Comprobaciones de uso de variables

El compilador debe chequear que el número máximo de niveles de anidamiento de funciones locales dentro de métodos de clase sea 1.

- **Tema 25**:
  Se debe incorporar a las reglas de declaración de variables, la posibilidad de que adelante del tipo aparezca la palabra reservada **CHECK** (incorporarla a la lista de palabras reservadas). Por ejemplo:
  ```
  CHECK INT a; b; c,
  ```
  El compilador debe chequear e informar, para cada variable declarada con la cláusula **CHECK**, si la misma aparece del lado izquierdo en más de una expresión de ámbitos diferentes y al menos una vez del lado derecho en cualquier ámbito.

- **Tema 26**:
  El compilador debe chequear e informar, para cada variable declarada, si no aparece en el lado izquierdo de al menos una asignación en el ámbito donde se declaró.

- **Tema 27**:
  El compilador debe chequear e informar, para cada variable declarada, si no aparece en el lado derecho de asignaciones en el ámbito donde se declaró, pero aparece en el lado izquierdo de un ámbito diferente.

- **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.

Temas 29 a 31: Conversiones

Para los tres temas: El chequeo de tipos y la incorporación de conversiones implícitas se efectuará durante la generación de código intermedio para Tercetos y Árbol Sintáctico, y durante la generación de código Assembler para Polaca Inversa

Tema 29: *Conversiones Explícitas*

- El compilador debe reconocer el uso de conversiones explícitas en el código fuente, que serán indicadas mediante la palabra reservada asignada en el TP02.

  ```
  TOF(<expresión>)      // para grupos que tienen asignado el tema 7
  TOD(<expresión>)      // para grupos que tienen asignado el tema 8
  ```

- Se debe considerar que cuando se indique la conversión **TOD**(expresión) o **TOF**(expresión), el compilador deberá generar código para efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. (**DOUBLE** o **FLOAT** respectivamente). Dado que los otros tipos asignados al grupo son enteros (1-2-3-4-5-6), el argumento de una conversión, debe ser de alguno de esos dos tipos.
- Sólo se podrán efectuar operaciones entre dos operandos de distinto tipo (uno entero y otro flotante), si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante mediante la conversión explícita que corresponda. En otro caso, se debe informar error.
- En el caso de asignaciones, si el lado izquierdo es de uno de los tipos enteros asignados (1-2-3-4-5-6), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos.

Tema 30: *Conversiones Implícitas*

- El compilador debe incorporar las conversiones en forma implícita, cuando se intente operar entre operandos de diferentes tipos (uno entero y otro flotante). En todos los casos, la conversión a incorporar será del tipo entero (1-2-3-4-5-6) al tipo de punto flotante asignado al grupo. Por ejemplo, el compilador incorporará una conversión del tipo **INT** a **FLOAT**, si se intenta efectuar una operación entre dos operandos de dichos tipos. En otras situaciones, se debe informar error de incompatibilidad de tipos.
- En el caso de asignaciones, si el lado izquierdo es de uno de los tipos enteros asignados (1-2-3-4-5-6), 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 asignado al grupo, y el lado derecho es de tipo entero (1-2-3-4-5-6), el compilador deberá incorporar la conversión que corresponda al lado derecho de la asignación.

Tema 31: ***Sin Conversiones***

El compilador debe prohibir cualquier operación (expresión, asignación, comparación, etc.) entre operandos de tipos diferentes, informando en cada caso, cuál es la combinación de tipos que está provocando la incompatibilidad.

Salida del compilador

1) Código intermedio, de alguna forma que permita visualizarlo claramente. Para Tercetos y Polaca Inversa, mostrar la dirección (número) de cada elemento, de modo que sea posible hacer un seguimiento del código generado. Para Árbol Sintáctico, elegir alguna forma de presentación que permita visualizar la estructura del árbol. Ejemplos:

Tercetos:

```
20  ( + , a , b )
21  ( / , [20] , 5 )
22  ( = , z , [21] )
```

Árbol Sintáctico:

```
Por ejemplo:
Raíz
     Hijo izquierdo
          Hijo izquierdo
          Hijo derecho
     Hijo derecho
          Hijo izquierdo
               …
          Hijo derecho
```

Polaca Inversa:

```
10 <
11 a
12 b
13 18
14 BF
15 c
16 10
17 =
18 …
```

2) Si bien el código intermedio debe contener referencias a la Tabla de Símbolos, en la visualización del código se deberán mostrar los nombres de los símbolos (identificadores, constantes, cadenas de caracteres) correspondientes.
3) Los errores generados en cada una de las etapas (Análisis Léxico, Sintáctico y durante la Generación de Código Intermedio) se deberán mostrar todos juntos, indicando la descripción de cada error y la línea en la que fue detectado.
4) Contenido de la Tabla de Símbolos.

Nota: No se deberán mostrar las estructuras sintácticas ni los tokens que se pidieron como salida en los trabajos prácticos 1 y 2, a menos que en la devolución de la primera entrega se solicite lo contrario.